User Interaction in Deductive Interactive Program Verification

Zur Erlangung des akademischen Grades eines
Doktors der Naturwissenschaften

von der KIT-Fakultät für Informatik
des Karlsruher Instituts für Technologie (KIT)

genehmigte
Dissertation

von

Sarah Caecilia Grebing

aus Mannheim

Tag der mündlichen Prüfung:	07. Februar 2019
Erster Gutachter:	Prof. Dr. Bernhard Beckert
Zweiter Gutachter:	Assoc. Prof. Dr. André Platzer

Contents

List of Figures and Tables

List of Listings

Acknowledgements

The journey of developing this thesis was accompanied by supervisors, colleagues, family and friends who contributed by supporting and encouraging me. I would like to take this opportunity to thank these persons.

First and foremost, I would like to express my great appreciation to my supervisor Prof. Dr. Bernhard Beckert, who gave me the opportunity to perform my research in his group and who supported me continuously during my time as a PhD student. Without his support and encouragement this thesis would not have been possible.

It was a pleasure working with my former and current colleagues Dr. Vladimir Klebanov, Dr. Simon Greiner, Dr. Christoph Gladisch, Dr. Daniel Grahl, Michael Kirsten and Mihai Herda at Karlsruhe Institute of Technology, as well as the members of the KeY project who always supported me with valuable feedback and discussions.

Among my colleagues, I am especially grateful for the support of two persons: one is Dr. Mattias Ulbrich, who collaborated with me in the development of the proof scripting language concept and the seamless interaction concept. The second person I am particularly grateful for his assistance and feedback is Alexander Weigl. He supported the realization of the concept for proof debugging and the integration of script-based interaction into the KeY system, as well as provided critical feedback and comments. Also, I would like to thank Associate Professor Dr. André Platzer for agreeing to be my second advisor and for his valuable feedback. I am much obliged to Prof. Dr. Uli Furbach for awakening my interest in logic and research in this area. I would like to offer my special thanks to Prof. Dr. Wolfgang Ahrendt for collaborating with me and giving me many good advices, as well as to Prof. Dr. Peter Schmitt for sharing his long-term experience and giving helpful feedback. I also wish to acknowledge the help provided by Dr. Ulf Schubert in sharing his experiences in conducting the user studies.

I would like to offer my special thanks to Dr. Florian Merz, Dr. Florian Böhl and Azadeh Shirvanian, who assisted during the studies in the data analysis and performed the moderation tasks, as well as to all participants of my user studies. I am also thankful to all students who worked with me, in particular Jonas Klamroth, Florian Lanzinger and An Thuy Tien Luong.

Dr. Thorsten Bormer deserves a very special gratitude. He has always been my constant support in every situation from the beginning on. With his patience and motivation he helped me through the discouraging phases of my time as a PhD student. Without him, this thesis would not have been written.

For the motivating words and continuous support of all of my friends I have made during my time in Karlsruhe and during my study in Koblenz I am very grateful. Finally, I wish to thank my family, in particular my mother, my brother and my sisters, as well as Elke and Reinhard for their support throughout my study.

This work is partially funded by the German Federal Ministry of Education and Research (BMBF) in the context of the Software Campus program.

Deutsche Zusammenfassung

Formale Methoden, wie beispielsweise Programmverifikation, können dafür eingesetzt werden die Zuverlässigkeit von Software zu erhöhen. In der Programmverifikation wird mit Hilfe eines Beweissystems geprüft, ob die untersuchte Implementierung formal spezifizierten Anforderungen genügt. Dieses Problem, ob ein beliebiges Programm eine nicht-triviale Eigenschaft erfüllt ist im Allgemeinen unentscheidbar – jedoch lässt sich durch die in den letzten Jahren erheblich verbesserte Leistungsfähigkeit solcher Beweissysteme dieses Beweisproblem für immer mehr Programme und Eigenschaften vollautomatisch lösen. Für *komplexe* Programme oder *komplexe* Eigenschaften ist dieses Problem dennoch nur mit einem bestimmten Maß an Benutzerführung lösbar. Der Benutzer muss dabei den Beweisprozess steuern, was einen hohen Aufwand bedeutet. Weiterhin ist das Finden einer für den Korrektheitsbeweis hinreichenden Spezifikation ein iterativer, zeitintensiver Prozess.

Die Kernidee dieser Arbeit ist es, die Benutzerführung in interaktiven Programmverifikationssystemen zu verbessern, indem Interaktionen in bestehenden Systemen mit Hilfe von Techniken aus dem Gebiet der Mensch-Maschine-Interaktion analysiert und auf der Basis der gewonnenen Erkenntnisse Verbesserungsansätze entwickelt wurden.

Die Grundhypothese dieser Arbeit ist, dass die Lücke zwischen der Beweisidee des Benutzers und des tatsächlich, teil-automatisch geführten Beweises eines der Hauptprobleme in der interaktiven Programmverifikation darstellt. Um ein besser benutzbares System zu entwickeln, muss diese Lücke während des gesamten Beweisprozesses möglichst klein gehalten werden.

Basierend auf dieser Hypothese ist ein Ziel der Arbeit, Benutzern interaktiver Programmverifikationssysteme auf verschiedenen Ebenen des Prozesses und des Problems Unterstützungen anzubieten: Zum einen werden hierzu verschiedene etablierte Interaktionsstile zur Beweisführung kombiniert, um die jeweils am besten geeignete Interaktion in Beweissituationen durchführen zu können, zum anderen werden dem Benutzer verschiedene Sichten auf das Beweisproblem angeboten und ihm der Wechsel zwischen diesen Sichten erleichtert. Diese Idee folgt dem Benutzbarkeitsprinzip, bei kognitiv anspruchsvollen Aufgaben Möglichkeiten zu bieten, sich verschiedene mentale Modelle des Problems zu bilden, um auf eine Lösung zu kommen. Als Grundlage dieser Arbeit dienen etablierte Interaktionskonzepte für die Beweisführung. Hierbei haben sich zwei unterschiedliche Vorgehensweisen durchgesetzt: die interaktive Beweisführung, in der der Benutzer während der Suche des mathematisch-logischen Beweises weitere Informationen zur Verfügung stellt (z.B. im KeY-System oder in Isabelle/HOL), sowie der annotationsbasierte Ansatz (z.B. in Systemen wie VCC oder Dafny), in dem der Benutzer alle Informationen für den Beweiser zu Beginn zur Verfügung stellen muss und die Beweisführung auf der mathematisch-logischen Ebene nicht für den Benutzer zugreifbar ist.

Bei der Interaktion auf der mathematisch-logischen Ebene haben sich wiederum zwei unterschiedliche Interaktionsstile etabliert, die sich in *direkte Manipulation*[1] und in den *skript-basierten* Interaktionsstil einteilen lassen. Systeme, die *direkte Manipulation* erlauben, bieten dem Benutzer die Möglichkeit mit Repräsentationen von Objekten der Aufgabendomäne durch Selektion der Objektrepräsentationen und Auswahl von anwendbaren Aktionen darauf zu interagieren. Systeme mit einem *skript-basierten* Interaktionsstil erlauben es dem Benutzer, Aktionen durch textuelle Eingaben bereitzustellen. Diese Aktionen werden auf der Basis des aktuellen Systemzustands und der Eingabe ausgeführt.

In dieser Arbeit wird der Ansatz verfolgt, dass, je nach Beweissituation und Beweisproblem, eine Benutzerführung auf der Ebene der Eingabeartefakte, d.h. Quellcode und Spezifikation, genauso wie auf der Ebene des mathematisch-logischen Beweises notwendig ist. Dabei ergeben sich zwei Herausforderungen: Der Benutzer muss bei der Interaktion auf *jeder* Ebene durch geeignete Interaktionsstile unterstützt werden. Weiterhin muss der *Wechsel* von einer Ebene zur anderen durch geeignete Hilfsmittel vereinfacht werden, damit der Benutzer den Überblick über den Beweis beibehalten kann.

Die wesentlichen Beiträge der Arbeit sind in zwei Teilbereiche aufgeteilt:

Qualitative, explorative Benutzerstudien

- Die Planung, Durchführung und Auswertung von zwei Fokusgruppendiskussionen zur Benutzung des Theorembeweisers Isabelle und des interaktiven Programmverifikationssystems KeY, um Probleme im Beweisprozess und in der Benutzung der Werkzeuge zu explorieren und zu vergleichen.

- Die Planung, Durchführung und Auswertung einer formativen Benutzerstudie mit Benutzern des KeY-Systems mittels eines teil-strukturierten Interviews und praktischen Aufgaben, um die Interaktionen der Benutzer während des Beweisprozesses zu analysieren und Meinungen zu einem prototypischen Hilfsmechanismus zu explorieren.

Interaktionskonzept für die Interaktive Programmverifikation

- Eine Skriptsprache für interaktive Programmverifikation aufbauend, auf den Erkenntnissen der Benutzerstudien, die auf die Bedürfnisse von interaktiven Programmbeweisen zugeschnitten ist (z.B. durch ein flexibles Matching zur Selektion von offenen Zielen).

- Ein Interaktionskonzept und dessen Umsetzung für eine Kombination aus direkter Manipulation und skriptbasiertem Interaktionsstil für Programmverifikationssysteme mit explizitem Beweisobjekt und direkter Manipulation als primären Interaktionsstil für die Beweiskonstruktion.

[1]In der englischen Literatur als *direct manipulation* bezeichnet.

- Ein Interaktionskonzept zur Fehleranalyse in Beweisen, welches die Analogie zwischen skriptbasierter Beweisfindung und Software-Debugging ausnutzt.
- Ein Konzept zur Beweisexploration für interaktive deduktive Programmverifikation, die es dem Benutzer erlauben soll, Hypothesen zum aktuellen Beweiszustand zu überprüfen, ohne dabei den Beweisfortschritt zu unterbrechen.
- Ein Interaktionskonzept, das die Erkenntnisse aus den Studien und den vorherigen Beiträgen der Arbeit aufgreift und die drei Interaktionsstile auto-aktiv, skriptbasiert und direkte Manipulation miteinander kombiniert. Dieses Interaktionskonzept ermöglicht dabei einen nahtlosen Übergang zwischen den verschiedenen Interaktionsstilen und den Repräsentationen des Beweisproblems.

Qualitative, explorative Benutzerstudien

Als Ausgangspunkt dieser Arbeit dienen drei explorative, formative qualitative Studien, die mit Probanden des interaktiven deduktiven Programmverifkationssystems KeY und mit dem Theorembeweiser Isabelle/HOL durchgeführt wurden (als Repräsentanten für den Interaktionsstil der direkten Manipulation, bzw. dem skriptbasierten Interaktionsstil). Während KeY es dem Benutzer erlaubt mittels direkter Manipulation den Beweis zu führen, ermöglicht Isabelle dem Benutzer eine skript-basierte Beweisführung. Mit den Studien wurde das Ziel verfolgt, das Vorgehen der Benutzer bei der Beweisführung in Systemen mit direkter Manipulation bzw. skriptbasierter Interaktion zu explorieren und gegenüberzustellen. Dabei stand weniger eine konkrete Beweisaufgabe im Vordergrund, stattdessen war der Gegenstand der Studien das allgemeine Vorgehen im Beweisprozess.

Es wurden zwei Fokusgruppen-Diskussionen durchgeführt, die das Ziel verfolgten, erste Erkenntnisse zur Aufgabendomäne und zu Problemen mit der Interaktion in interaktiven Beweissystemen zu erhalten. Dabei wurde eine Diskussionsgruppe mit Benutzern des Theorembeweisers Isabelle/HOL und eine Diskussionsgruppe mit Benutzern des KeY-Systems durchgeführt.

Zusätzlich wurde die Interaktion in bestimmten Phasen des Beweisprozesses für das KeY-System näher untersucht. Dabei wurden in einer dritten Benutzerstudie Benutzern des KeY-Systems verschiedene Situationen des Beweisprozesses präsentiert und die Aufgabe gestellt, mit dem Beweissystem zu interagieren. Zu den Aufgaben und dem KeY-System wurde des Weiteren ein teil-strukturiertes Interview durchgeführt.

Interaktionskonzept für die Interaktive Programmverifikation

Auf der Basis der Erkenntnisse aus den Fokusgruppen-Diskussionen und der Nutzerstudie für KeY wurde ein Interaktionskonzept erarbeitet. Das entwickelte Konzept hat zwei Hauptziele: zum einen soll der, in Programmverifikationsbeweisen sehr große, Beweiszustand strukturiert dargestellt werden. Zum anderen sollen die beiden zuvor untersuchten Interaktionsstile kombiniert werden, sodass Benutzer beide Stile zur Beweisführung nutzen können. Durch die Kombination eines skript-basierten Interaktionsstils mit direkter Manipulation wurde eine Analogie zwischen der Suche nach

Fehlern in Programmen und der Suche nach den Gründen für einen fehlgeschlagenen Beweisversuch deutlich. Diese Analogie wurde ausgenutzt, um etablierte Interaktionen aus dem Software-Debuggen auf das Feld der Programmverifikation zu übertragen und damit weitere Interaktionen zum skript-basierten Beweisprozess zur Verfügung zu stellen.

Um diesen Ansatz im KeY-System umzusetzen, wurde auf Basis der Benutzerstudien die Beweis-Skriptsprache KPS entwickelt. Im Sprachkonzept lag der Fokus darauf, die Besonderheiten von Programmbeweisen zu unterstützen. Darunter fallen beispielsweise die häufig vorhandene Vielzahl an ähnlichen Beweiszielen, die ähnlich behandelt werden können. Um Benutzern der Skriptsprache KPS eine Beweisführung zu ermöglichen, die den Mitteln der direkten Manipulation nahe kommt, wurden in den Studien beobachtete Aktionen der Benutzer im Anschluss an die Studien als textuelle Befehle in die Skriptsprache aufgenommen. Des Weiteren wurden durch eine Erweiterung der Sprache um Kontrollflussanweisungen die wiederholte Anwendung von Regeln vereinfacht, sowie durch spezielle Selektoren die Möglichkeit geboten, dass Benutzer ähnliche Beweisziele gleich behandeln können.

Das in dieser Arbeit entwickelte, kombinierte Interaktionskonzept wurde prototypisch im *Proof Script Debugger*, einem Interface für das KeY-System, umgesetzt. Erste Experimente zu Eigenschaften der Beweis-Skriptsprache KPS wurden unter Zuhilfenahme der prototypischen Implementierung durchgeführt. In diesen Experimenten konnten mit KeY durchgeführte, typische Programmbeweise erfolgreich in der Skriptsprache umgesetzt werden. Dabei konnte Verbesserungspotential für zukünftige Arbeiten identifiziert werden, wie etwa eine kombinierte Darstellung zwischen Skript und Beweisbaum zum Beweisverständnis.

Basierend auf den Erkenntnissen der Studien, sowie der in dieser Arbeit entwickelten Interaktionskonzepte, wurde abschließend ein Interaktionskonzept erarbeitet, das die drei prominenten Interaktionsstile für Programmverifikation kombiniert und so dem Benutzer flexible Interaktionsmöglichkeiten bietet. Dieses Konzept greift im Speziellen die Beobachtung aus den Studien auf, dass zwischen den verschiedenen Bestandteilen des Beweiszustands ein häufiger Wechsel stattfand, indem verschiedene Sichten auf das Beweisproblem nebeneinander dargestellt werden und der Wechsel zwischen diesen Sichten unterstützt wird.

Erkenntnisse zu den in dieser Arbeit entwickelten Konzepten und Lösungsansätze dienen dazu, zukünftig bessere Nutzerinteraktionen für Programm-Verifikationssysteme umsetzen zu können. Für zukünftige Arbeiten verbleibt vor allem die quantitative Evaluierung der hier entwickelten Konzepte und Lösungsansätze.

1. Introduction

There are many options to improve the quality of software systems by checking that the system fulfills its intended requirements. Among the most rigorous options is deductive program verification, providing a mathematical-logical correctness proof that the program fulfills its requirements given in a formal specification language. At the same time, software verification is often associated with large efforts [Kle+09a; Bau+12].

On the one hand, the progress that has been made in the area of automated verification tools in recent years, one example being systems based on SMT solvers [MB08], as well as the ever increasing processing power have improved the efficiency and effectiveness of verification tools such that many simpler software verification problems can now be solved almost automatically. On the other hand, even with these improvements, user interaction is not only still a necessity, but remains one of the main bottlenecks of real-world deductive program verification [Bau+12]. The user has to guide and control proof construction and find a suitable specification that is sufficient for the correctness proof in an iterative, time-consuming process. One promising approach to reduce the user effort in this interactive verification process is to analyze and improve on the usability of state-of-the-art verification tools.

One main hypothesis for this thesis is that there exists a gap between the proof idea and plan of a user and the actual proof in a verification system. We argue that this gap is one of the main obstacles for an effective and efficient interactive deductive program verification process and keeping this gap small during the verification process results in more usable deductive program verification systems.

In the iterative verification process, when the user did not provide sufficient guidance for proof construction, for example the right lemmas, the user will encounter an open proof by the verification system. In this situation the user needs to gain orientation and understand the steps performed by the system for this partial proof in order to be able to find the cause why the verification system was not able to find a complete proof. Knowing the cause enables the user to decide which next action in the process may be promising in order to proceed with the proof. Actions can be to either advance the proof by manually applying rules or instructing the system to proceed with the proof search or to change the program or its specification.

Two instances where gaining orientation in the proof process can be particularly difficult is when either the user has to abandon working within the verification system on the mathematical-logical level and return to changing the program and specification to be verified, or when the user has asked the verification tool to perform many automated steps. In both cases, the potentially large difference between the original proof state the user was familiar with and the newly resulting proof state can be cause for the difficulties in understanding the current proof situation.

The core of this thesis is to explore the aforementioned hypothesis and to improve the user interaction and user guidance in interactive deductive program verification systems. For this, we will first analyze the user interaction in state-of-the-art systems using methods from the area of human computer interaction, i.e., by performing user studies. Based on the analysis results and the insights gained during the analysis we develop improvements for the user interaction and user guidance in interactive deductive program verification.

Based on the user studies one goal of this thesis is to provide support for users of interactive deductive program verification systems on different levels of the proof process and of the problem: we integrate different established interaction styles to provide users with the appropriate style in every proof situation and we also provide different views onto the proof state and allow for a seamless change between the views such that users can choose the appropriate view in each situation. Providing different views follows one of usability principle identified by Easthaughffe [Eas98] for interactive theorem provers, which states that, for cognitively challenging tasks, different views onto the problem support users in building a mental model of the problem to be able to come up with a solution.

In this work, we will investigate some of the different interaction approaches for proof construction that have emerged in state-of-the-art verification systems – the interactive proof guidance, where users supply additional information during proof search on the mathematical-logical representation of the proof (such as in the KeY system [Ahr+16], KeYmaeraX [Pla18], KIV [Bal+00], PVS [ORS92], Coq [BC04] or Isabelle/HOL [WPN08]) and the annotation-based approach, where users provide all additional information before starting the proof attempt and have no access to and further influence on the mathematical-logical proof (e.g., in Dafny [Lei10], VCC [Dah+09] or OpenJML [Cok11]). For the interaction on the mathematical-logical proof two different styles are prominent for proof construction, the direct manipulation and the text-based interaction style. Systems that offer a direct manipulation interaction style allow users to interact on representations of objects of the task domain by selecting the representations and performing actions on them. Text-based systems allow users to interact with the system using a command language. The commands are then interpreted based on the input and the system state.

In this thesis, we follow the approach that depending on the proof problem and proof situation users need both – the interaction on the level of the proof input artifacts, i.e., the source code and its annotations, as well as on the mathematical-logical proof. Two challenges arise when trying to achieve this: Users need to be supported by appropriate interaction styles on *each* level they interact on and, additionally, the switch between these levels has to be supported such that users keep the overview over the proof while switching levels.

1.1. Structure and Contribution of this Thesis

We start with an introduction of the foundations of this thesis in Part I, where we give an introduction to interactive deductive program verification in Chapter 3. We especially focus on preliminaries concerning the KeY system, as KeY is used as showcase in Part II and Part III. We furthermore introduce preliminaries of the usability of software systems in Chapter 2 as necessary for the self-containment of this thesis.

The contributions of this thesis can be divided into two parts which is reflected by the structure of this thesis. In Part II we present our qualitative, explorative user studies where we explored the *context of use* of interactive verification systems. In Part III, we present interaction concepts we have developed based on the results of our user studies. The following list of contributions, in the order they are presented in this thesis, gives a brief overview of the main parts of this thesis before we briefly introduce Part II and Part III of this work subsequently.

Part II – Qualitative, Explorative User Studies

Chapter 4 describes work based on planning, conduction and evaluation of two focus group discussions: one for the interactive theorem prover Isabelle/HOL and one for the program verification system KeY. The goal of the discussions was to explore positive and negative aspects of the proof process, as well as the usage of both tools and to gain a first insight into the proof processes.

Chapter 5 presents the results of planning, conduction and evaluation of a user study using semi-structured interviews and practical tasks of the KeY system. The goal of the user study was to gain insight into the proof process and analyze the corresponding user interactions. To gain insights into the usage of and explore opinions on a mock-up version of a mechanism was another target of the user study.

Part III – Interaction Concepts for Interactive Deductive Program Verification

Chapter 7 introduces a concept for a proof scripting language based on results of the user studies and which tailored to the needs of interactive program verification and that is compatible with *direct manipulation* interaction.

Chapter 8 describes an interaction concept and a prototypical realization of this concept for the analysis of failed proof attempts in the process of interactive program verification based on techniques from software debugging systems.

Chapter 9 presents a concept for proof exploration in the interactive program verification system KeY based on the information gathered from the user studies.

Chapter 10 concludes with an integration of the three interaction concepts *autoactive, script-based* and *direct manipulation* for a program verification framework that allows for *seamless* user interaction.

1.1.1. Qualitative, Explorative User Studies

The starting point for this thesis are explorative, formative and qualitative user studies that have been conducted with users of the interactive deductive program verification system KeY and the generic proof assistant Isabelle/HOL.

The systems serve as representatives for the direct manipulation and the text-based interaction style. While KeY allows users to construct proofs using direct manipulation on a graphical representation of the proof, Isabelle/HOL allows users to interact text-based, by offering a structured command language.

The goal of the user studies was to explore and compare the proof guidance between systems with direct manipulation and systems with script-based interaction. We aimed to gain first insights into the task domain and to explore issues arising when using the target of evaluations. We therefore conducted a focus group for the Isabelle/HOL system as target of evaluation and one for the KeY system.

In a third user study we analyzed the user interaction in specific phases of the proof process in the KeY system in more detail. To gain this kind of insight, we conducted semi-structured interviews with practical tasks with users of the KeY system where the participants were shown different proof situations and asked to interact with KeY and *think-aloud* while interacting.

One more concrete working hypothesis in this thesis, which is directly linked to our hypothesis about the gap between the proof idea of the user and the concrete proof performed by the verification system, is that the analysis of unfinished proof attempts is one of the main time-consuming tasks in program verification. Therefore, we argue that this task needs improved user support to increase the efficiency and effectiveness of program verification systems. To be able to find evidence for or against our hypothesis and to be able to refine our hypothesis we conducted the user studies.

To be able to develop improvements for the user interactions in the proof process, we aimed to observe the individual proof processes of KeY users and investigate whether we are able to extract a common proof process. Also which different information the participants accessed during this verification process was of interest. Related is also the questions which intents do users have while performing actions in the verification systems and which decisions do the users have to make during the proof process.

Specifically, by observing the participants we wanted to determine whether participants switch their focus while proving programs correct, e.g., by switching between the details of the proof state and the program to be verified. In case of these focus switches, further information relevant for developing interaction concepts is in which parts of the proof process the switch occurs and which intents the participants have when switching focus.

To be able to support users in gaining the orientation in a partial proof, i.e., when encountering an unfinished proof attempt, it may also be possible to support the process by an improved presentation of information of the proof state. With the user studies we also intended to find clues on how to present proof state information appropriately.

By conducting focus groups with users of Isabelle/HOL and KeY we also expected to identify strengths and weaknesses of the different interaction styles and how interactive program verification systems offering an explicit proof object may learn from the other interaction styles without transforming solely to their counterpart.

1.1.2. Interaction Concept for Interactive Program Verification

Based on the knowledge gained in the user studies we have we developed an interaction concept for interactive deductive program verification systems. With this concept we aim to achieve two main goals: Firstly, to visualize the large and complex proof state in program verification proofs in a structured manner. And secondly, to integrate the main interaction styles that have emerged for interactive program verification systems to a consistent interaction concept such that users are able to use the style for proof guidance they consider promising in a given proof situation.

Combining the script-based and direct manipulation style allowed us to develop an analogy between the search for a defect in a program and finding the cause for a failed proof attempt. We use this analogy when combining both interaction styles by adapting well-known functionalities from the field of software debugging for interactive program verification and are therefore able to provide further interactions in the script-based proof process.

To realize this approach in the KeY system we have developed the proof scripting language KPS. The focus of the language concept is to support the peculiarities of program verification proofs, e.g., to be able to handle similar proof goals effectively. To allow users of KPS proof guidance that is similar to the proof guidance using direct manipulation, we included actions that were observable during the user study as textual commands into the language. Furthermore, we extended the language by control-flow structures and a flexible selection mechanism to simplify the repeated application of proof rules and the selection of similar proof goals. The combined interaction concept developed in this thesis was prototypically realized as *Proof Script Debugger*, a new user interface for the KeY system. First experiments that evaluate properties of KPS where conducted using this prototype. In these experiments, typical, existing proofs performed with KeY were successfully formulated as KPS scripts.

Another insight gained in the user studies was that the participants performed explorative actions throughout the proof process to determine how to continue with the proof. To support the users in these activities, we have developed a concept for proof exploration, together with a prototypical implementation. This concept allows the user to defer the original proof task to perform parts of the proof with, e.g., slightly changed assumptions. The provided user interface for the proof exploration mode aims to help the user in keeping track of the different exploration actions and also to adjust the proof guidance based on the insights gained via proof exploration.

Lastly, one conclusion of the observations in the user study was that switching between the different parts of the proof state and the program, together with its specification is a prominent activity that needs to be well-supported by verification systems to obtain a usable tool. We therefore developed a seamless interaction concept that is based on placing views for related parts of the proof state close to each other, together with mechanisms showing the relations between the components of the proof state. At the same time, the user interface deliberately restricts the amount of information presented by only showing two views onto the proof state at once. This allows the user to focus on details that are displayed in the current pair of views and to shift the focus by moving from one view to the next.

1.2. Previously Published Material

Some parts of this thesis have been either completely or in parts published before. The contents of Chapter 4 ("User Study with Focus Groups"), as well as the descriptions about focus groups in Chapter 2 ("Usability of Software Systems: Background and Methods") have been published in a modified version in two publications [BGB14b; BGB14a] by the author of this thesis, together with the co-authors Bernhard Beckert and Florian Böhl. Furthermore, the descriptions about the interaction styles in Chapter 2 ("Usability of Software Systems: Background and Methods") have been published in a publication [GLW18] by the thesis author together with the co-authors An Thuy Tien Luong and Alexander Weigl. Parts of Chapter 6 ("Summary and Conclusion") have also already appeared in three publications [BGB14b; BGB14a; BG15].

Chapter 7 ("Proof Scripting Language") is an extended version of a work co-authored by the thesis author together with Mattias Ulbrich and Bernhard Beckert [BGU17]. The results presented in Chapter 8 ("Proof Script Debugger") are a modified version of material presented in two publications [BGU17; GLW18], which are in large parts written by the thesis author. The modifications are adaptations to previous results of the realization of our interaction concept in a prototypical implementation.

The following list of publications are papers I co-authored and which are part of this thesis.

User studies and evaluations

2014 Bernhard Beckert, Sarah Grebing, and Florian Böhl. "How to Put Usability into Focus: Using Focus Groups to Evaluate the Usability of Interactive Theorem Provers". In: *Proceedings Eleventh Workshop on User Interfaces for Theorem Provers, Vienna, Austria, 17th July 2014.* Ed. by Christoph Benzmüller and Bruno Woltzenlogel Paleo. Vol. 167. Electronic Proceedings in Theoretical Computer Science. Open Publishing Association, 2014, pp. 4–13. DOI: `10.4204/EPTCS.167.3`

Bernhard Beckert, Sarah Grebing, and Florian Böhl. "A Usability Evaluation of Interactive Theorem Provers Using Focus Groups". In: *Software Engineering and Formal Methods – SEFM 2014 Collocated Workshops.* Lecture Notes in Computer Science. 2014

2015 Bernhard Beckert and Sarah Grebing. "Interactive Theorem Proving – Modelling the User in the Proof Process". In: *Workshop on Bridging the Gap between Human and Automated Reasoning - A workshop of the 25th International Conference on Automated Deduction (CADE-25).* Ed. by Ulrich Furbach and Claudia Schon. Vol. 1412. CEUR Workshop Proceedings. CEUR-WS.org, Aug. 2015. URL: `http://ceur-ws.org/Vol-1412`

Interaction concepts

2017 Bernhard Beckert, Sarah Grebing, and Mattias Ulbrich. “An Interaction Concept for Program Verification Systems with Explicit Proof Object”. In: *Hardware and Software: Verification and Testing – 13th International Haifa Verification Conference, Haifa, Israel 13-15, 2017, Proceedings.* Vol. 10629. Lecture Notes in Computer Science. Springer, 2017, pp. 163–178. DOI: 10.1007/978-3-319-70389-3_11

2018 Sarah Grebing, An Thuy Tien Luong, and Alexander Weigl. “Adding Text-Based Interaction to a Direct-Manipulation Interface for Program Verification – Lessons Learned”. In: *13th International Workshop on User Interfaces for Theorem Provers (UITP 2018).* Ed. by Mateja Jamnik and Christoph Lüth. To appear. 2018

Part I.

Foundations for this Thesis

2. Usability of Software Systems: Background and Methods

Contents

In the course of this thesis we use different methods to analyze the user interaction in interactive (deductive) program verification systems and their context of use. Based on these results we develop different interaction concepts.

Evaluations of program verification systems are already performed in verification competitions and by using verification benchmarks (e.g., [Bey17; Kle+11]). The focus of such evaluations is the effectiveness of the verification system, i.e., do the systems support the verification of certain types of verification problems. However, in the interactive verification process the user is also part of the verification loop. In this work the interaction between the user and the proof system is put into focus. To identify which aspects of the systems affect the user interaction of the system, different usability principles and evaluation methods from the area of *Human Computer Interaction (HCI)* can be applied. In this thesis we will build upon the basics of HCI to identify and sharpen our hypothesis about the user interaction in interactive deductive program verification. We will evaluate the context of use of program verification systems and the user interactions in these systems and we will develop concepts for functionalities that aim to improve this interaction. The models, methods and principles from HCI which are built upon in this work are introduced in the following.

2.1. Human-Computer Interaction

The ACM Special Interest Group on Computer–Human Interaction provides a working definition for *human-computer interaction* (HCI) in their "Curricula for Human-Computer Interaction":

> Human-computer interaction is a discipline concerned with the design, evaluation and implementation of interactive computing systems for human use and with the study of major phenomena surrounding them. [Hew+92]

Carroll [Car03] states precisely that HCI "is concerned with understanding how people make use of devices and systems that incorporate or embed computation, and how such devices and systems can be more useful and more usable."

Both quotes have in common that the focus lies on interactive systems and their users. Studying the interaction between humans and computer systems involves knowledge about the user (e.g., psychological and cognitive aspects as well as social aspects), the computer system (e.g., computer science and engineering) and the tasks the user wants to perform using the computer system (e.g., the context of use).

Models about users and their interaction with software systems are developed in this discipline, one example being Norman's Execution-Evaluation cycle [Nor13; Ben10] as a model about the user interacting with a system's user interface.

For the evaluation of the use of computer systems qualitative and quantitative methods are being adapted, improved and developed. They include questionnaire-based methods, as well as methods such as interviews or focus group discussions.

To design and develop interactive systems with a focus on usability, frameworks for design processes that involve users and design rules have been developed and proposed based on results from research and experience in this field.

2.2. User-Centered Process

The main goal of this thesis is to analyze the context of use and the user interactions in interactive program verification and to provide concepts to improve this user interaction. To achieve this goal we follow a *user-centered* approach.

It is state-of-the-art to design software systems using processes like agile development processes or waterfall. To develop systems that target usability as a system's property, a user-centered design process (UCD) is often followed, where the users and their needs are put into focus in each phase of the design and development process.

For example, Nielsen proposes the usability engineering life cycle [Nie93] that incorporates usability activities into the regular development life cycle of products. A more general approach of user-centered design is presented in the ISO standard 9241:210 [ISO10]. The standard provides a framework for a user-centered design process: Each phase is described, but leaving out the specific methods used in each phase. Benyon [Ben10] also provides a very abstract framework description for a user centered design process for interactive systems. All aforementioned frameworks have the common goal to design and develop a *usable* system by including the perspective of users into the design process. Furthermore, common in UCD is that several iterations of the process are performed until the final design is accomplished.

As an example for a very general framework for a user-centered design process, we will describe the abstract process presented by Benyon [Ben10] in more detail, by explaining the different phases in the process.

The user-centered design process consists of the four phases: *envisionment*, *understanding*, *evaluation* and *design*. This framework does not define an explicit starting point and the order in which the different phases are executed is also not relevant. Rather, it is considered that the process can be started in any of the phases. Central is the *evaluation* phase, where outcomes of the other phases are evaluated. Evaluations in this phase can be qualitative or quantitative evaluation methods or the judgment of artifacts by an expert in design. It is important that the methods used are suitable for the artifacts that are being evaluated.

In the phase *understanding* the context of use is analyzed. The main objective of this phase is to understand the requirements of the system that is to be designed. Here, the usage of already existing systems of the domain can be evaluated, as well affected users can be involved through discussions. The phase *envisionment* is concerned with the illustration of the design ideas for communication and evaluation purposes. These illustrations can for example include sketches, paper prototypes or mock-up prototypes.

In the *design* phase the system is designed conceptually, by developing an abstract concept about which goals should a system achieve, which functionalities should be covered by the system and which information should be presented to the user. In the conceptual design phase the components of a software system are described together with their relations to each other and their abstract interactions in a system.

In the *physical* design phase the abstract conceptual design is transformed into a concrete design. This includes defining the functional behavior of the software system and the data representation and data-flows in the system. Furthermore, the design of interactions using, for example, concrete tasks and activities, and the design of the *look and feel* of a system are parts of this phase.

2.3. Usability

As motivated above, a user-centered approach can be followed to design and develop a system that targets *usability* as a system's property. In the following we will give a definition for the term *usability* and introduce *usability principles*, which are general *design rules* that support designers of systems to make informed decision about design choices [Dix+04].

2.3.1. What is Usability?

In our context, usability is an attribute of a software system that contributes to the practical acceptability of the system, comparable with other desirable attributes like reliability or compatibility [Nie93]. The standard DIN ISO 9241-11:2018[ISO18] provides the following definition of usability:

Definition 2.3.1 (Usability)**.** *Usability is defined as the "extent to which a system, product or service can be used by specified users to achieve specified goals with effectiveness, efficiency and satisfaction in a specified context of use."*

The definition already states the usability of a system needs to be considered in the *context of use* of a system, product or service. As context of use, not only the technical context, such as the hardware or software of a system is considered, also the task domain, the user groups and the environment in which a system is used. The standard defines the context of use as follows:

Definition 2.3.2 (Context of Use [ISO18])**.** *Users, tasks, equipment (hardware, software and materials), and the physical and social environments in which a product is used.*

According to the standard, the three metrics *effectiveness, efficiency* and *satisfaction* are to be considered for usability. Other, more specific metrics are, for example, given by Nielsen, who provides the five attributes *learnability, efficiency, memorability, errors* and *satisfaction.*

The three metrics as defined by the standard are:

Definition 2.3.3 (Effectiveness [ISO18])**.** *Accuracy and completeness with which users achieve specified goals.*

The effectiveness refers to the property whether users can use a system to achieve the goals for which they intended to use the system. Effectiveness also includes the error rate and the recoverability from errors when users perform tasks with a system.

Definition 2.3.4 (Efficiency [ISO18])**.** *Resources expended in relation to the accuracy and completeness with which users achieve goals.*

Efficiency refers to the resources, e.g., time or effort, the users need to complete a specified task respectively reach a specific goal. Finally, satisfaction refers to the users' attitudes when using a system:

Definition 2.3.5 (Satisfaction [ISO18])**.** *Freedom from discomfort, and positive attitudes towards the use of the product.*

The attribute *satisfaction* already covers first aspects that are central in the more recent area of *user experience.* In the field of user experience, the goal is not only to obtain a product or system that is usable according to Definition 2.3.1 but in addition to create a positive experience for users before, during and after the use of a product. Usability is thus one important aspect of user experience.

2.3.2. Usability Principles

One way to obtain a usable system is adhering to general design rules which are based on psychological, sociological and computational theories. These design rules provide designers a practical guidance to make informed decisions about the design of a system.

Usability principles as one particular type of design rules are independent of their application area and described by Dix et al. [Dix+04] as being of “high generality and low authority”, i.e., while applicable to a wide range of systems the strict adherence to this kind of design rules is not imperative.

Usability principles are complemented by other types of design rules, differing in, e.g., their level of generality: *standards*, as specific rules with high authority or *guidelines*, which rank between principles and standards both in authority and generality.

As design rules may be conflicting, trade-offs between different principles have to be made when designing systems. For this, designers need knowledge to decide which consequences their choices have regarding the usability of their systems. [Dix+04]

In this work we have developed concepts for user support in interactive program verification systems. In these concepts we will refer to different usability principles, which we will briefly introduce in the following. If possible, the following description contains a quote capturing the essence of a principle together with the source we have taken them from which contains a detailed description of the usability principle. To apply a usability principle for a specific system, it has to be instantiated accordingly.

Flexibility “the multiplicity of ways in which the user and system exchange information” [Dix+04].

Substituitivity “Allowing equivalent values of input and output to be arbitrarily substituted for each other” [Dix+04].

Strive for Consistency “Consistent sequences of actions should be required in similar situations; identical terminology should be used in prompts, menus, and help screens; and consistent color, layout, capitalization, fonts, and so on, should be employed throughout. Exceptions, such as required confirmation of the delete command or no echoing of passwords, should be comprehensible and limited in number” [Ben05].

Consistency “Likeness in input-output behavior arising from similar situations or similar task objectives” [Dix+04].

Anticipation “Bring to the user all the information and tools needed for each step of the process” [Tog14].

Observability “Ability of the user to evaluate the internal state of the system from its perceivable representation” [Dix+04].

Visible System Status “The system should always keep users informed about what is going on, through appropriate feedback within reasonable time” [Nie94; MN90; Nie95].

Predictability “Support for the user to determine the effect of future actions based on past interaction history” [Dix+04].

Recognition rather than recall “Minimize the user’s memory load by making objects, actions, and options visible. The user should not have to remember information from one part of the dialogue to another. Instructions for use of the system should be visible or easily retrievable whenever appropriate" [Nie94; MN90; Nie95].

Customizability "Modifiability of the user interface by the user or the system" [Dix+04].

Gestalt principles are a set of principles (first formulated by Wertheimer [Wer23]) that describe the human perception of elements according to their spatial arrangement. One example for a Gestalt principle is that elements that are spatially arranged next to each other are recognized as belonging together.

2.4. Interactions

The following Section introduces preliminaries to interaction and is based on [Mac13; Ben10; Dix+04]. We will include some explanations of basic terms that will also be used throughout this thesis, alongside and adapt the original explanation of Dix et al. for the self-containment of this thesis.

We are focussing on the interaction between a human user and a computer system, where we will call the two actors *user* and *system* in the following. Communication is taking place through user interfaces which allow the system to communicate results to the user and allows the user to input data into the system for computation. An interface can be viewed as a translator between the two actors. Users interact with a system to accomplish a specific *goal* within a specific area, which is referred to as application *domain*. The application domain has its own concepts and terminology. For this thesis we will consider the application domain *deductive program verification*. Concepts or aspects of a domain may be represented as objects in a system's interface that a user can manipulate by using operations. These operations are called *tasks*. A *goal* in this context is then "the desired output from a performed task" [Dix+04].

Intentions are high-level actions of the user "that are required to accomplish a desired goal" and can consist of a sequence of more detailed actions. A simple example for a goal would be to "clean up a folder" on a computer system. One intention to reach this goal may be that the user decides to "delete a file". The actual sequence of actions is then to "select the folder, right-click on it and select the menu entry delete".

2.4.1. Models of Interaction

To be able to describe, understand and predict interactions, models have been developed based on research results in the fields relevant to HCI. The models are always a simplification of the actual behaviour, however, they can be used to identify sources of problems. Here, we will give two examples of interaction models: the *Norman's Execution-Evaluation Cycle* and the *Interaction Framework*. The models presented in the following are user-centric and interaction is described "in terms of *goals* and *actions* of the users" [Dix+04].

Norman's Execution-Evaluation cycle The execution-evaluation cycle (or also called *Seven stages of action* or *Norman's Action Cycle*) was developed by Don Norman [Nor13]

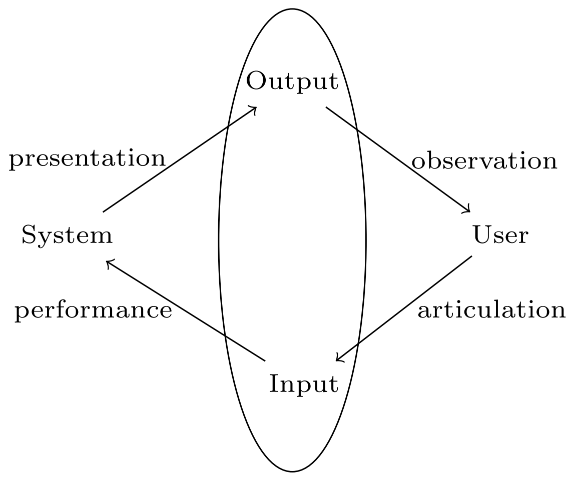


Figure 2.1.: Interaction Framework with transitions between components taken on [Dix+04]

and consists of seven stages of actions divided into two phases: The first phase starts with the user *establishing a goal* and based on this goal *forming an intention.* This intention is then refined into an *action sequence* that is suitable to reach the goal. The user then *executes the actions* which result in the manipulation of (domain) objects in the system. The second phase starts with the user *perceiving the system state* and is followed by *interpreting* the state according to his expectations or domain knowledge. The system state is then *evaluated with respect to the goals and intentions.*

Norman's model is suitable to be used to explain interaction problems with the two concepts *gulfs of execution* and *gulfs of evaluation.*

The gulf of execution refers to the problem that a system does not provide the actions a user needs to reach the intended goals, i.e., the system and the user's language differ. Moreover, the gulf of evaluation refers to the effort a user has when interpreting the presentation by the system in terms of his goals, i.e., the system's presentation and the user's expectations differ. Small gulfs lead to an effective user interaction. [Dix+04]

Interaction Framework Norman's model only focussed on the user's part of the interaction and neglected the system's part. To also capture this behavior Abowd and Beale [AB91] have developed a model capturing both the system and the user – the elements of this model and their connections to each other relevant for the interaction are shown in Fig. 2.1. The components of the model as shown in Figure 2.1, each having an own language, are the *system*, the *user* and the interface consisting of *input* and *output.*

In this model the different languages of each component need to be translated as part of the interaction between system and user, each translation belonging to one step of the interaction cycle. The translation between the user's goals and tasks into the input language is called *articulation* step. The user input is translated into the system's language in the *performance step.* The system in turn changes its state as triggered by the user's input. This resulting state needs to be *presented* in terms of the output's language. The user can then *observe* the output through the interface by translating the output language to the user's language and relating it to the user's goals. Using this model, problems with the interaction can be phrased and analyzed in terms of the translations considered in the interaction framework.

According to Dix et al. [Dix+04], the goals and the task to achieve this goal are phrased in the form of "psychological attributes that highlight the important features of the domain for the user". The translation in the articulation step is a mapping between these attributes to the input language. The interaction is simpler if the mapping is clear and the coverage of attributes in the language is high.

The *performance* is assessed by determining whether all system states can be reached via the translated input language in contrast to the direct access of the system's functionalities, e.g., via the system's API.

The system's state needs to be presented to the user in terms of the output language of the output device. The relevant parts of the system state need to be presented in terms of the output language such that differences between two system states can be perceived in the next stage.

After a system change a user *observes* the output and needs to interpret the results to evaluate them and relate them to the formed goal. The framework can be used to judge the usability of an interactive system regarding a particular task. Interactions are ineffective if one or more of the translations presented in the model is not an adequate mapping.

2.4.2. Interaction Styles

Interaction between a user and a software system is performed through the user interface. The way in which users can interact using this interface is hereby called the *interaction style.* There are different types of interaction styles, from which we will cover the basics of *direct manipulation* and *text-based interaction* in this section, as these two styles are of interest for this thesis. The following descriptions are mainly based on [Ben10; Ben05; Dix+04; Pre+94]

Command Language Interaction The command language interaction or sometimes also referred to as command line interaction (CLI) style is the classic text-based interaction style. Users formulate an action as a command followed by arguments, which can be for example options for the actions or textual representations of the objects that are being manipulated by this action. The action is only executed when users explicitly execute the commands and the resulting state is directly presented to the user. Throughout this thesis we will use the term *text-based* interaction interchangeable with the term *command line interaction.*

In this thesis we will also consider an extended text-based interaction style, which we will refer to as *script-based* interaction. The *script-based interaction* is a more sophisticated form of the textual interaction style. In addition to the commands which are present in the CLI interaction the user is able to use control flow structures to combine the commands to more complex actions. The effect of an action is usually presented to the user in a text-based fashion, often only perceivable after the execution of a whole script. Instead of repeatedly supplying proof commands when using CLI (e.g, in a batch mode or interactively by a user), in the script-based interaction style an interpreter decides, based on the proof and the script state, which actions to take.

Text-based interaction places the action into the center of the interaction and the objects that are being manipulated

Efficient interaction with the system by experienced users and the support for repetitions of actions are advantages of text-based interaction. Furthermore, especially when considering script-based interaction the history of performed actions is captured and always accessible by the user for inspection.

The downside of the text-based and script-based interaction is the high learning rate for the command languages, as well as the violation of the usability principle *recognition rather than recall*, where users need to recall the name and the syntax of the commands from their memory. In common integrated development environments this disadvantage is minimized by providing so called *auto-completion* features, for examples that users retrieve a list of matching commands while typing.

Direct Manipulation Interaction In the direct manipulation interaction style objects of the task domain have a visual representation in the user interface, which users can select and manipulate by performing actions on them. The central idea of the actions is that they are "rapid, reversible and incremental" [Sch83]. In contrast to the text-based interaction, where the actions are in the center of the interaction, in direct manipulation interfaces the objects which can be manipulated play the central role and actions that can be applied to or on the objects that are presented to the user upon object selection. This concept adheres the *recognition rather than recall* principle, as users do not have to remember which actions are applicable on a specific object, rather the actions are presented to the user.

Other advantages are that users can observe the immediate result of the selected action, which contributes to a visible system status, and the *easy recoverability from errors*, provided as part of the general reversibility of actions. Moreover, the possibility to observe the direct effects as well as to see which actions are available support novice users in learning how to use a system.

The downside of direct manipulation is the amount of objects of the task domain and their variety of types that need to be represented: a large amount of objects require a lot of screen space, the same holds for large objects that can not be depicted in a dense form [Sch83]. This property gives rise to issues such as that the user is overwhelmed with information and has difficulties finding the right object to manipulate if the representation is not carefully chosen.

Another general disadvantage of direct manipulation is a missing shortcut for repeatedly applying actions to several objects in the task domain.

2.5. Task Analysis

In this thesis one focus of the user study presented in Chapter 5 is to gain insight into how users perform a verification task using the program verification system KeY.

There exist a variety of methods to analyze and model the way users perform tasks. Common examples for methods to analyze (user) tasks are the Hierarchical Task Analysis (HTA) (e.g., described in [Ann03]), and GOMS (Goals-Operators-Methods-Selection Rules) [CNM83]. HTA is a representation of tasks in a structure chart notation. Tasks together with their subtasks and actions are structured hierarchically in the tasks models. Additionally, repetitions and alternatives can be expressed in these models [Ben10]. In the GOMS methods tasks are presented in terms of *goals, operators, methods* and *selection rules* [Ben10].

In this work, we will use a third method that shares similarities with plans in hierarchical task analysis [Ben10], called *sequence models* [BH98], which are used in contextual design to model observations of the way participants or users perform tasks in the course of, for example, an interview with practical work tasks. Besides sequence models, in contextual design also further models are created to capture all facets of the work that is being observed. For example, physical models to model the physical environment in which the observed tasks are executed or flow models to capture how work is divided between different people and how they communicate, e.g., by modeling different responsibilities and communication structures. According to Benyon [Ben10], contextual design is a modular method and for smaller projects a reduced version may be used.

We chose to use sequence models to capture our observations of participants accomplishing practical tasks with KeY in our user study (presented in Chapter 5). We believe sequence models allow a natural and flexible way to capture our observations.

2.5.1. A brief Introduction to Sequence Models

In the following we will give a brief introduction to sequence models and their components based on the descriptions by Beyer and Holtzblatt [BH98] and Benyon [Ben10]. *Sequence models* are a graphical representation used in contextual design to model the way tasks are performed from the perspective of an observer, for example during a user study. An example for a sequence model is presented in Figure 2.2. Each observed task and each participant in a study is captured in a separate sequence model. After completion of the study, the single sequence models are consolidated in a single, more abstract model to be able to recognize patterns and similarities and to be able to redesign work practice in a system.

Sequence models are built upon a sequence of *steps* that are performed by the user to achieve a certain *intent.* A set of steps is initiated by a *trigger.* Sequence models are derived from observing how and why tasks are being accomplished. The models vary in their level of detail according to the goal that should be achieved with the models.

Intents. An *intent* is the reason why a sequence of steps is being performed. A sequence model has a primary intent (in Fig. 2.2 the primary intent is to buy a drink),

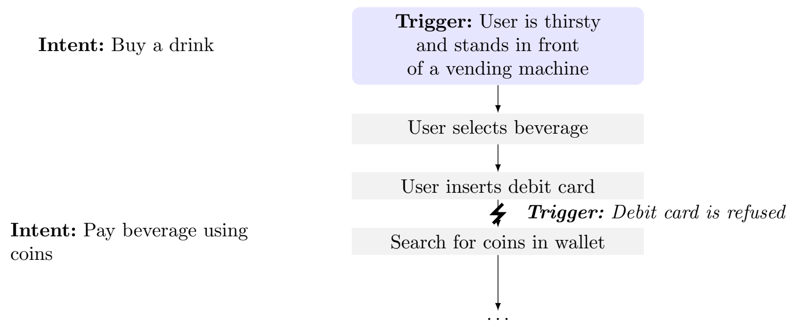

Intent: Buy a drink
Trigger: User is thirsty and stands in front of a vending machine
User selects beverage
User inserts debit card
Trigger: Debit card is refused
Intent: Pay beverage using coins
Search for coins in wallet
...

Figure 2.2.: Example sequence model showing first activities for the task to buy a drink from a vending machine.

which is noted at the beginning of the sequence. During the task completion secondary intents may become visible and are added as such to the model (e.g., the intent to pay the beverage using coins in Fig. 2.2). While primary intents are the reason for the whole sequence, secondary intents are the reason for a specific sub-sequence of steps [BH98]. Secondary intents are often added retroactively to the sequence model and are derived by inspecting sequences of actions. One example for such a retroactively added intent would be to add the intent ”User wants to pay beverage” to the step ”user inserts debit card” in Fig. 2.2.

Triggers. A *trigger* is an event that initiates a sequence of steps. When observing users of a software system triggers can be explicit prompts by the system for input. Other output by the system that is perceived by the user may be a trigger as well. In Fig. 2.2 a trigger is for example that the user is standing in front of a vending machine and is thirsty.

Steps. Steps in a sequence model can be thoughts or observed actions while a participant is accomplishing a task. The level of detail of steps varies depending on the purpose of the study. When aggregating sequence models to reveal common strategies, concrete steps are abstracted from detailed interactions with the user interface more towards purposes. Although not always expressed by participants, each action has a purpose. Steps may not always be sequential, but it can be observed that loops may occur or splits at decision points. If this is observed, sequence models can contain splitting at choice points as well as loops that connect steps. An example for a step in Fig. 2.2 is the selection of a beverage.

Breakdowns. If problems or disturbances arise during the task completion, *breakdowns* are added to the sequence model, which are indicated by a lightning symbol at the edges between steps and are labeled with a short description of the disturbance. A problem in the task depicted in Fig. 2.2 is that the debit card is refused by the vending machine. The disturbance in our model is also modeled as trigger, as the refusal of the debit card trigger that the user has to perform another step.

Aggregation of Sequence Models After capturing the actions of participants in single sequence models, the models are aggregated and abstracted to capture all workflows observed throughout the completion of a task. The steps are merged and abstracted from concrete steps towards the purpose of the steps. The result of the aggregation phase is a consolidated sequence model that contains abstract activities, possibly annotated with a set of triggers and intents, if visible.

The aggregated sequence model can now be used to identify problems in the course of task completion and to (re)design how tasks can be approached in a system. For example, whether an intent may be reached by partly automating actions or by leaving out steps that seam unnecessary. Furthermore, sequence models also help to identify alternative sequences, as well as activities that are unavoidable for task completion and therefore need to be included in a redesigned system. [Ben10]

In Chapter 5 we have used sequence models to understand the way users approach the task of verifying software together with their intents and strategies.

2.6. Evaluation Methods

As introduced in Section 2.2, the *evaluation* phase plays a central role in the user-centered process, where the outcomes of the other phases are evaluated using different *evaluation methods*.

The evaluation method has to be suitable to answer the questions of interest, as well as suitable for the target of evaluation, i.e, if no prototype exists, usability testing may not be the appropriate method to use.

Evaluation methods can be categorized along different properties, e.g., the types of results they produce, the phase in the research cycle the method is used in or the kind of information available to evaluate. Butz and Krüger [BK14] for example categorize methods along their purpose, the kind of results and their approach. Dix et al. [Dix+04] identifies at least eight factors for distinguishing or categorizing evaluation methods. In the following we will distinguish the methods along their goals and type of results (i.e., qualitative or quantitative) and along the phase in which the evaluation is performed within the user-centered design process (i.e., whether the evaluation method is summative or formative).

Qualitative and Quantitative Evaluation Evaluation methods can be distinguished as being either *qualitative* or *quantitative*. Quantitative evaluations produce objective, reproducible numerical results that can be related to each other and can be analyzed with statistical methods. A quantitative evaluation method is used to investigate a hypothesis that is formed a priori.

In comparison, qualitative evaluation methods are often inductive and are used to explore the research target, e.g., when there is no or only a first research hypothesis. The hypothesis is formed or adapted based the data collected during evaluation.

Qualitative evaluations are especially suitable if the subjective viewpoint of the participants is of interest and detailed insight into the research target is in the focus of the study. Data resulting from qualitative studies are statements of the participants that can not be easily analyzed using statistical methods, rather methods such as content analysis have to be used [Dix+04; KM05].

Formative and Summative Evaluation A formative evaluation takes place at the beginning or during the user centered design process. The evaluation goal is to explore issues, the context of use and to improve the evaluated system, i.e., the goal is to be able to *form* a product or system according to the evaluation results. Formative studies are often carried out on paper prototypes, wireframes and also a full functional prototype, if available. In contrast, a summative evaluation is carried out towards the end of the process to measure the overall usability or user experience. In contrast to formative evaluations, summative evaluations are performed on the full system and not on prototypes [Dix+04; BK14].

2.6.1. Questionnaires

Questionnaires are used to query users and depending on the property of the questions a questionnaire-based method can be either qualitative or quantitative: If questions are open-ended, the data that is collected through the questionnaire is of a qualitative nature, while closed questions in a questionnaire can result in quantitative data.

To measure user satisfaction or user experience standardized questionnaires exist. One example is the user experience questionnaire (UEQ) [LHS08] to assess the user experience of an interactive product. UEQ measures the following usability and user experience aspects: efficiency, perspicuity, dependability, originality, stimulation and novelty. Another example is the SUMI questionnaire which is used to assess the "quality of use" of a product by assessing the five usability dimensions *efficiency*, *affect*, *helpfulness*, *control* and *learnability* [Kir].

The questionnaire-based quantitative methods need a certain amount of participants to be representative, for example, in case of SUMI 12-20 participants are suggested, depending on the way they have been selected and how accurately they represent the user base.

The advantage of questionnaire-based methods using pre-existing, standardized questionnaires is the rather low effort in conducting the evaluation. However, for a very small user base these methods may only be partly suitable to obtain statistical significance. Furthermore, questionnaires are rather inflexible when posing open-ended questions and trying to explore what a participant really meant with an answer.

Also the data analysis for the quantitative parts of a questionnaire requires comparably low effort when compared with the analysis of qualitative data. Qualitative data needs interpretation and abstraction of the data set, before the results can be related to each other, compared and aggregated.

2.6.2. Interviews

A more flexible query method compared to questionnaires are interviews. In an interview an interviewer asks mainly open-ended questions. If the interview is structured the interviewer has a script that contains the questions that will be posed exactly as noted in the script. Semi-structured interviews allow for more flexibility: the interviewer still has a script with prepared questions, however the interviewer is allowed to explore topics raised by participants without having to exactly comply to the script. The participant's responses are recorded in order to be analyzed after the interview sessions.

Compared to questionnaire-based query methods, an advantage of semi-structured interviews is that the interviewer is able to react flexibly to the participants responses if an answer is unclear or more detail to an issue is desired.

Interviews can be enriched by adding tasks respectively scenarios – in our context, one possibility for including practical tasks is that participants may use a prototype of the system under evaluation. Enriching interviews with scenarios allows exploring opinions and expectations of participants towards the presented artifacts and highlights possibilities for improvement. While performing the tasks users can be motivated to *think-aloud*, i.e., to verbalize which actions they perform and *why* they are performing them. Moreover, the tasks together with the *thinking-aloud* methodology allows exploring how the participants perform the tasks and use the system. The intents for performing actions in the target of evaluation can be either perceived directly if users formulate them or can be asked for by the interviewer.

2.6.3. Focus Groups

Focus groups are a standard qualitative evaluation method in many areas to explore opinions about specific products or topics, e.g., in market research. In the field of human-computer interaction they are used for exploring user perspectives on software systems and their usability in an early stage of the usability engineering process [Fer+01; Nie93; Cap90]. In focus groups, it is possible to use stimuli or prototypes to explore requirements for a potential prototype or functionality.

Based on the results of the focus group discussions, (prototypical) mechanisms for improving usability can be developed, which can then be evaluated with methods such as usability testing and user questionnaires to quantitatively measure increases in usability. Focus groups provide the subjective experience of the users and require only a small number of participants (five to ten). The number of participants required to get significant results is much smaller than for quantitative evaluations, which makes focus groups well-suited for the relatively small user base of interactive verification systems. The duration of the discussion groups is approximately around one to two hours and it is guided by a moderator who uses a script to structure the discussion. Focus groups have three phases: pre-processing, performing the discussion and post-processing.

Focus groups require fewer participants than evaluations using questionnaires and the effort for conducting the discussion is less than that of one-on-one interviews [VH03; Ell+05]. Still, it is a non-trivial task to conduct a focus group. The discussions have to be well-structured as well as lively and open to be productive. And it is a challenge to steer the discussions towards the topics of interest without predisposing possible answers or biasing the results in other ways.

2.6.4. Preparation and Conduction of Focus Groups and Interviews

As we will present two user studies in this thesis, one using focus groups (presented in Chapter 4) and one using interviews with practical tasks (presented in Chapter 5) we will go into detail into the preparation and conduction of focus groups and interviews. Both methods require a preprocessing phase, a conduction phase and a post-processing phase which all share similarities. We will emphasize differences when necessary in the following. Otherwise, the presented phases apply to focus groups as well as interviews.

Pre-Processing Phase. Tasks in the pre-processing phase are recruiting the participants, preparing the script, recruiting and preparing moderators respectively interviewers, preparing the technical setup and possibly conducting pre-evaluations.

In general, the composition of the focus group as well as the participants for interview sessions should be representative for the user base of the tool being evaluated. But participants may also be selected from certain sub-groups, such as beginners or experts. Both the level of expertise in the relevant domain and the experience level for the evaluated tool are relevant criteria. For focus groups it is furthermore crucial to have a group of participants who are motivated and keen to debate.

The moderator respectively the interviewer must not be one of the stake holders and must be neutral in his or her opinion about the evaluated software. This excludes, for example, developers of the evaluated tool. Nevertheless, the moderator respectively interviewer must understand the issues that are discussed to a certain extent, to be able to ask for details. A well-prepared and experienced moderator can greatly improve the results of a focus group discussion.

The script for a focus group as well as the script for a (semi-) structured interview session contains all tasks and all questions for the study to conduct. Only neutral questions can be asked explicitly (e.g., “Please name one good and one bad feature of the tool.”). Non-neutral questions such as “Is feature X useful?” are included in the script but are not asked explicitly. Instead, it is the moderator’s or interviewer’s task to guide the discussion respectively the topic the participants is talking about in an interview into the direction of these questions, e.g., by digging deeper when a participant brings up a certain issue. Similarly, in the focus group the moderator has to carefully balance neutrality and the desire to steer the discussion in a certain direction. The topics in the script should build on each other in a meaningful way, e.g., from a general topic towards a specific topic [Cap90].

For focus groups, as technical setup, it is advisable to use two adjacent rooms, one for the discussion itself and one for observers, including the experimenters and some domain experts (e.g., developers of the ITP). The discussion should be recorded e.g., with at least one camera and several microphones and can then be transmitted live to the observation room. This setup has to be well tested beforehand as any technical problem can seriously effect the post-processing of the recorded discussion. It is useful to provide a feedback channel from the observation room to the moderator (using a headset) to give hints and provide relevant domain knowledge.

A similar setup can be used for an interview session. However, one important prerequisite for interviews with practical tasks is to have the prototype available (e.g., as program on a computer) and to have suitable recording possibilities (e.g., a screen cast software that records the participants movements on the screen), together with voice recording equipment.

Before the conduction of interviews and focus groups, participants are informed about the evaluation method and introduced to the interviewer or moderator. Participants also are informed about the data collection during the evaluation and how their collected data is anonymized and used. The participants furthermore get a privacy statement which they are asked to sign before the conduction of the user study (see A.1 for an example of a privacy statements we have used for our focus groups).

Conducting the Discussion. The discussion itself starts with a round-robin introduction of the participants and some small warm-up tasks, and it ends with a cool-down task that allows to summarize the content of the discussion. The main part consists of sub-discussions that are related to specific topics such as usability aspects, tool features, etc. Each topic is introduced by the moderator, possibly using example problems, mock-ups of new features or similar material. After the recorded part of the discussion ends, there should be time for questions and feedback from the participants and the moderator (even if that part is not recorded, it is useful to take notes).

Conducting the Interview. Similar to focus groups, interview sessions start with an introductory phase where the participants answer general questions regarding their experience with the topic and regarding other topics that help to classify the participant for the analysis phase. The warm-up phase then follows, where more general questions about the topic or target of evaluation are asked. Here, examples can be to ask questions where participants have to reflect on their course of actions of prior usages of the target of evaluation. The main part of the interview has to be structured meaningful for the participants and scenarios or practical tasks are introduced and approached. The cool-down phase may then contain more exploratory questions or questions regarding visions for a perfect system. Similar to focus groups, also after interviews, it should be planned to allow participants to ask questions and to explain the purpose of the user study, if not already performed at the beginning. During the interview conduction notes should be taken about key points, issues and other events considered important for the project.

Data Analysis. The post-processing phases of the conducted interviews with practical tasks and the focus groups is similar. In the post-processing phase the recorded material has to be transcribed, analyzed, and evaluated.

The first analysis step is to check if the participants conformed to the expected user types or whether the group has to be divided into sub-groups (e.g., beginners and experienced users). Given this grouping, opinions expressed within the focus group or interview sessions can be associated with their user type during analysis, if applicable. This association allows drawing first conclusions for each user type.

One method suitable for categorizing and extracting the information from the discussion is *qualitative content analysis* [May96; Kuc14]. Similarly to the classification of users, the material has to be categorized and opinions have to be assigned to the categories. The categories can either be based on the research question and prior research or theories underlying the research question, the questions asked during the discussion, as well as the opinions given by the participants [May00]. Depending on the basis it is distinguished whether categories are formed inductively, from the (transcribed) material under evaluation, or deductively, from prior knowledge respectively theories. Often a mixture of both methods is used, as we also did for our focus groups and interview sessions.

First, for each explicit and implicit question in the script, an own top-level category is defined, e.g., "Strengths of the system related to the proof process". Then the discussion or the interview sessions are analyzed and for each opinion related to the top-level, if a suitable subcategory already exists, the opinion is assigned to that subcategory. If not then a new subcategory is introduced and the opinion is assigned to this category. For example, assume that the subcategory "user interface" was already defined and an opinion of one of the participants is: "The user interface is great!". Then this opinion would be assigned to the subcategory "user interface".

During this analysis, it is important to remain objective, to take all stated opinions into consideration, and to avoid bias when interpreting what has been said. It is useful to involve several persons in this task, including the moderator or interviewer. When the material is categorized a revision of the categories may be done. For example some categories may be merged together to a larger or more abstract category.

After the categorization the opinions assigned to each of the subcategories have to be carefully analyzed and conclusions for the usability of this subcategory have to be drawn. This is a creative process and depends on the experience of the project members as well as the underlying tasks and research questions. It may be advisable to also take care which user type stated the opinion, as beginners often have different usability issues than intermediate or expert users. Moreover, for focus groups the reactions of the group should be taken into account, because an issue which the majority of the group agrees on might be an issue which the majority of the users in general might have as well. Attention should also be drawn to issues occurring with a higher frequency than others, regardless of the part or phase of the discussion they are expressed. There might be a correlation between the frequency and the relevance of an issue.

For practical tasks, such as the tasks we posed during our user study presented in Chapter 5 the interactions on the screen have to be analyzed. Here task analysis methods (as described in Section 2.5) suit well.

3. Interactive Deductive Program Verification

Contents

3.1. Introduction

The focus of this thesis is the deductive interactive program verification. In this research discipline formal methods are being developed that answer the question, whether a program is correct with respect to a given formal specification. The program and its specification are transformed into an appropriate formalization – one or more mathematical-logical formulas (also called *proof obligations*) – which are then proven using deductive proof methods.

In this chapter we introduce the basic concepts of tool-supported interactive program verification and especially focus on the interaction paradigms that have emerged in this area. Mechanized proof systems have been developed to support the user in the proof process for example by checking the applicability of proof rules or proof methods in the respective situation as well as by handling trivial cases automatically leaving the complex parts of the proof task to the users.

The user interaction of program verification systems is the focus of this thesis. We will use the KeY system as a showcase to explore the *context of use* (see Ch. 2) in interactive program verification system and demonstrate our interaction concepts developed in the course of this theses mainly using the KeY system.

The KeY system [Ahr+16] is an interactive verification system for Java programs annotated with the Java Modeling Language (JML). In the following we will give a brief introduction to the logical foundations of the KeY system. KeY's core consists of a theorem prover for the Java Dynamic Logic (JavaDL) and a sequent calculus for this logic for reasoning about JavaDL formulas [Ahr+14].

We start by introducing the basic notions of logical calculi before describing the specification language for Java programs that is part of the input into the KeY system. The annotated Java program is transformed into a proof obligation in Java Dynamic Logic, when loaded into the KeY system. We therefore briefly introduce JavaDL together with the sequent calculus that is used to reason about annotated Java programs. Alongside, we introduce the (formal) definitions of the structures that are needed in the course of this thesis. Following the logical prerequisites, we introduce the characteristics of the interactive program verification process, which is characterized by iterations of unfinished proof attempts. In these proof attempts users have to analyze unfinished proofs for the reason why the prover was not able to find a proof and interact according to the result of their analysis to proceed with the proof search respectively proof construction. To interact with verification systems different interaction styles have emerged which each have their advantages and drawbacks.

The different concepts that have emerged are being introduced in Sect. 3.6 alongside with a brief introduction of verification systems that implement these interaction styles.

3.2. Logical Calculi

Throughout this thesis, we will work with verification tools that use calculi to reason about properties like the validity or unsatisfiability of a given formula. The presented concepts and methods to improve the usability of verification tools are built for those tools based on calculi. Most of the concepts we describe are not tied to a specific calculus, but can be applied to a broad range of calculi, as long as the calculus fits into the generic structure presented in the following. We will first describe this structure informally, using the resolution calculus for propositional logic as an example, and will then give a more formal description of the general form.

Consider a resolution calculus for formulas in propositional logic. To determine whether a given formula ϕ is unsatisfiable, it is first transformed into a set of clauses (e.g., using an initialization function that transforms ϕ into conjunctive normal form and then generates a set of clauses). This set of clauses will be called *construct* when considering the general notion for a calculus. Another constituent of calculi are calculus rules, for the resolution calculus this set contains only a single rule r_{res}: the resolution rule 3.1, which combines two clauses C_1 and C_2 iff one clause contains a propositional variable P (also called literal) in positive form and the other clause contains this literal in negative form $\neg P$.

$$r_{res} \ \frac{C_1 \cup \{P\}, C_2 \cup \{\neg P\}}{C_1 \cup C_2} \tag{3.1}$$

To be able to detect when to stop applying calculus rules and to detect whether the formula to prove is unsatisfiable the resolution has so called *closed constructs.* In case of the resolution calculus the closed constructs are all those sets of clauses obtained from (repeatedly) applying the calculus rule to the initial construct that contain the empty clause.

In its general form, a calculus for a formal language (with words taken from the set *Fml*) is a four-tuple (K, i, R, c) with

- K the set of all elements the calculus operates on (further also called *constructs*),
- $i\colon \mathit{Fml} \to K$ the initialization function that maps a given formula to its initial construct,
- $R \subseteq K \times K$ the set of calculus rules,
- $c\colon \mathit{Fml} \to 2^K$ a function that maps a given formula to its set of closed constructs.

Given these constituents we can define the generic notion of a proof for a given formula ϕ as a finite sequence of constructs, starting with the construct initialized with ϕ and ending in a closed construct and where all constructs, but the first one, result from an application of a rule. Formally, a proof for the formula ϕ is a sequence $(s_0, \dots, s_n)$ where $s_i \in K$, $(s_i, s_{i+1}) \in R$, $s_0 = i(\phi)$ and $s_n \in c(\phi)$. We say a formula ϕ is derivable in a calculus ($\vdash \phi$), if there exists a proof for ϕ.

The concepts presented in this thesis only require that the underlying verification system operates with deductive rules, i.e., that it uses a *(rule-based) calculus.* In particular, this means that given a formula ϕ (from a set of well-formed formulas *Fml*, e.g., the set of all FOL formulas over a signature Σ) there has to be an initialization function that translates the formula into a representation the rule-based calculus can operate on: $i\colon \mathit{Fml} \to K$. This representation is taken from the set K of all possible elements that may arise in the calculus – e.g., sets of clauses in the resolution method. For example, in a sequent calculus, that is used to demonstrate the validity of the formula ϕ, the initialization $i_{Seq}(\phi)$ is the sequent with empty antecedent and the single formula ϕ in the succedent.

A rule-based calculus comprises a set of rules $R \subseteq K \times K$ that defines how to transform one construct into another. In this definition, rules applied to a construct result in exactly one construct – as a consequence, when viewed on this general abstraction level, proofs are linear. Nothing has been said yet about the inner structure of the constructs from the set K on which the rules operate on: one typical inner structure of constructs are trees and the application of a rule from the set R may introduce branches in the tree by extending the leaves of the tree.

Finally, to be able to identify when the application of rules has produced a construct that proves ϕ, the calculus comprises function that determines the *closed* constructs $c(\phi)$ for ϕ. For example, for a sequent calculus, irregardless of the formula ϕ, the function c maps to the set of all proof trees where all leaves are marked with axioms. Given a definition for the semantic consequence relation $\vDash$ of the underlying language of the calculus, we can define the two important notions of soundness and completeness. We will reduce the notion of soundness to the soundness of single calculus rules with the help of a function $v : K \to \{0, 1\}$ with the property that (a) for all formulas $\phi \in \mathit{Fml}$, the function $v(i(\phi))$ evaluates to 1 only if $\vDash \phi$ and (b) for all closed constructs $\gamma \in c(\phi)$, $v(\gamma)$ evaluates to 1.

We then call a calculus rule $r \in R$ *sound* iff for all constructs k_1 and k_2 with $(k_1, k_2) \in R$ holds that $v(k_1) = 0$ implies $v(k_2) = 0$. A calculus is called *complete* iff for all formulas $\phi \in \mathit{Fml}$ whenever $\vDash \phi$ holds there is a proof for ϕ (i.e., $\vdash \phi$) using the calculus.

We distinguish two kinds of calculi – the *synthetic* and the *analytic* calculus. For a formula ϕ to prove in an analytic calculus the set of closed constructs is independent from ϕ. For a synthetic calculus the set of initial constructs is independent from ϕ. Throughout this thesis, we focus on analytic calculi.

3.3. Specification of Java Programs with JML

The *Java Modeling Language* (JML) is the de-facto standard for the specification of Java programs, developed since 1999 [Lea+13].

It is a "formal behavioral interface specification language for Java" [LC05] that allows to specify the behaviour of Java units in terms of contracts and invariants using first-order logic and following the *Design by Contract* [Mey92] (DbC) concept. In the DbC concept, the behavior of software modules is defined in terms of contracts between calling (caller) and called (callee) modules. If the caller guarantees the properties required by the callee's contract, the callee guarantees the properties stated in its contract after execution, and therefore the caller can rely upon these properties. This concept enables modularization as the responsibility of checking these properties is divided between the caller and the callee. The caller has to check before the invocation of the callee that the required properties are established and the callee has to check upon termination that the guaranteed properties hold.

Java units that can be specified in JML are *methods, classes and interfaces* [Ahr+16]. In the following we will briefly introduce the specification of classes and methods, as needed for the self-containment of this thesis. We refer the reader to [Ahr+16; LC05; Lea+13] for a detailed introduction and description on JML. The JML specification of Java units can directly be added to the Java source files as specialized comments. JML annotations are enclosed in `/*@` and `*/` or with `//@`. In the annotations it is allowed to use every side-effect free Java expression, i.e., it does not change the heap, together with first-order logic constructs, such as quantifiers (`\forall, \exists`) or logical connectors (`==>` or `<==>`). We call each element of the JML annotation that starts with a keyword and ends with a semicolon a *clause*.

In the following we will use the term *annotations* interchangeably with specification. Alongside with the JML annotations we will furthermore introduce the distinction of different annotation types based on their purpose in Sect. 3.3.4, called *auxiliary* and *requirement* annotations [BBK11]. Requirement annotations, as a formalized version of the requirements to be verified, are the reason a verification attempt is performed. In comparison, auxiliary annotations only serve as guidance for the proof system in order to find the proof [BBK11].

3.3.1. Method Contracts

The behaviour of a method is specified in JML using method contracts. We will introduce the different parts of a method contract by the example in Fig. 3.1. The example is a modified version of a standard example shipped with the KeY system. It shows a Java class `Max` containing one integer field `max` and a method `compute_max(int[] a)`. This method expects an array `a` containing entries of type integer as parameter and searches for the maximal entry in this array. The method writes the result of its computation to the field `max`.

A method contract starts with the *behaviour case* (cf. line 5). The type of behaviour case indicates whether the specified behaviour applies to the method terminating normally (`normal_behaviour`) or to the method terminating with an exception (`exceptional_behaviour`). It is also possible to specify both behaviours in one single case, then the keyword `behaviour` is used. Preceding a behavioural case access level modifiers of Java can be used to specify the visibility of the annotation. Here, it is important to note that besides the Java access rules about the access of elements, JML only allows to use elements in a specification that are at least as visible as the specification itself.

Following the behaviour case, *preconditions* of the method (introduced using the keyword `requires`) are specified. In the preconditions the properties a caller has to fulfill are specified (cf. line 6). In our example, it is required that the array `a` only contains positive elements. All visible fields of the class and the method's parameters can be used in the precondition. If a method's contract does not specify a precondition, the default value `requires true;` applies.

A contract can also contain *postconditions* (introduced by the keyword `ensures`). The properties stated in the postconditions are the guarantees the callee gives upon termination of the method. The locations mentioned in the postcondition refer to the post-state of the method. It is possible in the postcondition to refer to the state of locations before the method's execution by using the keyword `\old()`. Expressions within an old clause are evaluated in the pre-state of the method. It is furthermore possible to refer to the return value of a method by using the special keyword `\result`.

The postconditions of the method in our example state that the value stored in the field max contains a value that is indeed greater or equal to each entry in the array (cf. line 8). A second postcondition states that, if the array is non-empty, the array must contain an entry with a value that coincides with the value stored in the field `max`. As with preconditions also for postconditions a default value applies (`ensures true;`), if no explicit postcondition is provided.

In the example we have seen that two *ensures clauses* have been specified. This means for the contract that the conjunction of both clauses has to be guaranteed. The same applies for the specification of several *requires clauses*.

In method contracts it is also possible to specify *framing conditions* using the keyword `assignable` (cf. line 7). Framing conditions address the *frame problem* [BMR95] and specify which locations on the heap are at most subject to change during the method's execution and may be changed after the method's termination. The locations mentioned in the assignable clause must exist in the pre-state of the method – expressions used in the assignable clause are also computed in the pre-state. In our example, the specification states that the method at most writes to the field `max`.

In JML it is possible to use the keyword `\nothing` in the assignable clause to specify that a method is not allowed to write to any pre-existing location. The default assignable clause is `\everything`, specifying the opposite. In the JML version of the KeY system there is a further specification for the assignable clause, `\strictly_nothing`, which specifies that a method not only is not allowed to write to pre-existing program locations but also is not allowed to create new locations.

When specifying that a method may terminate with an exception, the behavior case `exceptional_behaviour` can be used. In this case further keywords are allowed that specify the exception that can be thrown by the method. One example is `signals (E e) Post;` which specifies the postcondition `Post` of the method in case an exception `e` of type `E` is thrown. The variable `e` is then bound in the context of the postcondition [Lea+13].

In the method's declaration it is possible to use the modifier `/*@pure*/`, which indicates that the method is side-effect free, i.e., not changing the global state (which is the same as specifying `assignable \nothing`) – this information allows to simplify proof obligations for programs where this method is called.

3.3.2. Loop Invariants

In program verification there are different ways to handle loops in programs in the verification attempt. One possibility is to use *loop invariants* which are used to abstract from the concrete behavior of the different loop body executions. A loop invariant provides a declarative description of how an arbitrary loop iteration behaves, i.e., the properties specified in the loop invariant have to hold at the beginning of the first loop iteration, as well as after each loop iteration. In addition to the invariant, to prove the termination of a loop, a variant annotation can be given which allows to specify the termination measure for the loop.

A loop invariant is introduced in JML using the keyword `loop_invariant`. In Fig. 3.1, one example for a loop invariant formalizing seven properties of the loop execution is shown in lines 16 – 24.

The variant of a loop is specified using the `decreases` keyword. The variant has to be an expression of type `int` that is always greater than or equal to zero and strictly decreasing with each loop iteration. In our example, the variant is depicted in line 23. The expression `a.length - k` is decreasing with each loop iteration as the loop variable `k` is increasing and the length of the array `a` is not changing.

Listing 3.1: Example for a JML-Contract for the method `max()`.

```
class Max {

  int max;

/*@ normal_behaviour
  @  requires (\forall int i; 0 <= i && i < a.length; 0 <= a[i]);
  @  assignable max;
  @  ensures (\forall int i; 0 <= i && i < a.length; a[i] <= max);
  @  ensures (a.length > 0
  @      ==> (\exists int i; 0 <= i && i < a.length; max == a[i]));
  @*/
  void compute_max(int[] a) {
        max = 0;
        int k = 0;

      /*@ loop_invariant
        @   0 <= k && k <= a.length
        @   && (\forall int i; 0 <= i && i < k; a[i] <= max)
        @   && (k == 0 ==> max == 0)
        @   && (k > 0 ==>
        @        (\exists int i; 0 <= i && i < k; max == a[i]));
        @
        @  assignable max;
        @  decreases a.length - k;
        @*/
        while(k < a.length) {
          if(max < a[k]) {
              max = a[k];
          }
            k++;
        }
   }
}
```

In the JML version used in the KeY system, similar to method contracts, the framing conditions can also be specified for loops using the keyword `assignable`. The intuitive meaning is that during the execution of the loop-body at most the locations specified by the assignable clause can be changed.

3.3.3. Class Invariants

As introduced, a class is also considered a Java unit. The specification of Java classes is done in JML using class invariants. An invariant is a Boolean expression that is true in each visible program state. Constructors are forced to establish the invariant and callees can rely on the invariants when calling an object's method. As the invariants only have to be maintained during the visible program states, from the caller's perspective, it is allowed that an invariant is violated during a method's execution as long as it is established upon termination. To be able to specify that a method neither relies on the invariant nor establishes it upon termination, JML furthermore provides the keyword `helper` that can be added to the method's signature. For example, for private methods in a class that are used within other methods of this class.

3.3.4. The Purpose of Specifications

From the pragmatic's perspective, specification constructs can serve different purposes depending on the context the specification is used in. For example, the pre- and postcondition pair of a contract of a method that is being verified is called a *requirement annotation* as in this context its purpose is to specify the method's behavior towards its environment and the contract is the reason to perform a verification attempt [BBK11].

If the specification's purpose is to serve as proof guidance, we call this specification *auxiliary annotation*. We call a loop invariant an *essential auxiliary annotation* within the context of proving the contract of its outer method, as it is not part of the methods contract towards its environment and serves as a guidance for the proof search. However, it is *essential* for the existence of a correctness proof for the method, i.e., without these assertions no proof can be found independently of the resources allocated to the proof system. Also a pre- and post condition pair of the contract of a called method within the scope of a verified method is an essential auxiliary annotation. In contrast to essential auxiliary annotations, also *non-essential auxiliary annotations* can exist. Often these assertions are added to the program to guide the proof search [BBK11].

3.4. A Brief Introduction to Java Dynamic Logic (JavaDL)

In the KeY system, the proof obligation to prove that a method fulfills its contract is a formalization of the contract's meaning: If the caller satisfies the preconditions, the callee guarantees after its termination the postcondition.

To express and prove this property the KeY system implements the program logic Java Dynamic Logic (JavaDL) [BKW16] and a sequent calculus for JavaDL. The following introduction is mainly based on [Ahr+16].

Java Dynamic Logic is an extension of typed first-order logic that is extended by two modal operators $\langle . \rangle$ and $[.]$, that both can contain a sequence of Java statements. Like other modal logics, JavaDL formulas are interpreted in different worlds or states and the modalities define the transition relation between the worlds. The transition relation in JavaDL is defined by the Java program in the modalities, which describes a transition between computation states. The JavaDL formula $[p]\phi$ is true in a state if in every state reachable via the execution of the Java program p the formula ϕ holds. This includes that the formula is also true if p does not terminate. If p is a deterministic program, as it is the case for every Java program considered in our setting (i.e., without features such as concurrency), there is at most one state reachable via executing p. In contrast to the $[.]$ modality, a JavaDL formula $\langle p \rangle \phi$, which is equivalent to $\neg[p]\neg\phi$, requires that p terminates.

Example 3.1. An example for a valid JavaDL formula is

$$(\mathtt{x} \doteq 0) \rightarrow \langle \mathtt{x++;} \rangle (\mathtt{x} \geq 0),$$

which is expresses that if the execution of the statement `x++` is started in a state in which $\mathtt{x} \doteq 0$ holds then program terminates and in the post state of the program $\mathtt{x} \geq 0$ holds.

In JavaDL it is possible to express the proof obligation that a program p satisfies its contract as $\mathit{pre} \rightarrow \langle p \rangle \mathit{post}$. The formula *pre* denotes the conjunction of all preconditions of the JML contract of p formalized in JavaDL and the formula *post* denotes the conjunction of all postconditions of the JML contract of p formalized in JavaDL. The proof obligation above means that if a program starts in a state that satisfies *pre*, then upon termination of p, the formula *post* holds in the terminating state of p. This proof obligation is a simplified version of the one that is generated from JML contracts in the KeY system. A JML contract may contain more constructs than merely preconditions, postconditions, as briefly introduced in the previous section. A complete description of the translation of all JML constructs into a proof obligation in JavaDL, as supported by the KeY system, can be found in [GU16].

JavaDL formulas can contain two types of variables, *logical* and *program* variables. Universal and existential quantification is only allowed over logical, but not over program variables. Furthermore, while program variables may occur in terms and formulas, as well as the modalities, logical variables can never occur within a program. Our Example 3.1 only contains the single program variable `x` and no logical variables.

The version of JavaDL in the KeY system has another extension to first-order logic: *updates*, written as $\{u\}\phi$, where u is the update, ϕ an arbitrary JavaDL formula and $\{u\}$ the *update application*. An *update* is a special modality operator which is used to describe the effect of a Java program, i.e., the state transformations. The formula or term ϕ prefixed by an update application is evaluated in a state that is produced by the update application. Another viewpoint of updates is that they can be seen as delayed substitutions to the formula they are prefixing.

We can distinguish between *elementary*, *parallel* and *empty updates*. An elementary update application has the form $\{\mathtt{a} := t\}\phi$ and captures the effect of an assignment of the value of a term t to a program variable `a`. The right-hand side of the assignment must be a side-effect free, simple expression. A parallel update application $\{u_1 \parallel u_2\}\phi$ is a parallel composition of the updates u_1 and u_2 which are executed in parallel. Parallel updates follow the *last-win semantics*, i.e., if two update applications contain conflicting assignments to the same program variable the last update is applied. To denote an update with no effect JavaDL contains the *empty* update `skip`.

3.4.1. A Sequent Calculus for JavaDL

To prove that a method is correct w.r.t. its JML specification, in KeY we have to prove that the JavaDL formula resulting from the transformation of the JML contract and the program is valid. To prove the validity of this formula, KeY uses a *sequent calculus* that syntactically transforms a proof obligation by using logical inference rules.

The constructs KeY operates on are *sequent proof trees*, i.e., trees where the nodes are *sequents*. A sequent has the shape $\Gamma \Rightarrow \Delta$, where the (possibly empty) set of formulas Γ is called *antecedent* and the (possibly empty) set of formulas Δ is called *succedent*. The calculus rules KeY uses to perform deductive verification operate on sequents. These rules are applied to the sequents in the leaves of the proof tree, which is then extended by new sequents resulting from the calculus rule application.

A sequent $\Gamma \Rightarrow \Delta$, where Γ denotes the set $\phi_0, \ldots, \phi_m$ and Δ denotes the set $\psi_0, \ldots, \psi_n$ is valid iff the universal closure of the formula $(\phi_0 \wedge \ldots \wedge \phi_m) \rightarrow (\psi_0 \vee \ldots \vee \psi_n)$ is valid [Sch16]. Thus, the intuitive meaning of a sequent is: assuming all formulas in the antecedent hold, at least one formula of the succedent has to hold [AG16].

For proof construction we will use a sequent calculus where the rules have the general form (as introduced in Sec. 3.2)

$$\mathit{ruleName}\ \frac{P_1, \ldots, P_n}{C} \tag{3.2}$$

with C and $P_1, \ldots, P_n$ being sequents.

The sequents in the premisses and the conclusion may contain *schema variables* that have to be instantiated with concrete entities (e.g., formulas) to obtain a rule instance which can be applied to a sequent. The goal for proof construction using sequent calculus is to apply calculus rules until a closing rule can be applied to each sequent.

Example 3.2. Examples for calculus rules are the *impRight* and the *cut* rule (in the following, ϕ and ψ are schema variables that stand for arbitrary formulas, in our example for first-order formulas).
The rule *impRight* is a rule that breaks up a formula in the antecedent containing an implication as the top-level operator, resulting in a single proof branch. Intuitively, to prove that $\phi \rightarrow \psi$ holds we can also add ϕ to our assumption

and have to prove that ψ follows from our assumptions:

$$\mathit{impRight}\ \frac{\Gamma, \phi \Rightarrow \psi, \Delta}{\Gamma \Rightarrow \phi \rightarrow \psi, \Delta} \tag{3.3}$$

The second rule, called *cut*, can be used to "add" a formula to the sequent. In one of the resulting branches this formula has to be proven to be derivable from the formulas already present on the sequent ($\Gamma \Rightarrow \phi, \Delta$) – in the second branch the formula can then be used as assumption for the remaining proof:

$$\mathit{cut}\ \frac{\Gamma \Rightarrow \phi, \Delta \qquad \Gamma, \phi \Rightarrow \Delta}{\Gamma \Rightarrow \Delta} \atop \text{provided } \phi \text{ is a ground formula} \tag{3.4}$$

The *cut* rule is a rule without a schema variable, besides the schema variables for the context Γ, Δ, in the conclusion, however the premisses contain schema variables that have to be instantiated. The instantiation of the cut rule generally requires user input during proof construction.

In the context of proof construction, the rule's meaning can be read as: to prove the validity of a sequent matching the conclusion of a rule, it suffices to prove the validity of all premisses.

To prove the validity of a formula ϕ we start with the sequent $\Rightarrow \phi$ as initial proof obligation, i.e., ϕ has to be shown to hold without any further assumptions or alternatives (hence Γ is empty and Δ does not contain other formulas besides ϕ in the general sequent definition $\Gamma \Rightarrow \Delta$). To construct a proof for this proof obligation, a calculus rule has to be found where a matching instance for the conclusion of that rule can be created such that it matches the sequent. In that case the sequent is replaced by the rules instantiated premisses (see Example 3.3).

Example 3.3. As an example, we want to apply the rule *impRight* introduced in the Example 3.2 to the sequent $\Rightarrow (p \rightarrow q) \rightarrow \neg q \rightarrow \neg p$.
Given the definition of the *impRight* rule:

$$\mathit{impRight}\ \frac{\Gamma, \phi \Rightarrow \psi, \Delta}{\Gamma \Rightarrow \phi \rightarrow \psi, \Delta},$$

the schema variables ϕ and ψ of the rule are instantiated by $(p \rightarrow q)$ respectively $\neg q \rightarrow \neg p$ and the sets of formulas Γ and Δ are instantiated with the empty set, resulting in the following rule instance:

$$\mathit{impRight}\ \frac{(p \rightarrow q) \Rightarrow \neg q \rightarrow \neg p}{\Rightarrow (p \rightarrow q) \rightarrow \neg q \rightarrow \neg p}.$$

Thus the sequent after rule application has the form $p \rightarrow q \Rightarrow \neg q \rightarrow \neg p$, which is now the next proof obligation.

As already mentioned while explaining the rule schema for a calculus rule, rules in the sequent calculus may have more than one premise. When consecutively applying calculus rules, starting from a single sequent as the proof obligation, the resulting proof object takes the form of a *proof tree.*

Definition 3.4.1 (Proof Tree (based on [Ahr+16]))**.** *A proof tree is a tree such that:*

1. *Each node is labeled with a sequent.*
2. *Leaves are either labeled with a sequent or the symbol* $*$*, marking a closed leaf.*
3. *If an inner node* n *is annotated with* $\Gamma \Longrightarrow \Delta$ *then there is an instance of a rule whose conclusion is* $\Gamma \Longrightarrow \Delta$ *and the child nodes of* n *are labeled with the premise or premises of the rule instance (one premise per child).*

A branch of the proof tree is called closed if its leaf is labeled by $*$. A proof tree is called closed if all its branches are closed. All other leaves are called *open.* A proof tree with at least one open branch is called a partial proof tree. Open leaves of the proof tree are also called *proof goals* in the following. The root's sequent is also referred to as the *(original) proof obligation* in this thesis.

The relation to the general constituents of a logical calculus (see Sec. 3.2) for the sequent calculus for JavaDL is identical to the one for a general sequent calculus which was used as an example in section 3.2: The set of all constructs K is the set of all proof trees that are formed according to Def. 3.4.1. To demonstrate the validity of a formula ϕ, the initialization function produces a proof tree with a single node: a sequent with empty antecedent and the single formula ϕ in the succedent. The closed constructs provided by function c maps the formula ϕ to the set of all proof trees where all leaves are marked with axioms where the root node is identical to the value of the initialization function for ϕ. Lastly, the set R of calculus rules are the sequent calculus rules for JavaDL.

Soundness and Relative Completeness of the sequent calculus for JavaDL Two essential properties of a calculus are *soundness* and *completeness.*

Definition 3.4.2 (Soundness [Ahr+16])**.** *A sequent calculus* C *is sound if only valid sequents are derivable in* C*, i.e., if the root* $\Gamma \Rightarrow \Delta$ *of a closed proof tree is valid.*

A sequent calculus is complete iff whenever a sequent is valid, it is derivable in the calculus. A sequent calculus for JavaDL cannot be both sound and complete – if there was a complete and sound JavaDL calculus this would amount to solving the undecidable Halting problem. Instead, the weaker notion of *relative completeness* is used to characterize the JavaDL sequent calculus, which states that, given an oracle that provides all valid propositions of first-order logic with integer arithmetic, which the could then be added as assumptions, the calculus would be complete:

Definition 3.4.3 (Relative Completeness [Ahr+16])**.** *If a sequent* $\Gamma \Rightarrow \Delta$ *is valid, i.e., the formula* $\wedge\Gamma \rightarrow \vee\Delta$ *is logically valid, then there is a finite set* Γ_{FOL} *of logically valid first-order formulas such that the sequent*

$$\Gamma_{FOL}, \Gamma \Rightarrow \Delta$$

is derivable in the JavaDL calculus.

The sequent calculus for JavaDL in KeY is sound and relatively complete [Ahr+16].

3.4.2. Symbolic Execution

The descriptions of the sequent calculus up to now did not address programs in the modalities yet. To build a proof for a proof obligation formalized in JavaDL, KeY's sequent calculus for JavaDL also contains rules for each Java statement that is supported by KeY. A common operation of sequent calculus rules is to reduce the reasoning of a complex formula to reasoning about simpler formulas, and in case of Java programs, to transform programs in the modalities into simpler ones. The strategy to apply rules that step-wise reduce Java programs in the modalities is called *symbolic execution*.

The rules working on programs are always applied to the *active statement* p in the modality $\langle \pi p \omega \rangle$ (where π is called the nonactive prefix, which contains sequence of symbols such as opening braces `{` or beginning constructs such as `try{`) and ω is called the the rest) [Ahr+16]. The following example illustrates the notions of nonactive prefix π, respectively the program rest ω – the active statement in this case is `x_1 = x > 7`:

$$\overbrace{\texttt{try\{}}^{\pi}\,\texttt{x\_1 = x>7;}\;\overbrace{\texttt{if(x\_1) x++; else x--;\} finally\{y = 5;\}}}^{\omega}$$

The symbolic execution strategy in KeY contains three techniques to handle programs in modalities: unfolding, updates and case distinctions.

Unfolding. By unfolding a program in a modality, the program statements are transformed into simpler expressions that are semantically equivalent to the original program. Fresh local variables are generated in the course of program unfolding that capture the computation results by assignments.

Example 3.4. For example, the program in the sequent

$$\Rightarrow \langle \texttt{try\{ if (x>7) x++; else x--; \} finally \{ y = 5; \}} \rangle x > 0$$

is transformed to a semantically equivalent program

$$\Rightarrow \langle \texttt{try\{ bool x\_1; x\_1=x>7; if(x\_1) x++; else x--; \}}$$
$$\texttt{finally\{ y=5; \}} \rangle x > 0.$$

To simplify reasoning about the possibly complex if-condition the intermediate computation result of the evaluation of the expression `x > 7` is stored in a fresh local variable `x_1`. In the next steps this Boolean expression can be simplified or evaluated by other calculus rules.

Updates. To apply state transitions defined by the symbolically executed program to the JavaDL formula following a modality, updates are used in the KeY system instead of using a syntactic substitution [BP06]. Using updates as semantic substitution mechanism also takes care of aliasing of objects which cannot be properly handled by simple syntactic substitution alone. Another option would be to use case distinctions to capture the two different cases whether two objects alias or not when evaluating state updates. A further advantage of updates is to postpone applying state changes to formulas as they may be simplified beforehand and thus may simplify the proof.

Example 3.5. For example, consider the symbolic execution of the following sequent:

$$\Rightarrow (x = 0) \rightarrow \langle \texttt{x++;} \rangle (x >= 0).$$

This sequent is handled in the JavaDL calculus through a transformation by unfolding and removing the variable declaration of `x_1`:

$$\Rightarrow (x = 0) \rightarrow \langle \texttt{x\_1 = x+1; x=x\_1;} \rangle (x >= 0).$$

In the next steps the active statement is then transformed to an update resulting in the following sequent:

$$\Rightarrow (x = 0) \rightarrow \{\texttt{x\_1 := x+1;}\} \langle \texttt{x=x\_1;} \rangle (x >= 0)$$

After the rest of the program in the modality is symbolically executed, the resulting updates are directly in front of a simple expression:

$$\Rightarrow (x = 0) \rightarrow \{\texttt{x\_1 := x+1;}\}\{\texttt{x := x\_1}\}(x\_1 >= 0).$$

At this point, i.e., when the modality is empty and therefore program's effect is fully transformed into updates, the updates can be applied to the term $x >= 0$, with the following sequent as result:

$$\Rightarrow (x = 0) \rightarrow (x + 1 >= 0).$$

Deferring the application of updates has the benefit of being able to simplify several accumulated updates, before applying their effects to a term.

Case Distinctions. Some Java expressions may have side effects when executed. For example, when accessing fields of an object reference, if the reference is null, a `NullPointerException` is thrown in Java. To be able to reason about programs containing such statements, a case distinction is made in the proof, in this case, whether the accessed object is null or non-null.

Example 3.6. As an example for a case distinction we will take an example from Ahrendt et al. [Ahr+16].
In the sequent

$$\Rightarrow \langle\ \texttt{v} = \texttt{o.next}; \texttt{v.prev} = \texttt{o};\ \rangle\ \texttt{o.next.prev} \doteq o$$

the value of the field `next` of the object `o` is assigned to the variable `v`. Here, the object `o` may be null, thus applying the next applicable symbolic execution rule results in a case split whether `o` is `null`, and thus in a splitting proof. The two sequents resulting from this case split are:

$$\mathtt{o}\doteq\mathtt{null} \Rightarrow \{\mathtt{v} := \mathtt{o.next};\}\langle\ \mathtt{v.prev} = \mathtt{o};\ \rangle\ \mathtt{o.next.prev}\doteq o$$

$$\mathtt{o}\not\doteq\mathtt{null} \Rightarrow \langle\ \mathtt{throw}\ \mathtt{new}\ \mathtt{NullPointerException}();\ \rangle\ \mathtt{o.next.prev}\doteq o$$

3.4.3. Taclets

In KeY calculus rules are implemented by so called taclets [RU16] which work on a distinguished formula or term called the *focus* of the rule. For KeY rules we can thus concretize the general rule schema shown in Eq. 3.2 on page 38 as follows, where $\Gamma, \Delta, A, B, G_1, \ldots, G_n$ denote possibly empty sets of formulas and f denotes a schema variable denoting terms or formulas:

$$\mathit{ruleName}\ \frac{\Gamma, G_1, A \Rightarrow B, G_2, \Delta \quad \ldots \quad \Gamma, G_{n-1}, A \Rightarrow B, G_n, \Delta}{\Gamma, f, A \Rightarrow B, \Delta} \tag{3.5}$$

In this rule schema $\Gamma, \Delta, A, B, G_1, \ldots, G_n$ are *schema variables.* In KeY, the sets Γ, Δ are called *context.* The schema variable f is called the *focus* of the rule. In the example rule schema above, for simplicity, we have only depicted the case where the focus f is a formula, in general the focus may also refer to a subterm. The focus of a rule is the part of the sequent that may be changed by the rule. In the general rule schema in Eq. 3.5 the focus of the rule is in the antecedent. Other rules may have their focus in the succedent, i.e., $\Gamma, A \Rightarrow f, B, \Delta$. Irregardless, rules always have at most one focus. Some rules do not have a focus at all, for example the *cut* rule (as shown in the Example 3.2).

Taclets are a domain-specific language to express calculus rules in the KeY system that is used to describe where a rule is applicable on a sequent, e.g., in the antecedent or succedent. The conditions under which a rule can be applied, e.g., which other formulas are required that a taclet is applicable, and how the sequent is modified after rule application, e.g., which formulas are added, or how formulas are replaced. In this section we will briefly introduce the general ideas of taclets using examples, a more detailed description can be found in [Ahr+16].

Taclets can contain a *find* clause, starting with the keyword `\find` followed by a *pattern.* In the Example 3.7 below, the find clause contains the schematic formula standing for an implication `==> b -> c` in the succedent.

The *pattern* in the find clause defines the occurrence of terms and formulas where a taclet can be applied to (the *focus* of the taclet or rule). In the example, the rule is applicable to each top-level implication in the sequent's succedent. As already mentioned, in KeY the rules only contain one focus.

There can be three different kinds of patterns in a find clause: *schematic sequents*, *a formula* or *a term*. While formulas or terms denote arbitrary (sub) formulas respectively terms on the open goal, the schematic sequents also contain information about the position of the formula or term relative to the sequent arrow, i.e., whether its occurrence has to be in the antecedent or succedent.

Some taclets do not contain find clauses, for example the taclet for the *cut* rule or axioms.

Example 3.7. The calculus rule *impRight* of our example is implemented as taclet in the following way:

```
impRight {
  \find(==> b -> c)
  \replacewith(b ==> c)
  \heuristics(alpha)
}
```

Taclets may contain *assume* clauses (also called *context assumptions*) which are specified by the keyword `\assume`. If assume clauses are present, the taclet can only be applied if the formulas defined in the assume clauses are present on the sequent together with the formula defined in the find clause. In contrast to the formula in the find clause, the formulas in the assume clauses are not subject to change within a rule application and may also contain more than one pattern.

To define how the rule application modifies the goal so called *goal templates* are used. We will introduce two important kinds of goal templates: the *replacewith* clause and the *add* clause.

The replacewith clause (started with the keyword `\replacewith`) can only be used with a find clause and specifies how the focus is modified to obtain a new goal. In our Example 3.7 the implication in the succedent that is matched by the find clause is replaced by adding the premise of the implication to the antecedent and the conclusion of the implication to the succedent.

Instead of altering a goal it is also possible to add formulas to the goal, which can be achieved by using *add* clauses in the taclet (starting with the keyword `\add`).

To allow rules to create more than one new goal (i.e., to split the proof), more than one goal template is used in a taclet. Each goal template is then separated by a semicolon.

Although many rules are expressible in the taclet language, in the KeY system some rules are *built-in* rules, e.g., rules that require more complex computations. One example for a built-in rule is the rule to use a method contract instead of expanding a method's body when verifying a Java program containing a method call. Applying this rule requires to find the corresponding method contract in the verification system programmatically.

3.5. Proof Process for Deductive Program Verification

To prove properties of programs using a verification system, different *proof artifacts* may interplay during the verification process (in the following also called proof process). At the beginning of the verification process the user has to express the properties of a software system using a formalism understood by the verification system. During the proof the user then has to interpret the results obtained by the verification system. We will call the input into the verification system proof *input* artifacts throughout this thesis. In our case proof input artifacts consist of a software system, together with its specification given in form of annotations. Other proof artifacts are, for example, the *proof state* and its constituents which are presented to the user during proof construction.

We consider a *software system* to be composed of different classes and each class may in turn contain different methods. The most basic components of a software system we consider here for verification are thus single methods.

The task in program verification is to prove a software system S correct w.r.t. to its requirement specification Req. Following the notion of Beckert, Bormer, and Grahl [BBG16], we will call the pair $\langle S, \mathit{Req} \rangle$, consisting of a software system and a requirement specification, a *verification concern* (or simply concern) in the following.

In order to accomplish the task of verifying a concern users interact with a verification system and follow a verification process. For this thesis we will distinguish between two levels of a proof process: the global proof process (finding the right formalization and decomposing the proof task) and the local proof process (proving a single lemma, theorem or proof obligation). Thus, we also distinguish between two kinds of activities users perform during the process. Users perform activities on the *global level* and on the *local level*. Between the different proof artifacts dependencies exist throughout the proof (process) on the different levels considered in this thesis.

Dependencies on the Global Level. Given a software system S and a requirement specification Req, where S is part of a larger system $\hat{S}$ the requirement specification Req depends on the surrounding system $\hat{S}$ in two ways:

1. the *usage* contexts of S in the surrounding system (e.g., the call contexts), and
2. the requirement specification $\mathit{Req}_{\hat{S}}$ of the surrounding system.

The call contexts of the surrounding system are needed because the user has to provide a suitable precondition in Req that fits all known usage contexts of S in $\hat{S}$. The dependency between the requirement specification $\mathit{Req}_{\hat{S}}$ of the surrounding system $\hat{S}$ and Req exists in two ways:

(a) Req serves as auxiliary specification for the proof of $\mathit{Req}_{\hat{S}}$ and

(b) properties formalized in $\mathit{Req}_{\hat{S}}$ may also be formalized in Req_S

In this thesis we will focus on the requirement specification Req that formalizes the functional behavior of S. Req therefore depends on the implementation of S.

Dependencies on the Local Level. The dependencies mentioned before all covered the global proof process. In the following we will illustrate the dependencies that exist for the local proof process. Activities on the local level aim at proving the program correct w.r.t. a requirement specification by providing auxiliary annotations respectively interacting with the verification system. This includes comprehending the proof- and program states. There exist three kinds of dependencies for the requirement specification *Req* on the local level:

(a) the dependency to the auxiliary specification *Aux* within the implementation of *S* (e.g., loop invariants),

(b) the dependency to requirement specifications of called systems, which serve as auxiliary specification on the local level (e.g., the method contract of a called method), and

(c) the dependency to the implementation of *S* (e.g. a method body), which also includes a dependency on the different system states of the system's execution.

The auxiliary specification within *S* (dependency type (a)) is needed by the verification system in order to guide the proof search for the correctness proof of *Req* w.r.t. the implementation of *S*. If another subsystem is called in *S*, the requirement specification of the called system is used as auxiliary specification to prove *Req* (dependency type (b)). As the auxiliary specification states properties about program states, the correspondence between the states and the auxiliary specification is a dependency for *Req* as well (dependency type (c)).

Proof Verification Conditions and Proof Obligations. To prove a concern, the proof problem can often be divided into smaller units which we will call *proof verification conditions* (PVC). An example for PVCs is to generate a proof obligation for each conjunct in the postcondition when considering a method contract. If all PVCs can be proven valid individually, the system is correct w.r.t to its requirement specification, i.e., we were able to prove the concern. In our case (in Ch. 10) PVCs are generated by transforming the annotations and the program into a formula in first-order logic, however, there is no single, fixed definition how PVCs have to be built from a concern. An example for the relation between the annotated program and PVCs is shown in Example 3.8.

In contrast to the terminology used in the KeY system, where the term proof obligation also refers to the transformed sequent in the root of the proof tree, as well as the sequents in the open goals, we will call the logical representation of a PVC a *proof obligation* in Chapter 10. Our terminology would roughly correspond to the sequents in each open goal in KeY after symbolically executing the program in the modalities.

Example 3.8. In the following we will give a simple example for PVCs. For this we will use an annotated program (written in Dafny) that returns the maximum of two integers that are passed as parameters and which is annotated with the usual formalization of the functional property as postcondition.

```
method max(x: int, y: int) returns (m: int)
  ensures m >= x && m >= y
  ensures m == x || m == y
{
  m := x;
  if (m < y)
  {
    m := y;
  }
}
```

One way to transform the annotated program into PVCs is to treat each `ensures` clause separately for each of the two possible program execution paths, resulting in four PVCs:

- a PVC where the first post condition `ensures m >= x && m >= y` has to be shown for the path through the program when $m < y$ is assumed to be true and the body of the if condition is executed,
- a PVC where the second post condition `ensures m == x || m == y` has to be shown for the path through the program when $m < y$ is assumed to be true and the body of the if condition is executed,
- a PVC where the first post condition `ensures m >= x && m >= y` has to be shown for the path through the program when $m < y$ is assumed to be false, and
- a PVC where the second post condition `ensures m == x || m == y` has to be shown for the path through the program when $m < y$ is assumed to be false.

3.6. Interaction Styles in Interactive Program Verification Systems

As proving non-trivial properties of programs is in general an undecidable problem, user interaction is required for the proof process in the form of proof guidance for the underlying verification system. User interaction can take place on (a) the program code, (b) the specification and (c) on the proof obligation.

Like most user interfaces today, verification systems combine different interaction styles. However, three main styles have emerged as prominent interaction styles for proof construction: *direct-manipulation*, *text-based/script-based* and *annotation-based* interaction.

In the following we will describe the three styles in detail. For this we present the abstract proof process using these styles, the feedback the prover provides and the interaction between the proof system and the users.

3.6.1. Annotation-Based Interaction

The design concept for annotation-based verification systems is to hide prover details from the user and to treat the prover as a black-box. User interaction is limited to annotations in the program's source code, e.g., by providing pre- and postconditions or loop invariants. Additional hints for the proof search are also given using these annotations. The proof process in annotation-based systems is depicted in Fig. 3.1. The annotated program is encoded into proof verification conditions which are then each given to a theorem prover that automatically checks the validity of each verification condition.

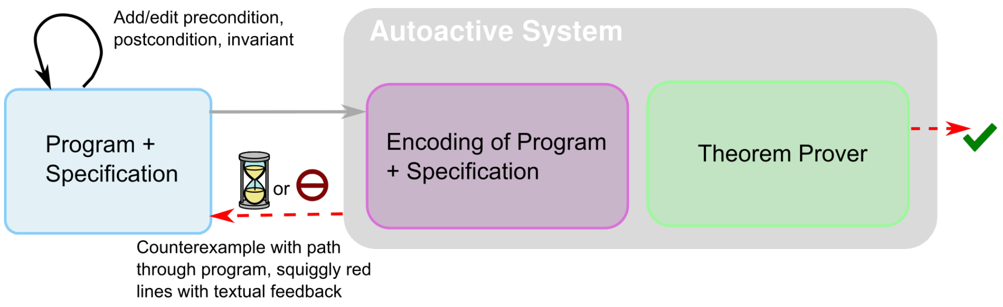

Add/edit precondition, postcondition, invariant
Autoactive System
Program + Specification
Encoding of Program + Specification
Theorem Prover
or
Counterexample with path through program, squiggly red lines with textual feedback

Figure 3.1.: Schematic view of annotation-based Interaction. The red dashed arrows indicate the feedback given by the system, black bold arrows with labels indicate the actions users may perform and the gray arrows depict information flow.

The verification process can have three results – the proof obligation is provable, the proof obligation is not provable or there has been a timeout for the proof search. In case a timeout is reached or the verification failed the user is informed via the feedback given by the prover. The chosen design enforces that, in case of timeouts or when the proof obligation could not be proven, the verification result needs to be well represented for the user. For failed proof attempts, feedback is given on the program level, e.g., in form of highlighting annotations or program statements that violate the proof obligation. Additionally, counterexamples can be provided by the underlying theorem prover in the form of variable assignments for program states (or traces) that do not satisfy the proof obligation. In the case of a timeout some systems provide tools to inspect the raw output of the theorem prover or statistics about the proof search (e.g., the number of quantifier instantiations for certain formulas) to be able to reformulate the annotations in case of performance problems [Bor14].

Advantages of this interaction paradigm is that the interaction takes place on the input representation of the problem. The annotations are comprehensible and are directly linked to the source code. At the same time, the amount of annotations may clutter the source code and the differentiation between requirement and auxiliary annotations becomes indistinct. Another disadvantage is that if a proof attempt is unfinished the user does not get detailed insight into the logical representation. To give more comprehensible feedback, annotation-based systems have integrated additional views on the proof state such that the black-box view is enhanced by providing details onto the proof states. Examples for annotation-based verification systems are VCC [Dah+09], Dafny [LW14], Spec♯ [BLS05], OpenJML [Cok11] or Frama-C [Kir+15].

3.6.2. Text-Based Interaction

In the text-based interaction style (also called command language or script-based interaction style throughout this thesis), the user interacts with the prover using a proof language. In state-of-the-art systems the proof language can vary between a declarative or a more imperative style.

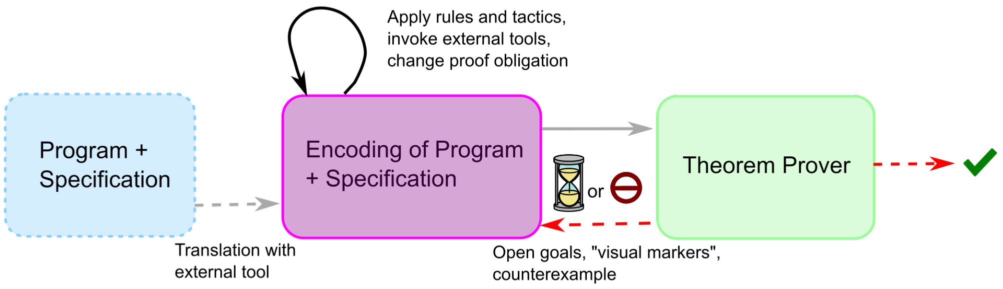


Figure 3.2.: Schematic view of text-based interaction. The red dashed arrows indicate the feedback given by the system, black bold arrows with labels indicate the actions users may perform and the gray arrows depict information flow.

A schematic view onto the proof process of a text-based proof system is given in Fig. 3.2. For text-based interaction, the program and its annotations have first to be encoded into a logical representation of the proof obligation (e.g., by the user or an external tool) and users interact on this encoded representation with the proof system. The users can apply proof rules, invoke proof strategies or use different additional lightweight tools, such as counterexample generators or SMT solvers. Feedback by the proof system is provided by presenting open proof goals or counterexamples to the user. The text-based interaction can be realized using a *command-language interaction* (CLI) or *script-based* interaction, using a proof scripting language.

The actions that are formulated using proof commands, can either be proper or improper proof commands, i.e., proper commands contribute to proof construction while improper commands are for inspection purposes [Wen99].

The *script-based interaction* is a more sophisticated form of the textual interaction paradigm. While in the command language interaction users provide proof commands which are directly evaluated by the verification system, in the script-based interaction, users provide proof scripts. Proof scripts can contain, in addition to the commands which are present in the CLI interaction, also control flow structures to combine the commands to more complex actions. The effect of an action is usually presented to

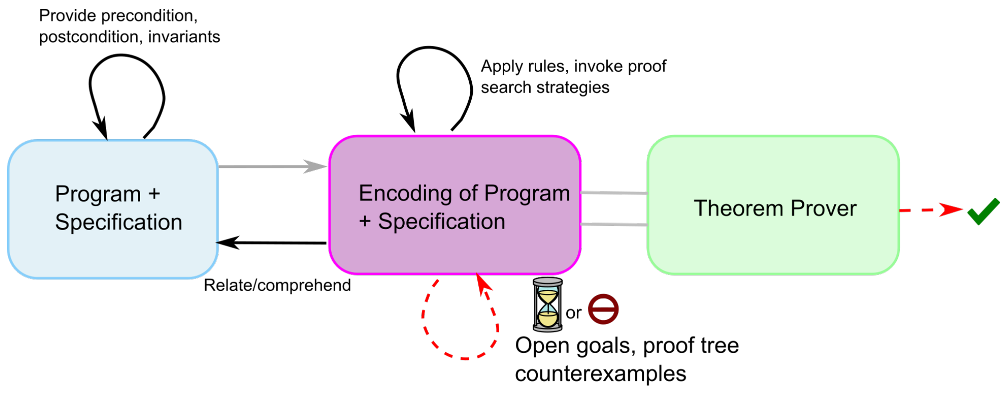

Provide precondition, postcondition, invariants
Apply rules, invoke proof search strategies
Program + Specification
Encoding of Program + Specification
Theorem Prover
Relate/comprehend
or
Open goals, proof tree counterexamples

Figure 3.3.: Schematic view of direct manipulation interaction. The red dashed arrows indicate the feedback given by the system, black bold arrows with labels indicate the actions users may perform and the gray arrows depict information flow.

the user in a text-based fashion, often only perceivable after the execution of a whole script. Instead of repeatedly supplying proof commands (e.g, in a batch mode or interactively by a user), an interpreter decides, based on the proof and the script state, which actions to take.

As a proof scripting language is a special type of programming language, constructs for modularization are incorporated in the language which allows the user to decompose large proofs to more understandable portions. Compared to direct manipulation interaction, the proof steps can be more coarse-grained. Depending on the proof language it is also possible to express the proof plan on a more abstract level than on the level of single calculus rule applications. However, state-of-the-art system using a text-based interaction allow only for a limited insight onto the logical level of proofs, e.g, viewing intermediate proof states of built-in proof strategies is not always possible.

As the proof representation and the actual proof problem are often shown to the user as two different views or representations the user always has to translate from the proof state back to the corresponding problem state.

Examples for script-based systems include Isabelle/HOL [NPW02], Coq [BC04] or PVS [ORS92].

3.6.3. Direct Manipulation Interaction

In the direct manipulation interaction style the user interacts on the logical representation of the problem and on the input, i.e., the annotated program. The abstract process for direct manipulation is depicted in Fig. 3.3.

The user provides the source code and its specification and lets the system transform the input into a proof obligation. Proof construction is done by pointing onto formulas and applying calculus rules. Additionally, the prover's proof search strategies can be configured and invoked by the user. The advantage of this interaction style is that the user has all necessary information available to make informed decisions about the next proof step. Interaction on the input artifacts includes that the user provides the auxiliary and requirement specification respectively corrects or adjusts the annotations or the source code. The user has full proof control in this interaction style. However, the interaction can be tedious as repetitive actions require the user to perform each interaction by clicking.

When the user encounters an error, either in the annotation or the source code, the large amount of available information makes error recovery a more time-consuming task as after error correction the user needs to find the last proof state from which the error was found. Also, as the user needs to interact on two different representations the user needs to relate the logical representation to the input representation.

Examples for verification systems that allow for direct manipulation interaction are the KeY system [Ahr+16], KeYmaeraX [Pla18] or KIV [Bal+00].

3.7. Program Verification Systems

In the following we present verification systems for each of the aforementioned interaction styles as primary style for proof construction. For each style we describe two representatives in more details, as they are also subjects in other parts of this thesis. Furthermore, other systems falling into the different styles are briefly described.

3.7.1. Direct Manipulation Interaction: KeY and KeYmaeraX

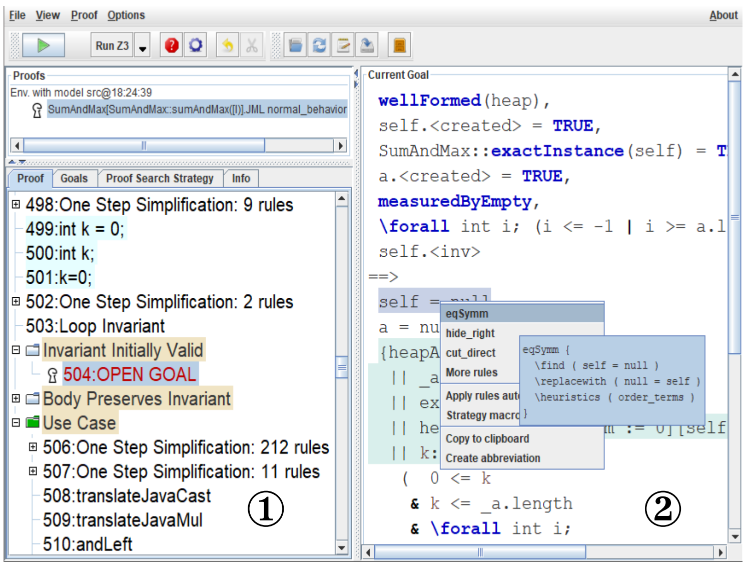
Program verification system that allow for direct manipulation interaction as central interaction style for proof construction are for example the KeY system, KeYmaeraX or KIV. We present two of these systems – KeY and KeYmaeraX – in more detail in the following.

The KeY system is an interactive theorem prover for the verification of various properties of Java programs (like functional correctness or information-flow) at source code level [Ahr+16]. It is developed at the Karlsruhe Institute of Technology, Technical University of Darmstadt and Chalmers University of Technology in Gothenburg, in the context of the KeY project[1]. KeY was successfully applied to verify real world Java programs, of which two recent examples have been implementations of Timsort [Gou+15] and Dual-Pivot Quicksort [Bec+17].

The Java programs are annotated with a KeY variant of the Java Modeling Language (the basics of JML are introduced in Sec. 3.3). The annotated source code is then translated into a proof obligation in JavaDL and discharged using a sequent calculus (details are introduced in Sec. 3.4).

KeY has an explicit proof object, i.e., KeY's user interface contains a representation of the current proof where all intermediate proof states can be inspected by the user.

[1]`http://www.key-project.org`



```
Env. with model src@18:24:39
SumAndMax[SumAndMax::sumAndMax([I)].JML normal_behavior
```

```
⊞ 498:One Step Simplification: 9 rules
  499:int k = 0;
  500:int k;
  501:k=0;
⊞ 502:One Step Simplification: 2 rules
  503:Loop Invariant
⊟ Invariant Initially Valid
    504:OPEN GOAL
⊞ Body Preserves Invariant
⊟ Use Case
  ⊞ 506:One Step Simplification: 212 rules
  ⊞ 507:One Step Simplification: 11 rules
    508:translateJavaCast
    509:translateJavaMul
    510:andLeft
```

```
 wellFormed(heap),
 self.<created> = TRUE,
 SumAndMax::exactInstance(self) = T
 a.<created> = TRUE,
 measuredByEmpty,
 \forall int i; (i <= -1 | i >= a.l
 self.<inv>
==>
 self = null
 a = nu
 {heapA
  || _a
  || ex
  || he                          m := 0][self
  || k:
   (  0 <= k
    & k <= _a.length
    & \forall int i;
```

```
eqSymm {
  \find ( self = null )
  \replacewith ( null = self )
  \heuristics ( order_terms )
}
```

Figure 3.4.: User interface of the KeY system

The user interface (see Fig. 3.4) contains two different views on the proof state, the proof tree (①), and the node view (②), where the sequents are shown.

The proof tree is a more high-level view on the proof state and contains all calculus rule-applications performed so far, as well as labels for the different proof branches in case the proof splits. Rule applications for symbolic program execution are highlighted with a different background color in the proof tree, to allow the user to distinguish these nodes from nodes where logical calculus rules have been applied to. In comparison, the node view is a more detailed view and can be obtained by selecting a proof node. This view shows the (intermediate) proof obligation in form of a textual representation of the sequent.

In its current implementation KeY also provides a view containing the proof input artifacts. After symbolic execution steps have been applied in the proof, the view highlights the program lines that have been executed symbolically in case a node is selected in the proof tree.

Proof construction in KeY is performed only by using direct manipulation (see Fig. 3.4). The user points to a term of the sequent in the node view and selects it by clicking, applicable calculus rules are then shown and the user can choose the appropriate rule. If the rule application requires additional parameters, an input dialog with drag-and-drop support is presented. Additionally, KeY offers several automatic proof search strategies (also called *macro* steps). In Table 3.5 we briefly summarize the effect of the macro steps in KeY that are mentioned throughout this thesis.

For easy reversibility of actions, users can click onto the proof tree and undo the actions of a whole subtree easily if they encounter that the automatic strategy has performed unintended steps.

Macro Step	Effect of the Application of the Macro Step
`One Step Simplification`	Applies simplification steps and presents the steps as one step in the proof tree.
`Finish Symbolic Execution`	Symbolically executes the program.
`Close Provable Goals Below`	Tries to close all goals below the selected goal, if not successful leave the goal untouched.
`Autopilot Preparation`	Applies the same steps as the macro step `Finish Symbolic Execution` together with simplifications rules and additionally splits the resulting goals into simpler goals.
`Autopilot`	Applies the same steps as the macro step `Autopilot Preparation` and applies the macro step `Close Provable Goals Below` to all goals.

Figure 3.5.: An overview over the macro steps in KeY and their effects.

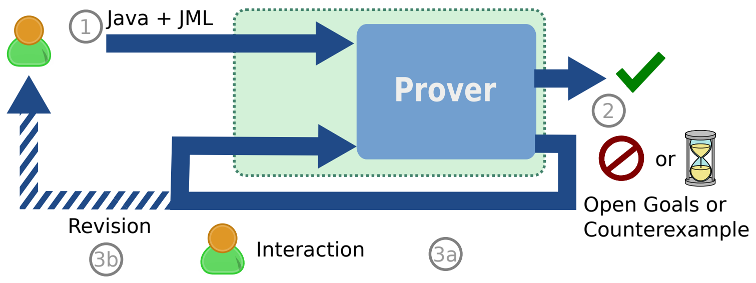


Figure 3.6.: Workflow in the KeY system

The way in which the direct manipulation interaction is implemented in the KeY system supports the user in focusing on one proof goal at a time, without losing the contextual information of the general proof as the information about the current state in the context of the whole proof is always accessible in the proof tree. When repeatedly applying the same proof steps to different goals, the user needs to find and select the respective proof goals and manually apply the proof steps to each of the goals.

The typical workflow of verification with the KeY tool is shown in Fig. 3.6: Initially, the user provides the proof input artifacts (step 1). After starting the automatic proof search, the result (step 2) is (a) the successful verification of the program or (b) either a counterexample or a proof with open goals that remain to be shown. In the latter case, the user may interact directly with KeY (step 3a) by interactively applying calculus rules (common examples are quantifier instantiations or logical cuts). Alternatively, the user may revise the program or specification (step 3b). Often, verifying programs in KeY involves both kinds of interactions, interspersed by automated proof search.

KeYmaeraX is an "axiomatic tactical theorem prover for hybrid systems" [Pla18; Ful+15]. Hybrid systems are represented as *hybrid programs* and the proof obligation in KeYmaeraX is a user-provided formula in *differential dynamic logic* [Pla12; Pla17].

The web-based user interface of KeYmaeraX [MP17] (see Fig. 3.7) allows interactive and automated verification using direct manipulation and text-based interaction interchangeably. The proof object in KeYmaeraX is an explicit proof object in form of a proof tree. The proof tree is represented as deduction paths in the user interface, where each path corresponds to one proof branch. Open goals are represented as sequent views in a tabbed pane. For each open goal a new tab is created. If the user wants to inspect the deduction path to the proof's root, unfolding actions are accessible in the sequent views that allow to step-wise add more details of the deduction path to the view.

Proof construction can be performed using a form of direct manipulation where positions in the sequent can be selected and upon right-clicking, possible rule applications are shown in a context menu. The rules are shown together with their rule description in the form of a sequent calculus rule. This representation is identical to the representation of the rules in their proof theory in order to adhere to the principle of familiarity [MP16]. Furthermore, menu-based interaction is supported for proof construction by providing different menus that contain the calculus rule names for selection on the goal currently in focus. The proof steps applied using direct manipulation or via the menus are persisted in the tactic view. In this view users can also perform proof construction in a text-based way as a Bellerophon tactic [Ful+17].

3.7.2. Annotation-Based Interaction: Dafny and Why3

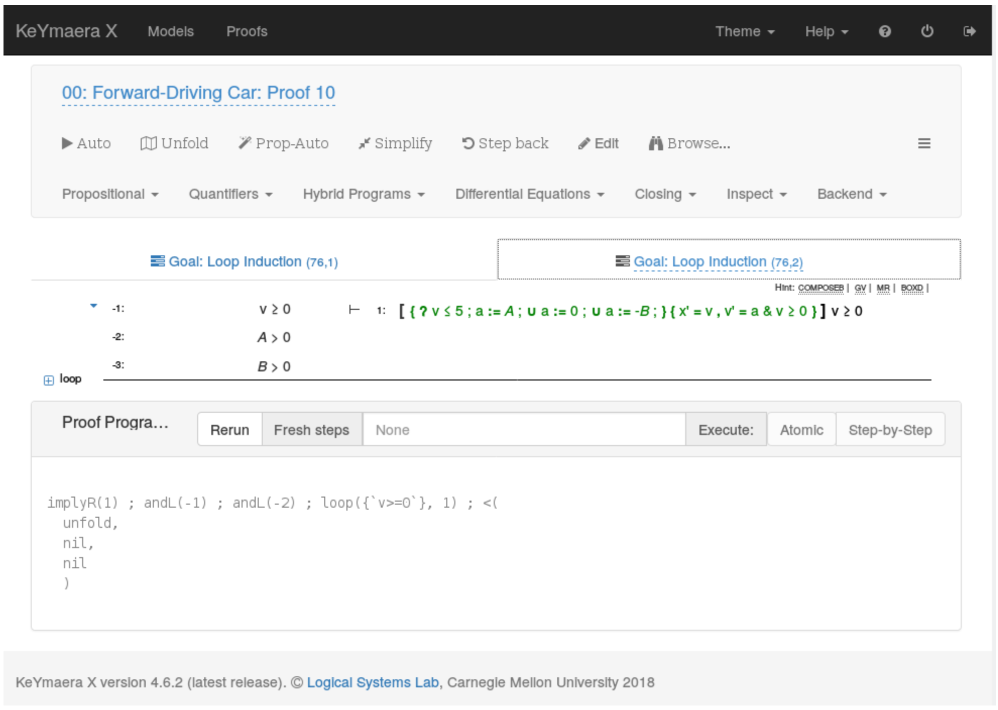
Examples for annotation-based program verification systems include Dafny [Lei10], Why3 [FP13], VCC [Dah+09] or OpenJML [Cok11]. We will present the two systems Dafny and Why3 in more detail in the following.

Dafny is a static program verifier for the verification of the functional correctness of annotated Dafny programs and allows for annotation-based interaction [Lei10; LW14].

The user provides the Dafny source code annotated with requirement and auxiliary annotations. The annotations of Dafny programs are similar to JML specification constructs. From this input, verification conditions are generated that can then be discharged using an SMT solver.

The Dafny tool is available as integration into Emacs and into the VisualStudio IDE (which is shown in Fig. 3.8). Especially the integration into a full IDE allows for the development and verification of Dafny code in one single system with the usual support for software development, such as syntax highlighting, debugging features and visualization of call and usage dependencies of software systems.

In case the verification attempt fails the user is provided details about the verification attempt on program level, e.g., by highlighting of possibly erroneous specification or program statements. Together with this highlighting the user is shown the corresponding paths through the program. Further details are presented using hover texts.



-1: $v \geq 0$ $\vdash$ 1: $[\{?v \leq 5;\ a := A;\ \cup\ a := 0;\ \cup\ a := -B;\}\{x' = v, v' = a \,\&\, v \geq 0\}]\, v \geq 0$

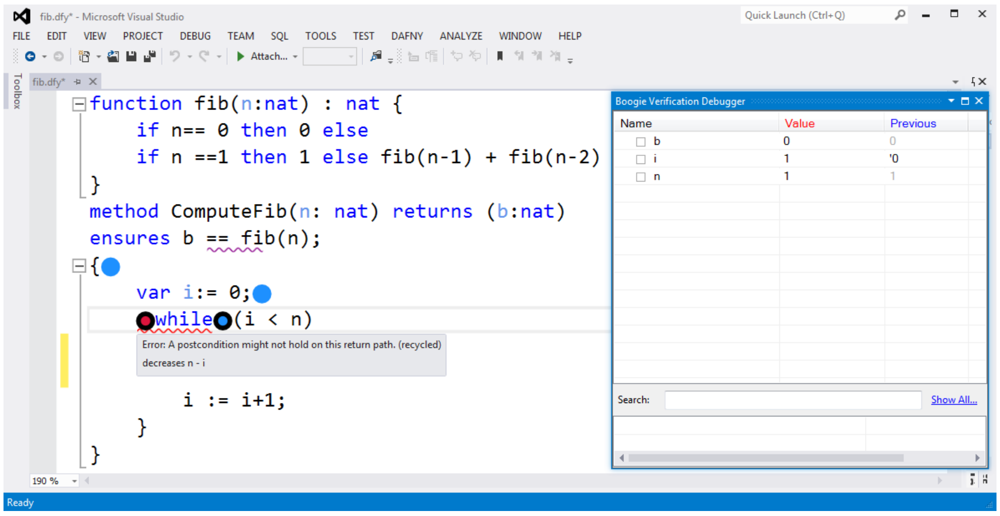
-2: $A > 0$

-3: $B > 0$

```
implyR(1) ; andL(-1) ; andL(-2) ; loop({`v>=0`}, 1) ; <(
  unfold,
  nil,
  nil
  )
```

KeYmaera X version 4.6.2 (latest release). © Logical Systems Lab, Carnegie Mellon University 2018

Figure 3.7.: User interface of KeYmaeraX


fib.dfy* - Microsoft Visual Studio
Quick Launch (Ctrl+Q)
FILE EDIT VIEW PROJECT DEBUG TEAM SQL TOOLS TEST DAFNY ANALYZE WINDOW HELP
Attach...
Toolbox
fib.dfy*
Boogie Verification Debugger
Search:
Show All...
190 %
Ready

```
function fib(n:nat) : nat {
    if n== 0 then 0 else
    if n ==1 then 1 else fib(n-1) + fib(n-2)
}
method ComputeFib(n: nat) returns (b:nat)
ensures b == fib(n);
{
    var i:= 0;
    while (i < n)
    Error: A postcondition might not hold on this return path. (recycled)
    decreases n - i
        i := i+1;
    }
}
```

Name	Value	Previous
b	0	0
i	1	'0
n	1	1

Figure 3.8.: User interface of Dafny showing an error message together with the program state at the location of the red circle.

The integrated Boogie verification debugger tool (BVD) furthermore depicts possible counterexamples by presenting partial program states, including heap locations, similar to a program debugging system for inspection.

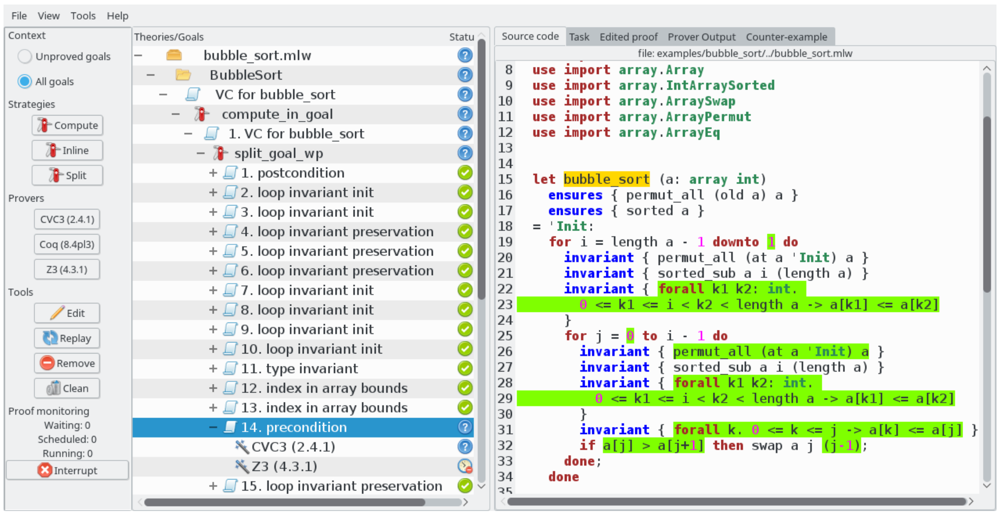
In contrast to the previously introduced verification systems, in Dafny the actual proof is not presented to the users, thus all interaction between the user and the verification system happens on program level.

Why3 is a verification system for the deductive verification of specified programs in WhyML, a first-order logic with polymorphic types, pattern matching, and inductive predicates [FP13]. Different front-ends allow translating programs in different programming languages into WhyML as intermediate language.

From the provided input program Why3 generates verification conditions that can then be discharged using various SMT solvers (e.g., CVC, Z3, Alt-Ergo) and theorem provers (e.g., Coq). A program fulfills the desired properties if all generated verification conditions are proven.

Interaction for proof construction is autoactive as well as interactive. The user can discharge all verification conditions similar to Dafny at once, however, due to the possibility of calling interactive theorem provers, interactive construction for proofs for verification conditions is also possible.

The user interface of Why3 (shown in Fig. 3.9) contains different views onto the proof state. Central is the list of proof verification conditions, which contains indicators about the verification progress of each verification condition. These indicators show which solver was used to prove the verification condition and the result, i.e., whether



```
– bubble_sort.mlw
  – BubbleSort
    – VC for bubble_sort
      – compute_in_goal
        – 1. VC for bubble_sort
          – split_goal_wp
            + 1. postcondition
            + 2. loop invariant init
            + 3. loop invariant init
            + 4. loop invariant preservation
            + 5. loop invariant preservation
            + 6. loop invariant preservation
            + 7. loop invariant init
            + 8. loop invariant init
            + 9. loop invariant init
            + 10. loop invariant init
            + 11. type invariant
            + 12. index in array bounds
            + 13. index in array bounds
            – 14. precondition
                CVC3 (2.4.1)
                Z3 (4.3.1)
            + 15. loop invariant preservation
```

file: examples/bubble_sort/../bubble_sort.mlw

```
use import array.Array
use import array.IntArraySorted
use import array.ArraySwap
use import array.ArrayPermut
use import array.ArrayEq


let bubble_sort (a: array int)
  ensures { permut_all (old a) a }
  ensures { sorted a }
= 'Init:
  for i = length a - 1 downto 1 do
    invariant { permut_all (at a 'Init) a }
    invariant { sorted_sub a i (length a) }
    invariant { forall k1 k2: int.
      0 <= k1 <= i < k2 < length a -> a[k1] <= a[k2]
    }
    for j = 0 to i - 1 do
      invariant { permut_all (at a 'Init) a }
      invariant { sorted_sub a i (length a) }
      invariant { forall k1 k2: int.
        0 <= k1 <= i < k2 < length a -> a[k1] <= a[k2]
      }
      invariant { forall k. 0 <= k <= j -> a[k] <= a[j] }
      if a[j] > a[j+1] then swap a j (j-1);
    done;
  done
```

Figure 3.9.: User interface of Why3. The highlighting of statements in the right part shows annotations and statements that either need to be proven or are used in the current proof.

a time out was reached, the prover has proven the verification condition or the prover was either not able to prove it or answered with “unknown”. To the right of the list of verification conditions users can choose to inspect parts of the proof state (i.e., the program or paths through the program to be verified) and more details on the results of the verification process, e.g., prover outputs or a counterexample (if one exists). If users choose a verification condition, the corresponding path through the program and the annotations that are proven or used in the proof are highlighted (as shown in Fig. 3.9).

Left of the central list of proof verification conditions, a panel is shown that contains access to proof construction actions. Proof construction is performed by selecting either a single proof verification condition or all verification conditions at once and either applying strategies, such as splitting the verification condition into smaller parts, or calling solvers for proving the selected the verification condition(s).

If the user chooses to perform an interactive proof search, the selected verification condition is transformed into a proof obligation for Coq and the interactive verification system Coq is called.

Other annotation-based verification systems Further annotation-based verification systems include OpenJML [Cok11] for Java programs annotated with JML. For annotated C source code the VCC system [Dah+09] was developed. Similar to Dafny, VCC is also integrated into the VisualStudio IDE such that developing and verifying C code is possible in one single system with the usual support for software development.

3.7.3. Script-Based Interaction: Isabelle/HOL and Coq

As two representatives of script-based verification systems we will introduce the proof assistants Isabelle/HOL [NPW02] and Coq [BC04]. Both systems offer script-based interaction as interaction style for proof construction, however, the script languages of both systems and their user interfaces differ.

Isabelle/HOL Isabelle is a generic proof assistant and proof framework, which allows to formalize the syntax and inference rules of different logics [WPN08]. Within the framework, Isabelle/HOL [NPW02] is a generic proof assistant for higher-order logics containing logic-specific tools as well as an extensive library of theories.

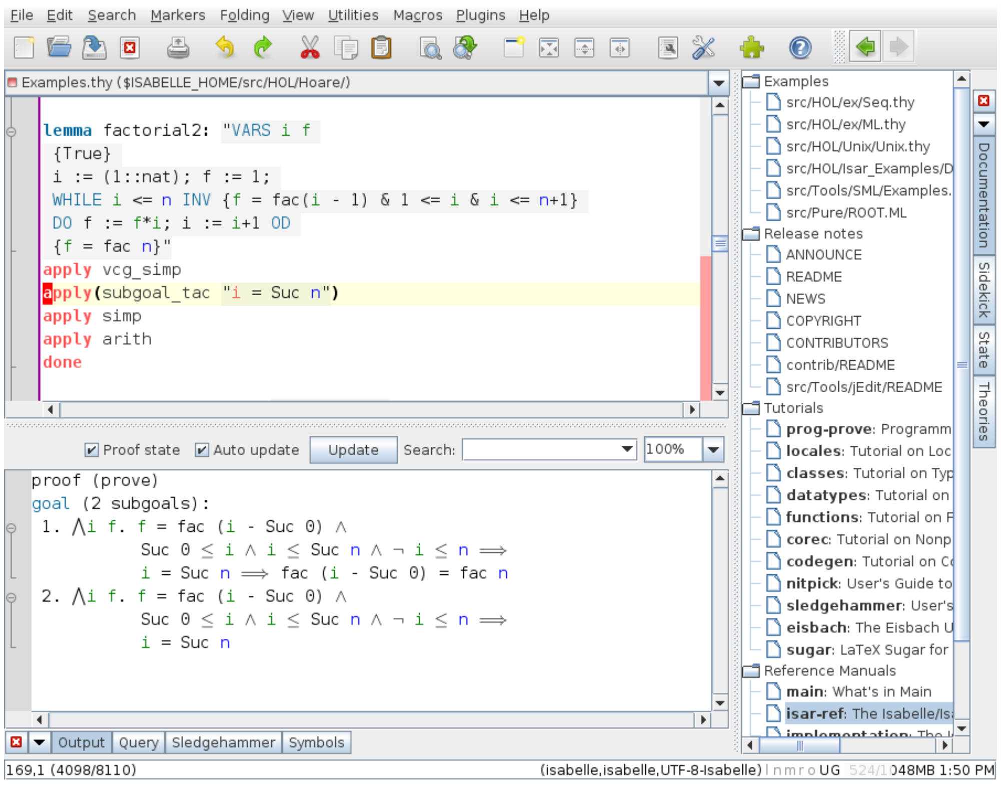
Besides its usage for general mathematical proofs, Isabelle can be used for the formal development as well as formal specification and verification of programs. One prominent example is the project L4.verified, where the functional correctness of the seL4 microkernel was proven using Isabelle/HOL [Kle+09b]. In the course of this project, also a proof of the C implementation was performed within Isabelle/HOL using Hoare Logics [Kle+14].

Depending on the tools available for a programming language, the user either provides annotated source code which is then translated into a proof obligation for Isabelle (as shown in [Hui01]) or the user formalizes the property in Isabelle/HOL itself, as it is also the case for general mathematical theorems.

Proof construction in Isabelle is text-based. For this, Isabelle provides the structured proof language ISAR [Wen99] and the *apply-style* proofs. ISAR allows users to formulate proofs in a text-book style as a form of more readable proof scripts where the user explicitly has to state what has to be proven in each step. Apply-style scripts are more linear and users instruct the proof system to apply tactics to the current sub-goals.

The user interface of Isabelle (see Fig. 3.10) is designed to support proof construction by textual interaction, i.e, writing proof scripts [Wen12; Wen18]. Central in the user interface is the view showing the current proof the user is writing. Syntax highlighting, as well as auto-completion features for script commands ease the task of writing proof scripts. In the toolbar below the menu bar users can access various functionalities for editing proof scripts. Proof scripts are executed in the background and Isabelle indicates errors and displays other proof management information (such as whether unproven lemmas have been used in a proof) by placing markers next to the lemma definition or statement in the proof script.

By placing the cursor on a proof script statement, users can view the proof state the statement is applied to below the central view. The proof state is depicted by showing the proof obligations of the open goals. Users can access various documentations and examples in the right panel of the user interface. Further user support is provided by features to search for lemmas and theories and additional tools such as Sledgehammer [Pau10; BBP13] and nitpick [BN10] that try to prove current goals or provide a counterexample if one exists.



```
lemma factorial2: "VARS i f
 {True}
 i := (1::nat); f := 1;
 WHILE i <= n INV {f = fac(i - 1) & 1 <= i & i <= n+1}
 DO f := f*i; i := i+1 OD
 {f = fac n}"
apply vcg_simp
apply(subgoal_tac "i = Suc n")
apply simp
apply arith
done
```

```
proof (prove)
goal (2 subgoals):
 1. ⋀i f. f = fac (i - Suc 0) ∧
          Suc 0 ≤ i ∧ i ≤ Suc n ∧ ¬ i ≤ n ⟹
          i = Suc n ⟹ fac (i - Suc 0) = fac n
 2. ⋀i f. f = fac (i - Suc 0) ∧
          Suc 0 ≤ i ∧ i ≤ Suc n ∧ ¬ i ≤ n ⟹
          i = Suc n
```

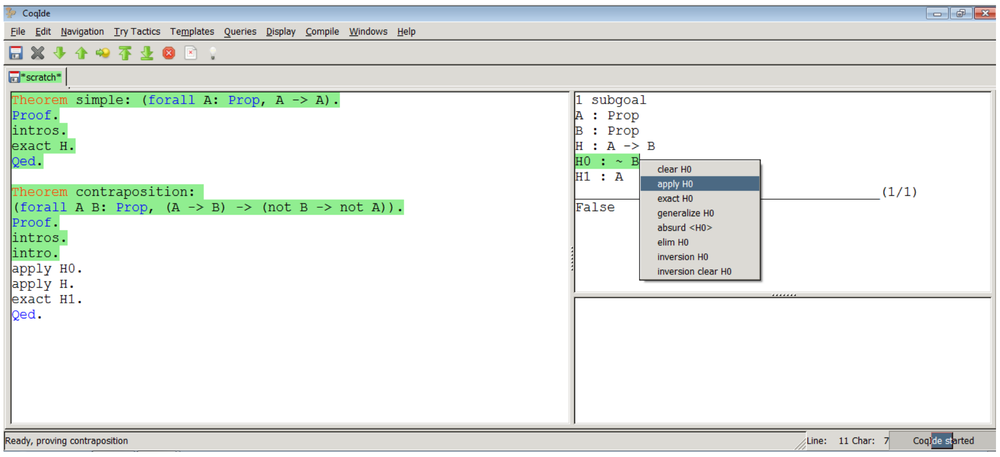
Figure 3.10.: The user interface of Isabelle/HOL.

Coq is a general proof assistant for proving mathematical theorems based on the theory “Calculus of Inductive Constructions” [BC04]. The user provides the formalization of the proof problem and starts the proof process by compiling the proof problem. According to Paulin-Mohring [Pau12], there are different ways to perform program verification within Coq: One way is by generating verification conditions from an annotated program as input for Coq, where the transformation is performed by an external tool such as Why3. A second way is to formalize a program language semantics in Coq and proving that a program is correct as mathematical statement. It is furthermore possible to use the type checking provided in Coq through the dependent types theory: the problem of proving a program correct with respect to its specification is reduced to the problem of checking whether the program, represented as Coq term, has the same type as the specification which is represented as type. The user interface of Coq is depicted in Fig. 3.11. Similar to Isabelle, in the user interface of Coq the text editor is central.

Proofs can be constructed using Coq’s specification language Gallina and its tactic language LTac [Del00]. LTac provides control-flow structures and means to select goals, e.g., by using match patterns. Proof scripts in Coq are the textual representation of the proof and can be executed as a whole (or until an error occurs) as well as step-wise.

Coq supports text editing by syntax highlighting as well as indicating errors by underlining the erroneous statements. In the left lower corner, feedback by the system is presented, e.g., error messages or failed tactics. The proof state is presented as a stack of open goals (in the upper right corner of the user interface). The current tactic is applied to the top-most goal if no selector is provided. The open goals are represented by the assumptions and formula to prove, where the assumptions are grouped together and are separated by a line from the formula to prove.

Besides textual interaction, Coq provides basic methods to use direct manipulation for proof construction. Users can select assumptions or the formula to prove and apply suggested tactics. The corresponding script statement is then added to the proof script.



```
Theorem simple: (forall A: Prop, A -> A).
Proof.
intros.
exact H.
Qed.

Theorem contraposition:
(forall A B: Prop, (A -> B) -> (not B -> not A)).
Proof.
intros.
intro.
apply H0.
apply H.
exact H1.
Qed.
```

```
1 subgoal
A : Prop
B : Prop
H : A -> B
H0 : ~ B
H1 : A
______________________________________(1/1)
False
```

Figure 3.11.: The user interface of Coq with the possibility to apply tactics using direct manipulation.

Other text-based verification systems The systems introduced so far allowed for script-based interaction with their own proof language that is similar to a programming language or text-book proof language.

A second variant are systems that allow for the classical command-line interaction, i.e., a command is read, directly evaluated in the current state and the newly created state is then presented to the user (also sometimes referred to as *read-eval-print-loop*). Amongst these systems is the Prototype Verification System (PVS) [ORS92], which is integrated into Emacs and also KeYmaeraX [MP17] allows for interacting with the proof system in this fashion. Further proof systems with command-line interaction include HOL4 [GM93] and HOL/Light [Har96].

Part II.

Exploring the Context of Use

Introduction

In this part of the thesis, we will use the qualitative research methods *focus groups* and *interviews with practical tasks* to explore the context of use[2] of interactive theorem provers.

We have chosen to perform a focus group discussion for the proof assistant Isabelle/HOL [NPW02] and for the KeY system [Ahr+16]. In addition to exploring the context of use of both systems, our aim was to explore opinions of users of these systems as well as to identify issues that arise when using expert systems for the task of proving mathematical problems or the correctness of programs. Before starting the user studies, we developed an initial hypothesis about problems in the proof process which we tried to strengthen using the studies.

We selected the evaluated systems, among other things, because of their contrasting user interaction approach: Although both systems allow for a combination of different interaction styles, their *main* interaction styles for proof construction are significantly different. While the KeY system allows the user to perform proof construction using the direct manipulation interaction style, Isabelle's main interaction style is text-based (more precisely, the interaction style is script-based). By using systems with different user interactions we expected that the findings about advantages and trade-offs of these styles may suggest that the systems complement each other when combined.

Furthermore, we conducted a user study consisting of interview questions and practical tasks with intermediate and expert users of the KeY system to explore the specific context of use of KeY. The main goals of this study were to gain insight into the usage of the KeY system as a program verification system and into details about the problem, respectively task domain. We wanted to explore the approaches of users dealing with different proof situations and the information they need in these situations to successfully proceed with the verification task.

The outcomes of the two explorative studies are:

- Information on how KeY users approach different proof situations and which interactions they use in the following situations: (a) at the beginning of the proof process and (b) after the automatic strategies have stopped (where users need to orientate themselves in the proof).
- General opinions of KeY users about the KeY system, especially the strengths and shortcomings of KeY.
- Ideas for room for improvement for KeY and program verification systems in general.
- Opinions about a new functionality that shows the history of formulas in the proof context, as well as opinions about the abstraction level on which such a history needs to be shown.

The qualitative research methods "focus groups" and "interviews with practical tasks" used in this part of the thesis were chosen to assess the context of use as well as

[2]See Chapter 7.4 for a definition.

the user interaction in interactive program verification systems, because they require a comparably small amount of participants and still are suitable to produce insights into the task domain and the user interaction. Compared to other qualitative user studies, the costs for conducting such experiments is comparably low. As interactive program verification systems are expert systems that are developed in academia the user base is not sufficiently large to perform quantitative evaluations.

4. User Study with Focus Groups

Contents

4.1. Problem Description and Research Hypothesis

The degree of automation of interactive theorem provers (ITPs) has increased to a point where complex theorems over large formalizations for real-world problems can be proven effectively. But even with a high degree of automation, user interaction is still required on different levels. On a global level, users need to find the right formalization and have to decompose the proof task by finding useful lemmas. On a local level, when automatic proof search for a lemma fails, they need to either direct the proof search or understand why no proof can be constructed and fix the lemma or the underlying formalization. As the degree of automation increases, the number of interactions decreases. However, the remaining interactions get more and more complex as ITPs are applied to more and more complex problems. When proving theorems, the automated proof search often leads the proof into a direction that differs from the way a human would conduct the proof. To interact with the theorem prover in a meaningful way during the proof process, users have to understand the prover's strategy and the state of proof construction and, thus, have to bridge the gap between their own model of the proof search and the current proof state of the tool. Open goals in partial proofs are the result of syntactic transformations that may not be intended to make it easy for humans to understand them. The intention of the transformations is rather to get the automated proof search closer to a complete proof. Therefore, users need to understand the prover's strategy and often have to look at *intermediate proof states*, resulting from rule applications onto the original proof obligation, to comprehend the current state.

Although it is easy to accept that there is a gap between a human user's model of the proof resp. proof search and the actual automated proof search, it is rather unclear how large the gap's impact on interactive theorem proving is for typical proof obligations. Nevertheless, the following is a central hypothesis for our work, which we wanted to explore during our evaluation:

> Bridging the gap between the user's model of the proof state and the state of the theorem prover at interaction points is one of the paramount challenges for efficiently and effectively usable general theorem provers.

In addition, we are interested in evaluating which tools or mechanisms are already present in today's provers that help to bridge the gap and how to extend existing mechanisms to help the user in understanding the proof states. The contribution presented in this chapter is that we conducted an experiment using the survey method focus groups to explore arguments for or against our hypothesis and to gain answers to our two questions: *a*) Which mechanisms of this kind are already used in theorem provers? *b*) What mechanisms are missing? Also, information about the users' experiences and opinions with and towards the systems was target of the study.

We start by describing our model of the proof process between a user and a proof system which should describe our hypothesis. Following this description, we introduce the design of our user study in Section 4.2 and present the conduction in Section 4.2.3. In Section 4.4 we present the results of the experiments and relate them to our hypothesis. Section 4.3.3 presents our results regarding mechanisms and tools for understanding the proof state. We conclude and discuss future work in Section 4.5.

The User's and the Tool's Model of the Proof Process

In the following we will present the underlying model we assume for our first hypothesis. ITPs are used to aid users in proving complex theorems in many areas of computer science and mathematics. For using such systems, the user needs to have a certain level of experience in proving theorems. In general, the user has a concept or plan of how to prove the desired theorem. We call this concept *user's model of the proof.* This can either be already a whole proof plan or just preliminary ideas on the proof process. This model also includes an assumption about the theorem prover's strategies as we do not consider the proof plan for a pen-and-paper proof as being the user's model, but the proof plan for how the user would prove the problem using a theorem prover.

One big difference between the user's model of the proof and the current partial proof is that the proof steps in the model are more coarse-grained and have an intuitive (summing up) semantic for the user (such as "simplification of the proof obligation"), whereas the prover's steps are more fine-grained and are syntactic manipulations of the proof state. While an intuitive semantics for each rule application exists (as given by the rule's author), a sequence of consecutive rule applications in the system may not have a clear intuitive semantic for the user.

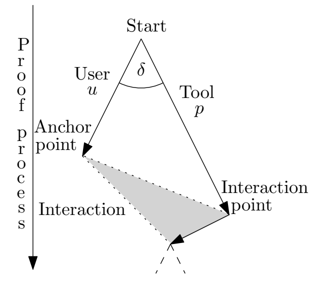


Figure 4.1.: Our model of the proof process to explain the hypothesis.

In Figure 4.1, we have sketched our idea of the relation between the actual proof performed by the prover's search strategy (p) and the user's proof model (u). At the beginning of the proof process, the user's model is either identical with or close to the proof obligation in the proof system. However, the more the automatic strategies of the prover modifies the original proof obligation in order to try to prove the proof obligation (depicted as the arrow p in Fig. 4.1), the more the actual proof state in the system differs from the user's model (depicted as the arrow u in Fig. 4.1). As the user has to guide the prover by interacting with it, the user needs to understand the (proof) process of the prover and relate the actual proof state to the user's model. For the comprehension of this relation, the user needs to inspect the current proof state (*interaction point*) and find a corresponding state in the own model of the proof (*anchor point* in Fig. 4.1). After the user interacts with the prover, the proof performed by the system below the interaction point is proceeding to some extent into the direction of the user's model, hence reducing the gap.

In some cases, no useful anchor point exists. Then the user has to understand the automatic proof construction and, in doing so, construct a new mental model u that is identical with or an abstraction of p. In contrast to the automatic proof process, if the user only applies rules manually and there is no automatic proof search, then p is identical to u (in case the user fully understands the effect of the applied rules).

In the standard case, however, where there is a gap between u and p, there should be mechanisms in the systems that help the user in relating the anchor point with the interaction point (indicated by the dotted lines in Fig. 4.1) to bridge this gap. In general, we can identify two parameters in our model of the proof process which can differ from proof system to proof system: the size of the gap between the actual proof and the user's model (δ), and the mechanisms that help to relate the user's model and the current proof state to aid the user in comprehending the proof state (dotted line between anchor and interaction point).

Apart from the gap, it could be the case that the user does not have a clear model of the proof or even none at all. Here, the gap as described is not applicable. In this case, the user uses the automation of the prover without any model in mind in order to use the resulting proof state to concretize the own fuzzy model. Therefore, the user has to comprehend the resulting proof state.

4.2. Study Design and Implementation

The questions of the scripts for our conducted focus groups to explore arguments for or against our hypthesis, are described in the following. The planned duration for both groups was two hours. Due to lively discussions, the actual duration was 2.5 resp. 3 hours.

4.2.1. The Script

The main questions and tasks in the script were the same for both discussions to obtain comparable results. The only two differences were adaptations of the questions and tasks to the differening terminologies, as well as adaptations of feature mock-ups to the specifics of the two systems. The full scripts for our experiments are available on the webpage `http://formal.iti.kit.edu/~grebing/SWC` (as the discussions were conducted in German, the original scripts are in German as well). Table 4.1 provides an overview of the explicit questions for the participants. The discussion was divided into three parts: the warm-up, the main and the cool-down part. The tasks of all three parts will be described in detail in the following. As a warm-up task, we asked about typical application areas of the proof systems and about their strengths and weaknesses related to the proof process. In the main part of the discussion, we had two topics: (1) support during the proof process and (2) mechanisms for understanding proof states. As a cool-down task, we asked the participants to be creative and imagine their ideal interactive proof system.

The Warm-Up Part

As warm-up task, we asked about typical application areas of the systems and about their strengths and weaknesses related to the proof process. Our intention for this part of the discussion was twofold. Firstly, we wanted the participants to slowly focus on the proof process of their system and "warm up" for the main part of the discussion. Secondly, our goal was to gain insight into the advantages and disadvantages of the systems as mentioned by the participants to draw conclusions about desirable features for interactive theorem provers. In addition, we expected to obtain detailed information about the proof systems such that we can name the issues and develop advices on how to improve the proof systems.

The Main Part

The main part of the discussion covered the following two topics:

1. The proof process: What does the proof process look like? How does the tool support the user during this process?

2. Mechanisms for understanding proof states: We confronted the participants with mechanisms that might help them to understand the current state of a proof during the proof process.

Warm up	1. Name typical use cases of the system. 2. Name a strength of the system related to the proof process. 3. Name a weakness of the system related to the proof process.
Main	For the global and the local proof process: 1. How do you conceive a formalization/specification for a given problem? a) Please try to sketch the process. b) Please point out steps of the process during which you get help/feedback from the system (if any). c) Do you repeat certain sequences of steps during the process? If so, please mark these loops. 2. (Discussion) a) How do you rate the feedback you get from the system? (If negative: Where would be room for improvements?) b) Which steps of the process consume most of your time? Why? c) Which steps of the process annoy you? Could they be automated? d) What do you do if you get stuck? e) How do you rate the granularity of the proofs (in the local process)? For the mechanisms: 1. Please describe the presented mechanism. 2. Please rate the presented mechanism. 3. What do you make of the approach?
Cool down	1. Be creative and describe your ideal interactive proof system. Disregard technical restrictions apart from the effectiveness of the automated proof search. Name capabilities the system should definitely have. Name properties it must not have.

Table 4.1.: Summary of the questions from the script for the focus groups.

Topic 1: The Proof Process. We divided the discussion for this topic into two parts, namely the *global proof process* (finding the right formalization and decomposing the proof task) and the *local proof process* (proving a single lemma or theorem). For each part, participants were asked to describe their typical proof process and discuss where the prover gives support and where support is missing. We also asked for the most time-consuming actions.

By discussing the proof process, participants remember their typical interactions with the system in the past. This retrospective view supports the subsequent discussion of how users get assistance from the systems during the proof process. Based on the participants' retrospections, we hope to identify repetitive or time consuming tasks, and activities where system feedback is missing.

Also, we expect information on participants' usages of the systems to solve particular tasks, on actions/phases in the task completions or where the participants switch to other tools or systems (e.g., text editors, or pen and paper), on how they inspect the proof state, and on how they guide the prover in finding a proof. We also expect to get ideas from the participants on how and where they would improve the systems.

Topic 2: Mechanisms for Understanding Proof States. For the second topic, we did not just ask for available or missing mechanisms. Instead, we initiated a more focussed discussion by presenting mock-ups of mechanisms which we envisioned as potential extensions for the tools. The mock-ups were presented as a sequence of user interface screenshots that have been modified according to the envisioned effect of a mechanism. These sequences of screenshots showed how a user can invoke the mechanisms and the corresponding effect of the mechanism[1]. The purpose of the presented mechanisms was to support users in understanding proof situations. The design of the particular mechanisms was based on first hypotheses and influences by first informal questionings of some users.

The mock-ups showed (a) a mechanism for tracing formulas, terms, and variables that are generated during proof construction back to the original proof obligation (for both tools), (b) a visual support for proof management that shows which lemmas contribute to a proof (for Isabelle), and (c) a mechanism for highlighting local changes between two adjacent nodes in the proof tree (for KeY). Thus, we made use of the possibility to use focus groups to get a first assessment of new features.

For all presented mechanisms we had the same course of action and questions. First, the participants were asked to describe what they believe the mechanism does (i.e., the mechanism was not explained by the moderator). This was done both to avoid bias introduced by the moderator and to see if the mechanism is intuitive. Then, the participants were asked for their opinion on the usefulness of the mechanism. We expected to gain feedback about the presentation of the mechanism. If the participants need too much time to understand the functionality, we have to revise the presentation of the mechanism for a second version in order to develop mechanisms that suit the user's needs and expectations We also had to some extent the opportunity to gain

[1]In the Appendix A.1 two examples of screenshots, which we have shown during the user study, can be found.

feedback for different presentations of the same functionality, one way of presenting a mechanism was to use an own window for the mechanism, the second way is to incorporate the mechanism in the provers graphical user interface.

With the mocked mechanisms as starting point, we expected to invoke a discussion about the usability problems with respect to the proof process for which the presented mechanisms might help. Additionally, we wanted to gain detailed insight into what annoys the users and in which way they would like to see their system improve. Lastly, we expected feedback on which mechanism was preferred most.

The Cool-Down Part

For the cool-down task, we asked the participants to be creative and imagine their ideal interactive proof system.

The main idea behind the cool-down task is that the participants leave the discussion with a positive experience. Our intention was that we also gain some more, possibly creative, ideas on which features an ideal verification system should or should not incorporate.

4.2.2. Participants and Setup of the Study

Participants. The participants for our experiments were recruited using personal contacts to the relevant communities. We ensured that each group included novice, intermediate, and expert users in different proportions. Besides that, the only criterion for selection was that participants had to be open about the idea of focus group discussions (mostly they were interested in a new experience and to learn something new about using their tools). Most participants were Master or PhD students, who had used KeY resp. Isabelle for their thesis work. We reimbursed participants' travel expenses, but they were not paid a further recompense. The KeY group had seven and the Isabelle group five participants. In the Isabelle group we had one novice, two intermediate and two experts users. In the KeY group we had one novice, two intermediate and four expert users.

Moderators. We had two different moderators, one for each discussion. Both were computer scientists working in academia but not in the area of ITP. As they were not expert moderators, they received an extensive training and briefing prior to the discussions.

Technical Setup. We followed the advices of expert project members about the spatial setup and used two adjacent rooms (one for the discussion, one for the observers) with a glass window between them. The spatial setup is depicted in Figure 4.2. The technical equipment consisted of one camera and four microphones for recording, a back channel from the observers to the moderator's headset, and lecture recording software capable of recording and live streaming.

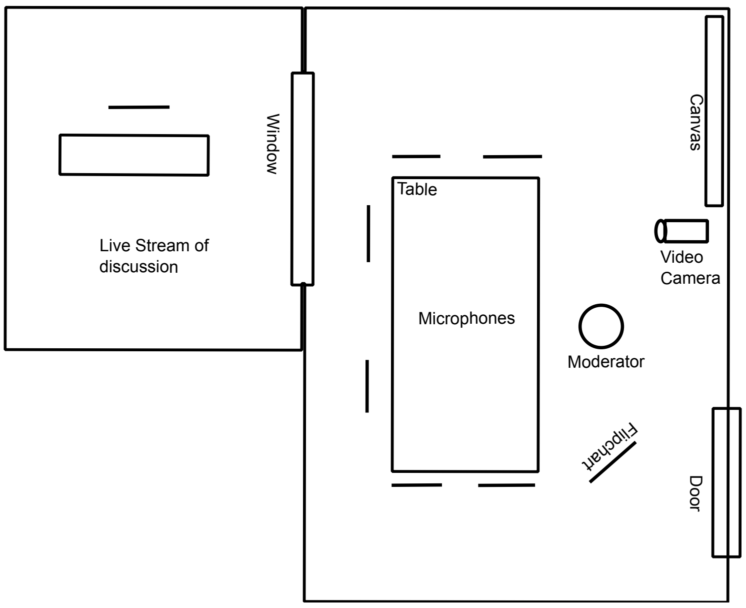


Figure 4.2.: Our room setup for the focus group discussions. The participants were sitting at the table facing each other.

4.2.3. Conducting the Focus Group Discussions

Our discussion had three stages: the warm-up stage, the main stage and the cool-down stage as described in Section 4.2.

The discussion was carried out according to the script, which is explained in detail in Sec. 4.2.1. All in all, both discussion groups were lively and the participants engaged well in the discussion. Our impression was that the participants were open towards this method and were upfront about their systems. Our moderators were not experienced with moderation tasks and sometimes asked suggestive questions such as “Do you all share person A’s opinion?”. While this required an extra-careful analysis of the transcribed material, the damaging effect of such questions for the results was minimal. A thorough analysis of the video material showed that often the group or certain participants confirmed or denied the statement before the corresponding suggestive question occurred.

After the discussion, all participants had the opportunity for a short offline talk with all project members and ask questions as well as express opinions about the focus group without being recorded. We believe these offline discussions were a good opportunity for the participants to gain more information about the focus group method as well as for the clarification of some issues which may not have been addressed sufficiently in the discussion.

4.3. Data Analysis

In order to analyze the data collected in the two focus group discussions, we used qualitative content analysis. Our categories were formed deductively based on the script's questions and inductively from the transcribed material.

4.3.1. Targets of Evaluation: KeY and Isabelle

In the following, we will briefly introduce the two systems under evaluation with the focus on those parts that were mentioned by the participants of the focus groups. Here, we start with the application areas of the systems as given by the participants.

KeY system. The KeY system is an interactive verification system for programs written in Java annotated with the Java Modelling Language (JML). As such, it is mostly used for the verification of Java programs w.r.t. a formal specification (usually a functional specification but also for other specifications, such as information-flow properties). KeY is also used for teaching and demonstrating formal methods, and as verification condition generator for other systems. KeY has an explicit proof object, i.e., all intermediate proof states can be inspected by the user. KeY uses a sequent calculus for Java Dynamic Logic [BKW16]. Its user interface shows proofs as a tree, the nodes of the tree contain intermediate proof goals (i.e., sequents). Each node N is annotated with the rule that was applied to some term or formula in N's parent node to construct N.

Isabelle. Isabelle is a theorem prover for higher-order logic. As mentioned by the participants, it is especially used for the formalization, verification and execution of algorithms, for proving in general and for the development of formal models. It has an implicit proof object, i.e., not all intermediate proof states are shown to the user, only goal-states where the system stops its automatic strategies. These automatic strategies are called *methods*, however the participants used the term *tactics*, therefore we use this term throughout the paper. Isabelle's proof tactics are sets of rules or lemmas that can be applied to the goal state. In this chapter, the *auto* tactic will often be mentioned, which applies a large number of rule sets automatically, and the *simp* tactic, which applies rules that simplify the goal-state. Within Isabelle it is alos possible to invoke different tools that generate counterexamples (e.g., *nitpick*, *quickcheck*) or that invoke SMT solvers to find a (sub-)proof (e.g., *sledgehammer*).

4.3.2. Strengths and Weaknesses of the Targets of Evaluation

Here, we discuss the strengths and weaknesses of the systems with respect to the proof process as mentioned by the participants. Interestingly, some characteristics of the systems that were first named as a strength led to lively discussions in later phases, which often brought up negative aspects of the same characteristics.

Table 4.2.: Strengths of the two systems according to the participants. The labels indicate whether a characteristic is linked to our (M)odel of the proof process (see Section 4.1) or rather to (O)ther aspects of interactive theorem proving (the classification is our own and not performed by the focus group participants).

KeY	Isabelle
· Expressive specification language (O)	· Underlying language very intuitive (M)
· Proof can be inspected in detail (M)	· Helpful community (O)
· KeY tries to simplify open goals (M)	· Large public library of theorems (O)
· High degree of automation for simple problems (O)	· Automatic tactics and tools ease proof process (M)
· All proofs follow a similar structure (M)	· Proofs can be modularized (M)
· Intuitive presentation of proof by using macros and proof tree (M)	· Flexible w.r.t. use of top down or bottom up approach (O)
· Allows user-defined rules (M)	· Code export for testing the model (M)
· Support of JML (O)	· User-adjustable syntax (M)

Strengths of the Targets of Evaluation

First, we discuss results of the focus groups w.r.t. the strengths of the systems, which are summarized in Table 4.2.

KeY System. The group on KeY agreed that the expressiveness of the system is an important strength. The participants liked how the Java Modeling Language can be used to annotate Java code. They appreciated that a proof with the KeY system always follows a certain structure, that this structure is visualized in form of a proof tree, and that this tree can be inspected at an arbitrary level of detail. Macros, which group rules similar to tactics in Isabelle, ease the interaction process and help to give the proof the direction intended by the user. According to the participants, the KeY system can solve easy problems without any or with only very little interaction. Furthermore, KeY supports user-defined rules, which can be of help during the proof process.

Isabelle. The group on Isabelle considers the underlying proof input language Isar to be one of the system's main advantages. It allows for proofs to be structured and presented in a standard textbook style that is very intuitive for humans. The large user community of Isabelle is considered to be an important strength. It provides a growing (and already quite extensive) library of theorems available to everyone. Furthermore, the community is a good resource of knowledge and friendly towards beginners. Isabelle provides a variety of tools that help during the proof process, e.g., *sledgehammer* and *nitpick*. The system can be used for a top-down as well as for a bottom-up proof approach.

Table 4.3.: Weaknesses of the two systems according to the participants. The labels indicate whether a characteristic is linked to our (M)odel of the proof process (see Section 4.1) or rather to (O)ther aspects of interactive theorem proving (the classification is our own and not the focus group's).

KeY	Isabelle
· Necessity of repeated trivial manual interactions (M)	· Finding the right tactic for a proof state is a non-trivial explorative task (M)
· Not possible to get practically usable counterexamples (M)	· Unexpected inference of types leads to unintuitive errors (M)
· Proof tree too detailed (M)	· Bloated formulas (M)
· Interaction on low-level logic formulas required (M)	· No insight into automatic tactics; unintuitive (M)
· Unintuitive mapping between formula and program (M)	· Messy downward compatibility for older proofs in newer system versions (O)
· Performance of automatic strategy (O)	· No support for proof refactoring (O)
· Practical scalability (O)	· Library: important mathematical foundations are missing (O)

Weaknesses of the Targets of Evaluation

The results of the focus groups w.r.t. weaknesses of the systems, i.e., room for improvements are shown in Table 4.3. For this brief overview, we omit some of the more technical remarks by participants that are not related to the general proof process in our opinion. For example, regarding KeY there were complaints about an unstable proof loading mechanism and memory leaks. Some Isabelle users complained about specific features of jEdit – a widespread editor for Isabelle proofs.

KeY System. Interestingly, the following characteristics of KeY that were named as strengths by the focus group were also identified as areas with potential for improvement. (a) The proof tree – whose existence was perceived as a strength of KeY – was considered to be too detailed. (b) Some participants stated that linking proof states to Java code would be helpful. (c) Interaction on the low-level logic formulas is necessary, sometimes trivial and tedious. (d) Manual interaction often needs to be repeated in (several) similar situations. (e) There are no useful tools to generate counterexamples.

Isabelle. According to the participants, an important downside of Isabelle is that the process of choosing the right tactics [2] and tactic parameters to conduct a proof is not always intuitive. If a tactic cannot be applied successfully in a situation, it is hard to find the reason. A technical problem is that type inference sometimes leads to very

[2] Applying a *tactic* in Isabelle is a hint by the user for the proof system how to resolve a situation, e.g., "use complete induction".

unintuitive errors. Additionally, formulas belonging to different properties that could be checked (and thus presented) independently are all combined in a single goal state which increases the size of the formula (e.g., invariants encoding type information for functions).

An often recurring task when working with Isabelle is to refactor proofs towards better understandability, however, there are no tools for refactoring. While the public library of theorems was also mentioned as a strength, a weakness is that some important mathematical foundations are still missing, i.e., in some theories lemmas are still missing.

Observations and relation of results to our model.

In the following, we relate results of the focus groups to our model of the proof process (Section 4.1) and to our first hypothesis(Section 4.1). We evaluate the characteristics (Table 4.2 and 4.3) w.r.t. to three challenges an ITP has to solve:

(A) Keeping the gap small. In general, mechanisms that help to keep the gap between the tool's proof state and the user's mental model small are seen as strengths of the systems – unintuitive behavior of the tools in the proof process is often mentioned as a problem. Several strengths of KeY help to keep the gap small: Proofs follow the same structure, macros help to guide the proof into the expected direction (similar to tactics which were mentioned as a strength of Isabelle), and users can introduce new rules that match their intuition (these rules have to be proven correct). Both tools allow the proof to be modularized (i.e., in Isabelle it can be split up into lemmas, in KeY into contracts) – this allows structuring the proof as a sequence of statements which may be more intuitive for humans. Some KeY users stated that they use the automatic proof search only if it closes a branch, as otherwise the resulting state is too unintuitive to continue interactively.

(B) Bridging the gap. Understanding a given proof state is an important challenge for users of both systems during the proof process. Consequently, mechanisms and characteristics of the systems that help the user's understanding are considered to be important strengths. Here, Isabelle provides a couple of useful tools (e.g., quickcheck and nitpick). Furthermore, the intuitive structure of the underlying language Isar is named as an important strength. Correspondingly, the absence of suitable mechanisms for certain situations is an important weakness. For example, participants criticized that KeY does not provide a useful tool to generate counterexamples. Such a tool is necessary to detect whether the prover is stuck because further user input is needed or the property to prove does not hold and no proof exists respectively no proof can be constructed. While there are tools to generate counterexamples for Isabelle, the counterexample representation could be improved in the eyes of some participants in case proof obligations contain functions. Currently, it is difficult to find the part of a proof obligation that is not provable.

(C) Supporting Interaction. Finally, as soon as users have a sufficient understanding of the proof state, they need to interact with the tool in an effective way. In this area there still seems to be a lot of room for improvement for both tools. The participants of the KeY focus group criticized that the interaction often needs to be performed not on the annotation level (as it would be desired) but on low-level logic formulas. Furthermore, low-level steps have to be manually repeated in similar proof situations. The Isabelle users were unhappy about the tedious task of finding the correct tactic to successfully continue the proof.

Conclusion

We observe a strong connection between the named strengths and weaknesses and our model of the proof process from Section 4.1. More than half of the mentioned characteristics can be associated with concepts introduced by the model. Furthermore, the results support our hypothesis (Section 4.1) that bridging the gap between the user's model of the proof and the ITP's proof state is very important during the proof process.

4.3.3. User Support during the Proof Process

We divided the part of the discussion about the proof processes into two parts, namely the *global* proof process (finding the right formalization and decomposing the proof task) and the *local* proof process (proving a single lemma or theorem). The participants were asked to describe their typical proof process respectively, and to name feedback mechanisms that the systems provide. Our expectations were that existing prover support and mechanisms to aid the user are adapted to the respective abstraction levels of the two processes.

Global proof process. For both, KeY and Isabelle, the participants described a similar proof process: it starts with the formalization of the system/problem and its main properties. Users considered the modeling task to be among the most time-consuming ones. However, system feedback in this phase is restricted to syntactical and simple consistency tests. Instead, feedback causing the user to revise the model on the global level results from the *local* proof process. It is not surprising that there is only little user support for the global process, as the tasks often require creativity and depend on the particular problem.

Local proof process. In the local proof process, the users are guided by their individual impression of the complexity of open goals/proof obligations. If the user considers the obligation to be "easy enough", he or she tries a fully automatic strategy. Otherwise, or in case the automation fails, the user tries to prove the proof obligation interactively. In this case there are two options: structured proofs (Isar/macros) or proof exploration (manual application of rules respectively tactics).

The case where the problem is considered to be easy and is tried to be proven automatically fits our model: It is the case where the user's proof plan has only one step leading to the proof state “proof complete”. In the other case, proof exploration corresponds to the user having only a partial proof model, or a set of different models from which the appropriate one has to be determined. In terms of Fig. 4.1, we observe multiple arrows originating from the proof obligation (the starting points of the arrows in Fig. 4.1).

Both KeY and Isabelle aid the user by providing search mechanisms or suggestion mechanisms for proof rules respectively lemmas: As stated by the participants, Isabelle supports the user in finding the right proof technique with a search mechanism for theorems in the library. KeY offers search mechanisms (e.g., textual search on the different proof artifacts) and suggests applicable rules for a user-selected formula.

System feedback for the local proof process. In the local process the systems provide different kinds of feedback, e.g., counterexamples, open or closed goals, and (partial) proofs containing open and closed goals. Some of these are explicit (e.g., via message boxes), others are implicit via a changed proof state.

The main difference between both tools is that KeY provides the full path to the open goals as proof tree, while no explicit tree is available in Isabelle.

Which part of the system (e.g., sequent, proof tree, formalization) is inspected by the user to decide on how to continue the proof depends on the problem. However, we also learned that different users use different information depending on their personal workflow.

From an abstract perspective, the approach of inspecting the proof state, especially in KeY, corresponds to a top-down analysis of the proof: the focus moves from the specification to single goals and sequents. At the beginning of the proof process, the specification is inspected more often and the shape of the proof tree plays an important role. Later in the process, the branches in the proof tree and the sequents in the open goals become more important. Also, the complexity of the proof problem, i.e., complexity of the program and its specification influences whether the sequents of the open goals are helpful or not.

In Isabelle, the strategy *try* (that carries out the estimation about the complexity of the proof problem for the user in a simple form) and other tools and tactics (e.g., *sledgehammer*, *quickcheck*, *nitpick*, *auto*) give feedback about the goal state. If the tactics cannot find a proof, the resulting goal states have to be inspected by the user. However, Isabelle does not provide information about the used rules or lemmas leading to an open goal. As stated especially in the Isabelle group, it is a matter of experience to decide how proof search should proceed.

The comments on the feedback mechanisms of the proof systems support our hypothesis: the user has to understand the system’s proof. The different proof artifacts are inspected and the user tries to recognize certain familiar shapes, for which he or she knows from experience how to continue in the proof process.

Proof granularity in the local proof process. One part of our hypothesis is that the granularity of the automatic strategies as presented to the user does not match the granularity in the user’s proof model.

When the application of automatic strategies and tools does lead to open goals, information about used lemmas or rules is often missing. One example is the tactic *auto*: if the tactic leads to a closed proof, showing only a single proof step is appropriate. If the tactic does not contribute to finding a proof, the proof system does not provide information about the concrete proof rules it applied and the resulting intermediate states (although this information may be available internally). Only the remaining goal states are presented to the user. Better feedback is provided by sledgehammer, as it displays the lemmas used in the underlying proof performed by an SMT solver.

Granularity of the proof and feedback of single steps also plays a role when publishing or refactoring a proof depending on the intended audience. In user-constructed proofs, Isabelle allows different levels of granularity. Often, proofs in Isabelle are more fine-grained than proofs on paper.

In KeY, there are three different granularity levels:(a) each rule application individually, (b) using the full automatic strategy, and (c) proof macros together with one step-simplification as middle-course. It seemed that proof macros are a preferred way of proving. However, they are not applicable in every proof situation.

In both systems, the granularity of the proof steps can be too fine-grained or too coarse-grained, depending on the proof situation (e.g., failed proof attempts) and the purpose of the proof (e.g., publishing a proof). We conclude that there should be a compromise between the two extremes, e.g., a mechanism that allows gaining insights into the Isabelle tactics, if required. For the KeY system, a mechanism would be useful that summarizes steps in the proof tree and only unfolds them on user inspection. This mechanism would extend existing mechanisms that collapse/unfold certain kind of proof nodes such as intermediate steps or closed proof branches.

Time-consuming tasks during the proof process. We suspected that inspecting open goals resp. finding relations between different proof artifacts would be time-consuming tasks. In order to find arguments for or against this assumption, we asked for time-consuming actions in the proof processes. As mentioned above, in the global process the modelling and specification task is time-consuming, as well as the proof attempts in the local process. Additionally, when the user wants to minimize the number of proof attempts in the local proof process, the setup for the automatic strategies is time-consuming in both systems. Other time-consuming tasks that were mentioned are the decision when to reconsider the whole model, proof refactoring (in Isabelle), and model refactoring (in KeY). In the local proof process, the following time-consuming actions are related to *understanding the proof state*: analyzing open goals, finding counterexamples, identifying the cause of a failed proof, as well as systematic proof exploration (in KeY), and *find_theorems* and proof exploration by using apply scripts (in Isabelle). These answers support our hypothesis, as they provide evidence that understanding the proof state is a laborious task. Also, other costly tasks were mentioned: automatic proofs (as the user has to wait for the prover) and trivial repetitive instantiations on different branches (in KeY), as well as redoing a proof and especially finding the correct point to which to backtrack before correcting the model or specification. In Isabelle, cleaning up proofs takes time as well.

Conclusion

Our observation is that many answers focused on understanding the proof state. For example, Isabelle users spend a lot of time cleaning up their proofs to make them accessible and understandable for other users. The answers related to the topic "understanding the proof state" in the regarding time-consuming actions also support this observation. To conclude, the answers support our hypothesis that understanding a proof is a central and important task in theorem proving. The participants spend time on understanding the proof state in order to be able to proceed with the proof or find the cause for a failed proof attempt. Comprehending the proof state is also necessary for proof exploration, e.g., when the user only has parts of the proof process in mind or when the user does not know how to start or proceed.

4.3.4. Mechanisms Supporting Proof State Comprehension

Prior to the discussion, we developed paper mock-ups of mechanisms for both verification tools which we believe may aid the user in understanding the proof (state) and therefore help to overcome the discrepancy between the user's proof model and the actual proof performed by the verification system. The mock-ups were presented to the focus groups as a sequence of screenshots that show how to invoke the mechanism and the effect of the mechanism in a particular proof situation.[3] Our intention was to gain feedback whether our developed mechanisms are comprehensible, serve our intended purpose (bridge or reduce the gap) and are of interest for the participants. The task for the participants was to describe the purpose and effect of the mechanism (as they saw it) and share their opinion about it.

Tracing terms/formulas/variables.

We showed two mock-ups (designs) for each system for the mechanism of tracing the origin of formulas respectively variables in an open goal: In Isabelle, we showed the parent formula of an open goal with renamed variables. Additionally, the relation between the original and the renamed variables was depicted. As a second mock-up, we showed a state with a number of open goals. By clicking on one of the goals, some of the used lemmas and definitions leading to that goal were shown.

For the KeY system, the starting point for both designs was the same: we selected one (sub-)formula of the sequent in an open goal. Then, for the first design, we showed a new window depicting the selected formula and its ancestors up to the original proof obligation (we summarized some of the intermediate parent formulas to not clutter up the screen). In addition, the names of the rules producing the formulas were given. The top-most parent shown was the part of the specification from which the formula originated. In the second design, we did not use a new window, instead we highlighted the parents in each inner node of the proof tree up to the root (which contains the original proof obligation).

[3]Examples of these screenshots can be found in the Appendix A.1. The full list of screenshots can be found `http://formal.iti.kit.edu/~grebing/SWC`.

When the groups where shown the mock-up of the mechanism for tracing formulas, the first reaction was clearly positive, particularly in the Isabelle group for the first mock-up. Almost all participants intuitively understood the mechanism. One participant reported that he simulates this mechanism by manual “reverse-renaming” in an external text editor. However, the question came up whether the additional information may be confusing or clutter the screen. It was suggested to implement the mechanism carefully, possibly using mouse-over tags and – in particular for KeY – include it into the existing GUI concept.

Inspired by the second mechanism for Isabelle (showing the used lemmas) some participants stated that it would be useful to have a mechanism showing the path or case distinctions leading to selected open goals on demand.

The second design in the KeY group triggered a new idea: some participants suspected a filtering mechanism and discussed about filtering the sequent and the proof tree.

What needs to be proven?

For the Isabelle system, a mock-up was given, showing which lemmas and theorems contribute to a proof (depicted as a simple coloured graph). Unproven lemmas were coloured red, lemmas whose proofs used unproven lemmas were coloured orange, and fully proven lemmas were coloured green. The lemmas already proven were depicted with a box with an ellipsis as description. The red and orange boxes were labelled with the name of the lemma that still needs to be proven resp. uses unproven lemmas. The participants described the mechanism as separating the used from the unused lemmas and that it would be useful in combination with, e.g., the automatic strategy *simp*.

Most of the participants showed a positive reaction to this mechanism. Some participants would prefer a textual representation of the used and unused lemmas. The design of our mock-up can be improved in general. The level of detail should be chosen carefully in order not to clutter up the screen (e.g., fold proven lemmas with the option to unfold) and the view should be hierarchic.

What happened during the proof process?

For the KeY system, we presented a mock-up that showed a comparison mechanism (also called *diff mechanism* in the following) relating two nodes in the proof tree (not necessarily adjacent nodes). We designed the mock-up such that all unchanged parts of the sequent were blurred out and the relevant changes were shown directly above each other. The participants needed some time to understand the idea and the blurring was found to be confusing, as the presentation of two different sequent parts can be mistaken as belonging to the same single sequent.

One participant noticed that something similar is implemented in the KeY system already as string diff mechanism, where the diff between two sequents is shown in one new window. However, this participant also mentioned that the mechanism needs improvement, which contributes to our idea that such a functionality should be implemented in the KeY system.

Already during the discussion, ideas for improvement came up, e.g., that the diff between two sequents should be shown in two windows adjacent to each other or above each other. Also, similar to a text-diff viewer, the changes should be marked using colours or typographical representations. Moreover, the two nodes which are being compared should be marked in the proof tree.

In conclusion, we suggest for future work to develop a user-configurable diff mechanism which shows the two sequents being compared in two windows. One window depicts the old sequent and one depicts the new sequent. In addition, the algorithm for comparing two sequents needs to be chosen carefully and has to consider the tree structure of the sequent. A string diff algorithm is not sufficient for comparing tree-shaped sequents, as certain differences are recognized in the wrong way. For example, it is wrong to assume that replacing n by `null` results from appending `ull` to n.

4.3.5. The Ideal Interactive Proof System

As a cool-down task, we asked the participants to name properties that an ideal interactive verification system should or should not have. Our goal here was twofold – we wanted to collect more ideas about desirable features of ITPs and evaluate find arguments for or against or hypothesis (Section 4.1) at the same time. We decided to omit comments that were of technical nature (e.g., "It should not have memory leaks.") as well as points that have already been mentioned in previous parts of the discussion.

Intuitive proof process. Both groups wished that an ideal interactive proof system would produce proofs "close to what an experienced user would expect."

This supports our paradigm of reducing the gap resp. keeping the gap small between the user's model of the proof and the ITP's current proof state.

Understandable proof states. The focus group on KeY prefers more interaction in terms of the original proof obligation (e.g, specification and program) while the Isabelle group wishes for semi-automatic proof steps (instead of the fully automatic tactics). In our opinion this illustrates that too many as well as too few details may have a negative effect on the understandability of the ITP.

Convenient interaction. One important feature that was wished for by both groups is a good performance of the ITP. The performance can impede usability if the user needs to wait too long between interaction steps.

Conclusion

In summary, participants of our focus groups asked for an ITP that (i) produces intuitive proofs, (ii) can present proof steps in an understandable way (and give counterexamples if the proof cannot be closed), and (iii) provides a convenient interface for interaction.

Table 4.4.: Time and effort needed for conducting the two experiments (approximate).

script preparation	20–40	hours
recruiting participants	10	hours
briefing moderators	40	hours
technical setup and testing	8	hours
discussions	5.5	hours
transcription of recording	20	hours
analysis	> 240	hours

4.4. Discussion

Meta-level: Using Focus Groups to Evaluate ITPs. Our experiments have shown that the focus-group method is not just for business software but can be applied successfully to specialized tools such as ITPs. We gained lots of insight from our experiments.

Focus groups are well suited for an explorative and qualitative investigation of strengths and weaknesses in usability and the usefulness of new features. They are particularly useful for systems with a relatively small user base such as ITPs. Focus groups are a huge step towards objective experiments in the area of usability that can be repeated in a uniform fashion due to the pre-formulated script. However, focus groups do not provide precise quantitative data. Another strength of focus groups is that participants voice ideas and discuss issues that they would not have talked about in single interviews, when topics are brought up by other participants. Our experience is that we gain detailed feedback to usability issues of the systems. In addition, the discussions provide an understanding of how users use the system to achieve their goals.

The effectiveness of focus groups and their advantages over unstructured discussions, however, do not come for free. Conducting focus group discussions takes careful preparation and is a non-trivial task. The effort and the required time is considerable. The approximate work-load of our experiments is given in Table 4.4 (person-hours for both experiments, not including the participants' and moderators' effort).

Discussion of the Concrete Insights. We identified strengths and weaknesses of the two systems as mentioned by the participants, which mostly can be generalized to most or all ITPs. A typical weakness is, for example, an inadequate understanding of what the effect of automatic proof search strategies is. Users may lose the comprehension of the proof by applying automatic strategies, as in some cases the strategies do not give feedback on which rules or transformations they apply and leave the user with a proof state that differs from the last state seen.

Moreover, technical issues that are annoying for the user and compromise efficiency were mentioned, e.g., unstable proof loading mechanisms or a user interface that is not sufficiently reactive. These answers point to where to focus on when improving the systems' usability.

Discussions about the proof process gave us first insights into the feedback mechanisms provided by both systems during the proof process, e.g., the different automatic tools in Isabelle. Also, issues that arise during the proof processes have been mentioned by the participants and ideas for improving certain aspects. Ideas mentioned by the participants included, for example, the presentations of the proof tree in KeY.

We gained first insights into how the users use their systems to accomplish certain proof tasks and where they switch to other systems in order to get a better understanding of the current proof state, for example by using an external text editor. By showing mock-ups of potential improvements we gained lively feedback and opinions about the presented mechanisms. From the opinions we tried to draw conclusions about which mechanisms are more desired than others and thus may be more supportive and subject for future work to develop.

The creative task at the end of the discussion led to interesting and creative interaction mechanisms for ITPs, but also generally desirable features for ITPs have been mentioned. Some of these features are already part of the systems; others need improvement.

4.5. Conclusion and Future Work

We have presented a qualitative user study using focus groups to evaluate the usability of interactive theorem provers (ITPs) with a rather small user base. Our goal of the two user studies using focus groups was to explore whether a gap between the user's model of the proof and the system's current proof state exists and whether this gap is a central problem for the usability of ITPs. In addition, we have developed mock-ups for mechanisms that aim to bridge this gap or keep it small. We have developed a first model of the proof process with the focus on the relation between the user's (partial) model of the proof and the current proof state.

We made the experience that this method is suitable to gain insight into the usability of ITPs and that we can formalize first hypotheses about usability issues of the systems, as well as a first collection of desirable features for ITPs. We have also shown that it is possible to perform this evaluation without expert moderators, when being aware of this fact during the analysis of the transcribed material.

As a result of this evaluation we have found evidence that our model of the proof process is reasonable and our findings indicate that the gap between the user's model of the proof and the system's current proof state is a central problem in interactive theorem proving.

At the same time, while the model already captures a lot of peculiarities of the proof process, it does not fully represent the full complexity of interactive proof search yet. We have, for example, encountered related activities of the users and issues in the verification task, such as the use of counterexample generators or finding the correspondence between the current proof state and the program (in the KeY system), that clearly show that our model does not capture all the details of proving yet. In Chapter 5, we present a user study where we investigate the proof process and the gap in more detail by conducting semi-structured interviews with users of the KeY system.

The user study also helped us to discover other usability issues in the systems not related to our hypothesis. These are often either technical or relate to other topics, e.g., performance of the automatic strategies. We believe that attention has to be drawn to these as well to enhance the user experience for ITPs.

In another part of the user study we presented mechanisms that should help to bridge or reduce the gap by concentrating on providing the user insights into what happened during the automatic proof search. The participants reacted positively towards the mechanisms and provided feedback for improvements or new ideas, such as user defined filter mechanisms for the proof tree in KeY.

The mechanisms that attracted interest during the discussions need to be further developed and prototypically implemented for future work. To ensure that the mechanisms suits the users needs and to evaluate whether they increase the usability, further user studies need to be conducted. In our user study presented in Chapter 5 we have shown a first mock up version of one of the mechanisms that was presented to the KeY users in the focus group discussions to investigate whether the mechanism is supporting users in bridging the gap. Another task for future work is to extend the proposed mechanisms, prototypically implement them in the KeY system and perform usability tests to evaluate the solutions. Additionally, the model proposed in this chapter could be extended to take into account that there are also different proof strategies for one proof and it is often user-dependent which proof style is used for a proof.

5. User Study: Interviews with Practical Tasks

Contents

5.1. Introduction

In this chapter we will present details on our explorative formative user study with users of the KeY system in which we performed interview sessions together with practical tasks. Our goal of performing the user study was (a) to gain insight into the proof process using the KeY system and (b) to explore whether a new mechanism, prototypically introduced into KeY, helps the user in bridging the gap between the concrete proof state and the model of the proof. We also wanted to gain insights into further room for improvement of the target of evaluation. We planned a session time of approximately 70 minutes for each interview session. We structured the usability test into different phases[1]: introduction, warm-up, task and cool-down phase (as also introduced in Chapter 2). In the *introduction phase* the users were interviewed by the moderator about their experiences using the KeY system. The *warm-up phase* started with an interview about the proof process of the participants using the KeY system. Then the participants were asked to specify and verify a Java method within the time frame of 10-15 minutes. We did not restrict the usage of system features in the warm-up phase. Our intention for this phase was to get insight into how users of KeY use the KeY system to find a proof.

Based on the focus group discussions presented in Chapter 4, we prototypically implemented a mechanism to support the display of the origin of a formula in the KeY system: It allowed the user to select a formula in the open goal and retrieve the path from the open goal to the original proof obligation in which the formula was affected by rule applications, in the following also called *origin of a formula*. Our intention with this mechanism was to support users to bridge the gap between the user's model of the proof and the current proof presented in KeY, as the user is able to trace back the history of a selected formula and see the changes during the proof process. A more detailed description of this mechanism can be found in Section 5.2.

For the main part of the interview session we included a *task phase*. For this task phase we developed tasks that should help to explore opinions about the mock-up version of our mechanism. We divided this phase into two parts with two different tasks each, one task with and one task without the possibility to use the mock-up version of the mechanism. One of the two different task types involved showing the user a partial proof for a proof obligation in first-order logic, obfuscating the predicate and function symbol names. The second task type involved a partial proof for the correctness of a method contract of a Java method.

For both types of tasks and both parts of the task phase the questions were identical: The user should describe the proof situation, they should name the history of two formulas of the open goal and name the next step to continue the proof process. At the end of the task phase the users were asked about their expectations about parent formulas of a given formula and proof.

In the *cool-down* phase participants were interviewed again about the new mechanism and possible application areas in the context of proving a program correct. Furthermore, we asked about room for improvement of the verification system.

[1]See `http://formal.iti.kit.edu/~grebing/SWC` for the full script in German.

Our explorative, formative user study contained questions on a broad range of abstraction levels, depending on the kind of information to be gained: concrete questions, e.g., on the history of given formulas on a sequent, with the purpose of improving our implementation of the history mechanism, as well as more abstract questions to determine whether such a history mechanism might be considered useful by users of a verification tool.

To which degree the proposed history mechanism simplifies the verification task in practice has to be investigated by performing detailed experiments as part of future work.

5.1.1. Research Questions For This Study

Our overall goal was to explore information KeY users are looking for in situations where the proof process stopped in order to gain orientation in the proof. Additionally, we wanted to explore opinions about issues and features of the KeY system concerning the usage for proof construction and proof comprehension.

This study was carried out to learn about:

1. Which activities have to be carried out in order to achieve the overall goal of proving a program correct with respect to its specification?
2. What is the workflow of the KeY users (including the modification of the software and the specification)?
3. Can we identify distinct phases or activities?
4. Can we identify patterns or strategies the participants used, especially to analyze unfinished proof attempts, respectively to find the orientation after the application of the general automatic proof search strategy of KeY?
5. What is the intuitive meaning of *origin of formulas and terms* in the context of interactive program verification with explicit proof object?
6. Where is room for improvement in general for the process, in the user interface and concerning mechanisms and features supporting the user in analyzing unfinished proof attempts?
7. Which other issues were mentioned about the usage of KeY?

Data Analysis For the data analysis parts of our user study, the aforementioned research questions translate into the following topics on which we wanted to gain information about. The data we have collected during our user study has been evaluated and analyzed according to these topics.

- the actions/activities users perform in the proof process,
- what information do users consider in general and how do they approach to understand proof situations,

- the information users are using in the proof process to decide which next action to perform,
- the support the KeY system provides in the proof process, and which support is missing in the proof process,
- the (usability) aspects of the mocked functionality showing the origin of formulas in the proof context, and
- general room for improvement in the KeY system.

5.2. Prototypical History Mechanism

In the focus group discussions for the KeY system we included paper mock-ups for a tracing mechanism for formulas in the proof process to support users in understanding the current proof situation and retracing the origin of formulas.

The designs that were discussed lead to suggestions for improvement. When adding a new view in an existing graphical user interface of a verification tool, special care has to be taken to not clutter up the screen and to integrate it into the already existing user interface concept. Additionally, a discussion about filtering the proof tree had taken place.

With these results we developed a mock-up implementation to show the history of formulas in the KeY system in a pre-defined proof situation. In the following we will introduce the concept for this mechanism in more detail. The prototypical mechanism was then used in the user study with KeY users in the task phase.

5.2.1. Origin of Formulas

Based on our first hypothesis (presented in Chapter 4.1) that when proving the correctness of programs with support of a verification system a gap between the model of the user about the current proof state and the current proof state in the proof system exists, we have developed a concept for a mechanism that aims to support the user in reducing the gap by providing additional information about the origin of formulas among different proof nodes.

There are different possibilities to define the origin of a formula in a program verification proof which involves different abstraction levels. The origin of formulas can be defined with respect to the proof input artifacts a user provides to the verification system on a global level, i.e., the program and its specification. The origin *specification* may further be refined to the different specification constructs. In JML, that would be for example the pre- and postcondition, the loop invariant or the framing condition. During the generation of the proof obligation also implicit assumptions are added to the proof obligation, for example that specific objects are non-null. This may additionally be an origin. We will call this kind of origin the *semantic* origin.

The second possibility to define the origin of a formula can be done with respect to the current proof structure, which we will also refer to as *syntactic* origin. We use an origin relation that is closely related to the definition of *ancestors* and *immediate descendents* given by Buss [Bus98].

We will introduce the terminology according to the terminology used in the context of the KeY system. Concluding this description, we will relate the terms used in our concept to the definitions of Buss – this implies that we view the proof tree in a sequent calculus with the root at the top and the leaves at the bottom, in contrast to Buss. The orientation of the notation for calculus rules however, is the same in both views.

Recall the structure of a sequent $\Gamma \Rightarrow \Delta$ from Chapter 3.4.1 and the general schematic sequent calculus rule where a rule may have none, one or n premises and one conclusion:

$$\mathit{ruleName}\ \frac{\Gamma, G_1, A \Rightarrow B, G_2, \Delta \quad \dots \quad \Gamma, G_{n-1}, A \Rightarrow B, G_n, \Delta}{\Gamma, f, A \Rightarrow B, \Delta} \tag{5.1}$$

In this general structural calculus rule $\Gamma, \Delta, A, B, G_1, \dots, G_n$ are possibly empty sets of formulas. In KeY, the sets Γ, Δ are called *context.* We call the formula f in the rule's conclusion *focus formula.* In the KeY system, calculus rule schemas are realized as taclets, where the focus is realized using the *find* expression [RU16]. A taclet in the KeY system can contain at most one focus. This limitation is considered in the following.

A formula in one of the sets $G_1, \dots, G_n$ will be called *auxiliary formula.* The sets of formulas A, B are called *assume* formulas. Not all rules contain assume formulas.

For the origin relation, as considered in our concept, we will adapt the definitions of Buss and define for an *auxiliary formula* $g \in G_1 \cup \ldots \cup G_n$ in the rules premises, the *direct parents* as:

- the focus formula f of the rule's conclusion and
- every *assume* formula $a \in A$ and $b \in B$ of the rule's conclusion.

Vice versa, g is called *direct child* of f and every *assume* formula $a \in A$ and $b \in B$.

For a formula c in the contexts of a rule's premise, Buss defines the *direct descendent* of c as the corresponding occurrence of the same formula in the contexts of the rule's conclusion, i.e., if c is the i-th formula in Γ in the rule's premise, then the direct parent of c is the i-th formula in Γ in the rule's conclusion.

In contrast, for our purposes in the definition of the direct parent relation we omit all proof steps that did not change the formula of interest for the parent relation. As a consequence, the parent of such a formula $\gamma \in \Gamma \cup \Delta$ is the same as the parent of its corresponding occurrence γ' in the rule's conclusion, resp. γ' itself if the direct parent of γ' is $\bot$. A formula may not have a direct parent in the case that it was already present in the proof's root sequent or if it is a *cut formula*, introduced by the calculus rule *cut.* We denote this special case with the symbol $\bot$. The *direct child* relation for context formulas is defined analogously to the *direct parent* relation: the child of a formula $\gamma \in \Gamma \cup \Delta$ in the rule's conclusion is the same as the child of its corresponding occurrence γ' in the rule's premise. We use $\bot$ if no *direct child* exists, i.e., for formulas in a leaf of the proof tree.

Having the *direct parent* relation, we can now extend it to the *origin* relation, which is closely related to the *ancestor relation* of Buss. The origin relation is the transitive closure of the *direct parent* relation. We define the origin of a formula f to be the chain of zero or more direct parents from f to the root sequent.

Discussions. Similar to our two concepts of parent and direct child relations, Buss defines the *immediate descendents* and the *immediate ancestor relation.*[2]

Our definitions differs in two aspects from the ones by Buss as follows. We explicitly include *assume* formulas as *direct parents* respectively *direct child*, while the definitions by Buss include these formulas in their *side formulas.* As our definition should be suitable to capture the intuitive meaning of *a formula is a parent of another formula* for the user, we considered that all formulas that are involved in a rule application should also be considered as parents, as if one of the formulas is missing on the sequent the rule application is not possible.

The second difference between our definition and the one by Buss is that we omitted the proof steps that do not contain changes to context formulas for the *parent relation.* This decision had mere practical reasons, as we wanted to use the definition to filter the proof tree view in the KeY system according to the *origin relation*, such that only those nodes are shown contain changes either to a selected formula or to one of its parents.

5.2.2. Mocked Mechanism

For the user study we modified the KeY system such that we were able to use a mock-up for the mechanism. For the specific tasks we added a menu entry that filtered the nodes of the proof tree, such that only those nodes were visible that either were branching nodes on the path from the goal node to the root node or contained a change to the formula that should be tracked for its origin.

We chose to include the branching nodes, such that the participants still retrieve some context information about the location of the open goal.

Furthermore, for the mocked mechanism we selected the nodes that should be included into the filtered view by hand.

As our intent was to gain insight whether such a mechanism would support the users we chose to display a rather coarse relation. That is, we highlighted the whole top-level formula even if only a subterm in this formula was affected by the change. This especially lead to the highlighting of the full sequent in the root node.

For the mocked mechanism we also used red as coloring, as we wanted to draw the participant's attention to the highlighted formula. For a refined, usable version of this mechanism it is necessary to draw the attention to a color scheme that is suitable for users with color vision deficiency and that still draws the user's attention to it.

In the first screenshot in Figure 5.1(a) we depicted the invocation of the mechanism using the context menu in KeY as it was accessed by the participants in the study.

[2]In contrast to our notion of a proof tree, Buss considers the root of the proof tree to be at the bottom in a graphical representation, hence the different naming of the relations compared to our definitions.

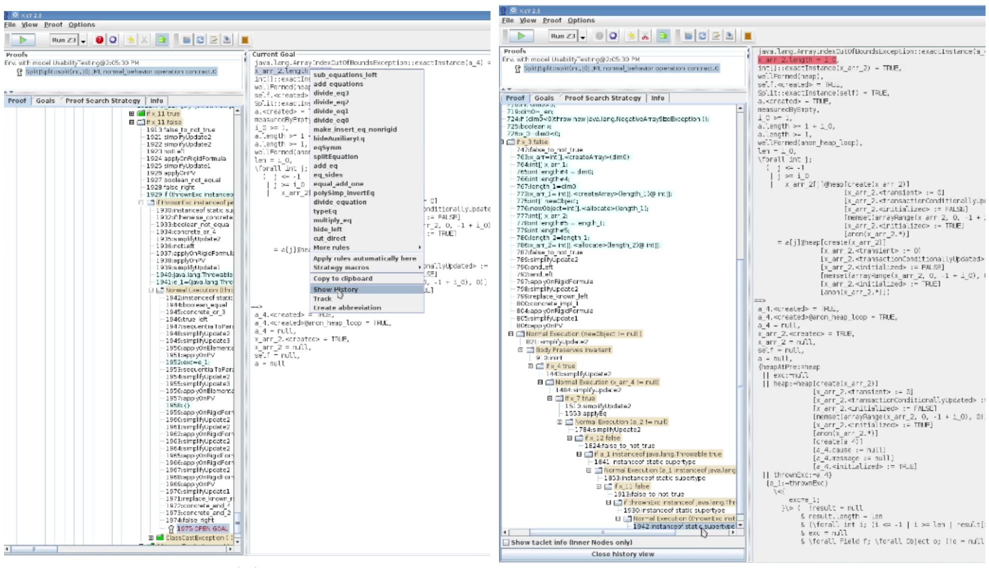


Figure 5.1.: Invocation of *Show History* on a formula (a) and the resulting view change (b).

In the second screenshot in Figure 5.1(b) the result of the invocation is depicted. The proof tree is filtered and only branching nodes as well as the nodes that contain immediate descendents of the selected formulas are shown in the tree. Selecting a node in the filtered tree shows the inner sequent of that node. The direct descendent is highlighted on the sequent.

We chose to include the mechanism into the current interaction concept in KeY such that the participants are able to use almost all of their preferred ways of navigating the proof tree and inspecting the sequents. Limitations of the mock-up have been that it did not show a fine-granular history for symbols or (sub)terms and that the regular filtering functionalities, such as the option `Hide Intermediate Proof Steps` did not fully work with the mock-up.

5.3. Methodology

We carried out an explorative user study with users of the KeY system. The user study consisted of two interwoven parts: a semi-structured interview and practical tasks the users should perform using the KeY system. We included tasks where the users were only able to use a limited set of KeY's features – features that do not change the proof state. The key idea behind this limitation was to explore whether all information for estimating and assessing the current proof state is presented by the system, such that the user does not need to change the proof state. This decision was made, because we consider that certain state changes require the user to gain orientation in the proof and according to our assumptions this may require effort from the user.

We also included tasks where the users were able to use a mocked mechanism that showed them information about the history of formulas during proof construction (described in 5.2.2). The key idea here was to explore whether this view on the proof state is helpful in the comprehension and estimation of the current proof without the need of changing the proof state.

The actions on the screen and the voices of the participants were recorded. For backup purposes we also recorded the voices of the participants with a second recording device.

To avoid experimenter-bias, the interview has been carried out by a student assistant (master-level) who was familiar with the very basics of the KeY system and of formal verification but was not involved in the design of the user study. Therefore, the interviewer was able to ask more basic questions to gain a more profound understanding of the contents the participants were talking about. If a domain or tool-expert had asked such questions the situation might have not been authentic for the participants.

On the other hand this setup also is liable to errors in the interview itself, e.g., that the interviewer misses seeing problems during the test. To overcome this liability the test-manager was part of the team as well, but was only allowed to interfere if a technical issue arose.

Participants

The profile of an ideal user for our study is someone who uses the KeY system in their daily work and is not part of the KeY development team. Additionally, the user should have mastered the learning curve of the KeY system, to be able to use the system for complex verification tasks effectively. We therefore intended to recruit intermediate and expert users of the system, as novice users, may not have overcome the learning curve for the system and may not have acquired the full necessary domain knowledge to use the program verification system effectively.

However, users with such a profile are rare among the user base of the KeY system. Firstly, KeY is a system being developed in academia where its purpose is also to showcase new research results in the field of software verification. Therefore, users of KeY often also develop extensions for KeY or at the core of the system. Additionally, KeY is an expert system where the task of verifying programs correct requires a certain level of expertise and experience. These two conditions already limit the user base drastically, compared to general purpose software systems. With our requirement to only include intermediate and expert users we limit the chance of finding suitable participants even more.

We therefore loosened the criteria for the selection of participants by allowing users who used the system for case studies and who also implement in the system. Additionally, we also allowed participants to take part in the study who develop at the core of the system or extensions of the system. We are aware that these participants can be biased in their opinions about specific parts of the system. However, to be able to extend the system with new methods a certain knowledge about the usage of the system and the domain is necessary and in many cases these participants used the system for the verification of programs (e.g., in case studies) before.

The loosened selection criteria for participants have been the following:

- intermediate or expert user of KeY,
- (daily) usage of KeY for program verification tasks or case studies, and
- may be part of the KeY development team.

To this end we have been able to recruit nine participants. This number may seem rather small, especially if compared with user studies for general purpose software systems. However, for performing an explorative study where the goal is to study the context of use and uncover issues in using the system and the newly created mechanism the number of participants were sufficient. To make statistically relevant propositions about the usability of the system or perform a summative user study, a quantitative study has to be performed.

Test-Design

As the task of specifying and verifying a piece of software may have a large completion time, we chose to:

1. Leave out the specification task, which is the first part before verifying a program correct.
2. Start at specific points in the verification process and describe to the user what has happened before, to give the user information about the situation in the proof process
3. We did not force the users to successfully end the task of proving a program correct, as this may take a while. We rather stopped the task either when an error was found, a subtask was successful or the planned duration time for the task was reached. If the planned duration time was reached we did not suddenly interrupt the participants but rather chose a situation that was appropriate to end the task.

To evaluate the study we used qualitative content analysis, similar to the methods described by Kuckartz [Kuc14]. We used an inductive categorization: we started by associating the given answers to the questions and in the context of the posed question we categorized the answers. The inductive categories were formed alongside with the script and the comments by the participants.

The user study also contained tasks that required the participants to use the KeY system for accomplishing parts of verification tasks. To be able to analyze the way the participants performed the tasks, we chose to use sequence models (see Ch. 2.5.1) to capture the sequences of actions we observed during the study.

5.4. Script Design

In the following sections we will introduce the script and tasks used in the user study.[3] We will give details on the purpose of the questions and insights into the tasks the participants were asked to approach during the user study.

Script: Introduction

The questions in the introduction part of the test served the purpose of categorizing the participants according to their experience with the KeY system.

The questions have been:

1.1 How long do you know the KeY system?

1.2 Since when do you use the system for more than toy examples?

1.3 In the last year, how many hours did you use the system on average for solving proof problems?

1.4 If you are also a developer of this system, which are your tasks?

1.5 Please rate your expertise with this system. Would you rate yourself as a novice, intermediate or expert user? Why would you classify yourself into this category?

Script: Warm-Up Tasks

For the warm-up task we showed a picture of a proof situation in the KeY system with many open goals and a large sequent as visual cue to trigger the conversation about open proof attempts (see Figure 5.2). The following description was given to the user before:

> To warm-up we would like to ask you to put yourself into the situation that you are sitting in front of the KeY system and you have reached a proof state in which the automatic strategies can not be applied anymore. You have many open branches and a quite large sequent.

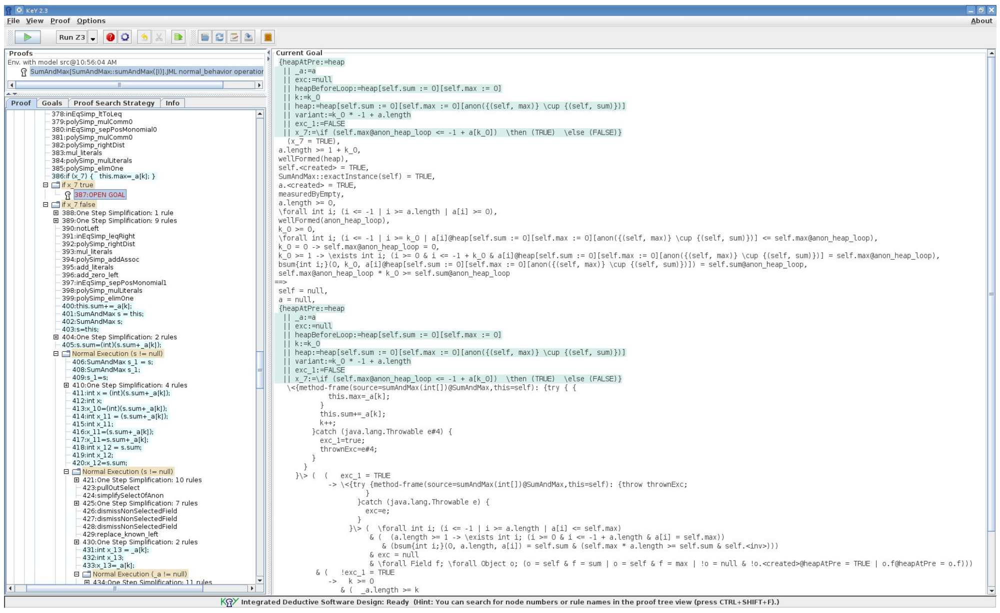
Questions that were asked in the warm-up part of the test were:

2.1 What could have happened? What could be the reasons that KeY opened a lot of proof branches and was not able to close them?

2.2 How do you approach the problem of finding out what has happened and determining the next steps?

2.3 Which possibilities do you have for that? Please sort them relative to each other according to your usage of the possibilities.

[3]The original script in German can be found on the webpage `http://formal.iti.kit.edu/~grebing/SWC`. We translated the script to the best of our knowledge for this chapter.



```
Env. with model src@10:56:04 AM
  SumAndMax[SumAndMax::sumAndMax([I)].JML normal_behavior operatior
```

```
378:inEqSimp_ltToLeq
379:polySimp_mulComm0
380:inEqSimp_sepPosMonomial0
381:polySimp_mulComm0
382:polySimp_rightDist
383:mul_literals
384:polySimp_mulLiterals
385:polySimp_elimOne
386:if (x_7) {   this.max=_a[k]; }
if x_7 true
  387:OPEN GOAL
if x_7 false
  388:One Step Simplification: 1 rule
  389:One Step Simplification: 9 rules
  390:notLeft
  391:inEqSimp_leqRight
  392:polySimp_rightDist
  393:mul_literals
  394:polySimp_addAssoc
  395:add_literals
  396:add_zero_left
  397:inEqSimp_sepPosMonomial1
  398:polySimp_mulLiterals
  399:polySimp_elimOne
  400:this.sum+=_a[k];
  401:SumAndMax s = this;
  402:SumAndMax s;
  403:s=this;
  404:One Step Simplification: 2 rules
  405:s.sum=(int)(s.sum+_a[k]);
  Normal Execution (s != null)
    406:SumAndMax s_1 = s;
    408:SumAndMax s_1;
    409:s_1=s;
    410:One Step Simplification: 4 rules
    411:int x = (int)(s.sum+_a[k]);
    412:int x;
    413:x_10=(int)(s.sum+_a[k]);
    414:int x_11 = (s.sum+_a[k]);
    415:int x_11;
    416:x_11=(s.sum+_a[k]);
    417:x_11=s.sum+_a[k];
    418:int x_12 = s.sum;
    419:int x_12;
    420:x_12=s.sum;
    Normal Execution (s != null)
      421:One Step Simplification: 10 rules
      423:pullOutSelect
      424:simplifySelectOfAnon
      425:One Step Simplification: 7 rules
      426:dismissNonSelectedField
      427:dismissNonSelectedField
      428:dismissNonSelectedField
      429:replace_known_left
      430:One Step Simplification: 2 rules
      431:int x_13 = _a[k];
      432:int x_13;
      433:x_13=_a[k];
      Normal Execution (_a != null)
        434:One Step Simplification: 11 rules
```

```
 {heapAtPre:=heap
  || _a:=a
  || exc:=null
  || heapBeforeLoop:=heap[self.sum := 0][self.max := 0]
  || k:=k_0
  || heap:=heap[self.sum := 0][self.max := 0][anon({(self, max)} \cup {(self, sum)})]
  || variant:=k_0 * -1 + a.length
  || exc_1:=FALSE
  || x_7:=\if (self.max@anon_heap_loop <= -1 + a[k_0])  \then (TRUE)  \else (FALSE)}
   (x_7 = TRUE),
 a.length >= 1 + k_0,
 wellFormed(heap),
 self.<created> = TRUE,
 SumAndMax::exactInstance(self) = TRUE,
 a.<created> = TRUE,
 measuredByEmpty,
 a.length >= 0,
 \forall int i; (i <= -1 | i >= a.length | a[i] >= 0),
 wellFormed(anon_heap_loop),
 k_0 >= 0,
 \forall int i; (i <= -1 | i >= k_0 | a[i]@heap[self.sum := 0][self.max := 0][anon({(self, max)} \cup {(self, sum)})] <= self.max@anon_heap_loop),
 k_0 = 0 -> self.max@anon_heap_loop = 0,
 k_0 >= 1 -> \exists int i; (i >= 0 & i <= -1 + k_0 & a[i]@heap[self.sum := 0][self.max := 0][anon({(self, max)} \cup {(self, sum)})] = self.max@anon_heap_loop),
 bsum{int i;}(0, k_0, a[i]@heap[self.sum := 0][self.max := 0][anon({(self, max)} \cup {(self, sum)})]) = self.sum@anon_heap_loop,
 self.max@anon_heap_loop * k_0 >= self.sum@anon_heap_loop
==>
 self = null,
 a = null,
 {heapAtPre:=heap
  || _a:=a
  || exc:=null
  || heapBeforeLoop:=heap[self.sum := 0][self.max := 0]
  || k:=k_0
  || heap:=heap[self.sum := 0][self.max := 0][anon({(self, max)} \cup {(self, sum)})]
  || variant:=k_0 * -1 + a.length
  || exc_1:=FALSE
  || x_7:=\if (self.max@anon_heap_loop <= -1 + a[k_0])  \then (TRUE)  \else (FALSE)}
   \<{method-frame(source=sumAndMax(int[])@SumAndMax,this=self): {try { {
              this.max=_a[k];
            }
            this.sum+=_a[k];
            k++;
          }catch (java.lang.Throwable e#4) {
            exc_1=true;
            thrownExc=e#4;
          }
        }
      }\> (  (   exc_1 = TRUE
            -> \<{try {method-frame(source=sumAndMax(int[])@SumAndMax,this=self): {throw thrownExc;
                    }
                  }catch (java.lang.Throwable e) {
                    exc=e;
                  }
                }\> (  \forall int i; (i <= -1 | i >= a.length | a[i] <= self.max)
                    & (  (a.length >= 1 -> \exists int i; (i >= 0 & i <= -1 + a.length & a[i] = self.max))
                       & (bsum{int i;}(0, a.length, a[i]) = self.sum & (self.max * a.length >= self.sum & self.<inv>)))
                    & exc = null
                    & \forall Field f; \forall Object o; (o = self & f = sum | o = self & f = max | !o = null & !o.<created>@heapAtPre = TRUE | o.f@heapAtPre = o.f)))
         & (   !exc_1 = TRUE
            ->   k >= 0
               & (  _a.length >= k
```

```
KeY Integrated Deductive Software Design: Ready  (Hint: You can search for node numbers or rule names in the proof tree view (press CTRL+SHIFT+F).)
```

Figure 5.2.: Visual cue at the beginning of the user study

2.4 In this situation, are there other alternatives or do you miss a mechanism that would be more suitable than the ones already existing?

2.5 If you could wish for a functionality that would support you in proving problems using the KeY system what would it be?

Key Ideas

With this part of the user study we wanted to cover the situation users are in when constructing a proof by applying KeY's automatic proof search strategies and being now faced with a proof state where user interaction is required in order to proceed with the proof process. With question 2.1 we wanted to explore what the expectations of KeY users are when reaching such a situation. Where do KeY users suspect the cause for the open proof. With question 2.2 we wanted to explore the user's workflow for comprehending the proof state and determining the next successful actions in the proof process. Questions 2.3, 2.4 and 2.5. cover the functionalities KeY offers or should offer in such a situation to comprehend and assess the proof state.

Script: Task – Beginning of the Proof Process

The first practical task in our study where the participants should use the KeY system was to start a proof process. We provided the following explanation for the proof task.

Listing 5.1: Proof task: removeElem()

```
public class RemoveElem(){

  /*@ public normal_behaviour
    @ requires a!= null && a.length > 0;
    @ ensures \result != null;
    @ assignable \nothing;
    @*/

    public /*@helper @*/ int[] removeElem(int[] a, int k){
      int res[] = new int[a.length-1];

      /*@ loop_invariant res!=null;
        @ assignable res[*];
        @ decreases res.length - i;
        @*/
      for(int i = 0; i<res.length; i++){
        if(i<k-1){
           res[i]=a[i];
        }
        if(i>=k-1){
          res[i] = a[i+1];
        }
      }
      return res;
    }
}
```

We would like to solve proof tasks using the KeY system. We have implemented a method with an array and an integer number k as parameters. The method should remove the k-th element from the array and return the rest of the array. For this we have started a first simple specification.

3.1 Please prove that the method `removeElem()` fulfills its contract. Please proceed as you would proceed in other verification tasks. Please, complement the specification or the program if necessary. Please, think-aloud what you are searching for and explain before you click the reason for clicking on that position.

After the explanation the participants were shown the first proof state in the KeY system after loading the proof problem. The actual proof problem which is shown in Figure 5.1 was provided if the participants asked for it.

The specification of the method `removeElem()` in this task does not state the property given in the explanation, however it is a valid specification. So it is provable if the participants use the automatic strategies of KeY. After encountering that participants may not ask to see the specification first (which happened with one of the first participants), the moderator added the question whether the program fulfills the specification stated in the task description.

The goal was to see how KeY users use the system to develop the specification and how their individual process looks like. For example do all users first try to provide a specification that is as complete as possible or do they develop the specification iteratively using the feedback given from the KeY system. We also wanted to see, whether KeY users try to relate the proof state to the program and how they are achieving this goal. Additionally, issues arising during proof construction were interesting for us as well as ideas for improvement when stated by our participants.

It was not our goal to see how our participants fully complete this task. To proceed with the user study, we placed a time limit from approximately 10-15 minutes for this task and interrupted the participants after reaching the limit in an “appropriate moment”. We consider with “appropriate moment” a moment, where the participants are not directly interrupted in an action, but for example when they switched to the text editor or mentioned that they are clueless in a situation.

Script: Task – Orientation in the Proof and Origin of Formulas

The next part of our study consisted of four proof situations. In each situation we showed the users a proof state and asked question about the proof state.

The introduction to this task was:

> We will now turn our attention to proof situations. In the following you will see different proof situations. You should interact with the KeY system to proceed in the proof process.

In total we showed the participants four proof states: two proof states from a program verification problem and two proof states from a more abstract proof problem in first-order logic.

The task and questions for all proof situations were the same: the participants should describe and explain the situation, they should state how they would proceed and they should explain why a specific formula is present on the sequent.

By explaining the proof situation we aimed to gather information how the participants find the orientation in the proof process, after the application of automatic strategies. We wanted the participants to only use functionalities of the KeY system that change the view on the proof state but not the state itself. Our expectation was to gain insight whether the state itself is presented in a way that users can orientate themselves without disturbing the state – as changing the state may pose the difficulty of finding back to the state they had originally inspected before the change.

We divided the task in two parts: one where the participants should solve two tasks only with functionalities present in the KeY system as is and a second part where the participants were able to additionally use a new functionality that does not change the proof state but adds a new view on the proof state: *Show History*.

After task completion of the two situations without the new functionality we added a question what the participants envision under the new functionality and recorded where the participants suspected the functionality to be placed in the user interface.

Additionally, to avoid a bias in the proof task, we swapped the tasks with and without the functionality for four participants. In the following we will therefore refer to the two groups as participants of group A and participants of group B, to distinguish the task distribution in cases where we swapped the tasks among the two groups.

Program Verification – Off-by-one-Error: Method `split()`

> We present a method that returns by means of an integer number and an array the first part of the array with the length of the integer number. We have worked out a specification which should characterize this property. We have started the proof process and have pressed the play-button. The following proof state is now the result (see Figure 5.3).
>
> 3.2.1 What has happened? Please describe the proof situation to me. Where are we in the proof? What is displayed on the screen?
>
> 3.2.2 How would you proceed?
>
> 3.2.3 Please explain to me, why does `x_arr_2.length = i_0` needs to be proven? (Where does it come from?)
>
> 3.2.4 (Please explain to me, why does `a_4 = null` needs to be proven? Where does it come from?)

For group A, this task should be solved without using the new mock-up feature, for group B using the mock-up feature. The resulting proof state from loading this program into the KeY system and invoking the automatic strategies is a state where the *Body Preserves Invariant Case* is not closeable. This is due to a small off-by-one programming error in line 23 of Listing 5.2, where the loop is executed one iteration too much (highlighted in the listing). This results in an `IndexOutOfBoundsException`.

Program Verification – Specification Mistake Missing Statement About Return Value: Method `isPalin(int[] a)` For group A, this task should be solved with the new feature, for group B without feature.

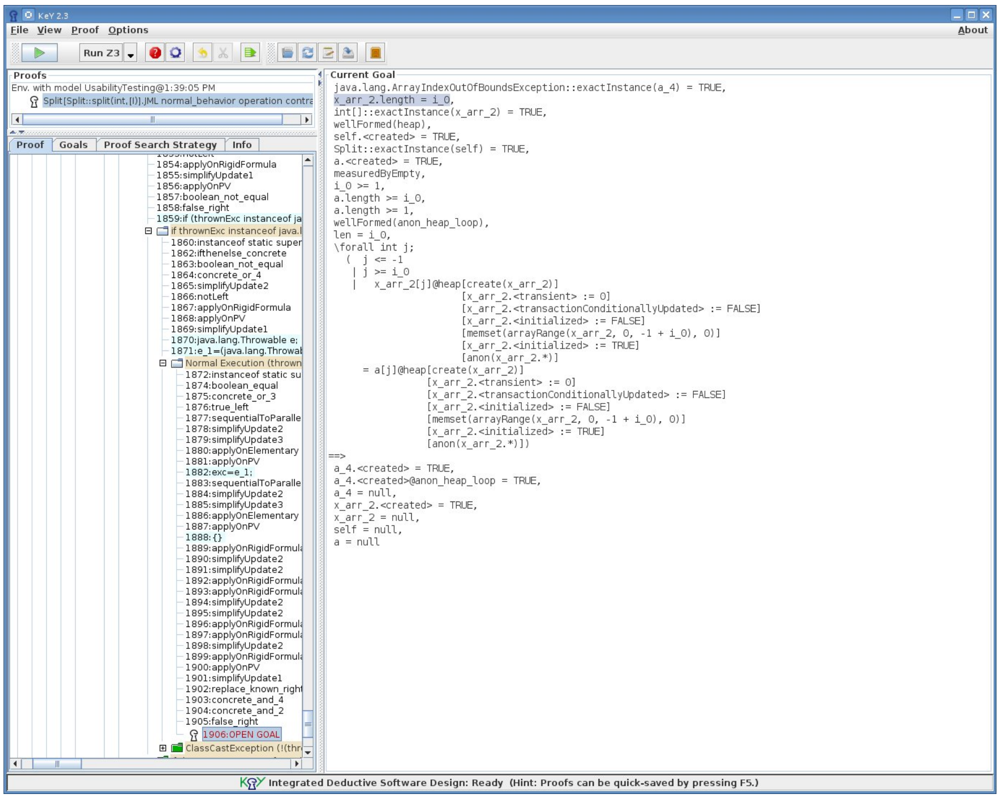
> We have a method that checks whether an array contains an Integer palindrome. A palindrome is a chain of symbols that have the same result when reading it from left to right and vice-versa. For example 1001 is a palindrome, 1010 is not a palindrome.

3.3.1 What has happened? Please describe the proof situation to me. Where are we in the proof? What is displayed on the screen?

3.3.2 How would you proceed?

3.3.3 Please explain to me, why does `[a-1 + i_0 * -1 + a.length] = a[i_0]` needs to be proven? (Where does it come from?)

3.3.4 Please explain to me, why does `x_18 = i_0` needs to be proven? Where does it come from?



Env. with model UsabilityTesting@1:39:05 PM
Split[Split::split(int,[I)].JML normal_behavior operation contra

```
1854:applyOnRigidFormula
1855:simplifyUpdate1
1856:applyOnPV
1857:boolean_not_equal
1858:false_right
1859:if (thrownExc instanceof ja
if thrownExc instanceof java.l
  1860:instanceof static super
  1862:ifthenelse_concrete
  1863:boolean_not_equal
  1864:concrete_or_4
  1865:simplifyUpdate2
  1866:notLeft
  1867:applyOnRigidFormula
  1868:applyOnPV
  1869:simplifyUpdate1
  1870:java.lang.Throwable e;
  1871:e_1=(java.lang.Throwab
  Normal Execution (thrown
    1872:instanceof static su
    1874:boolean_equal
    1875:concrete_or_3
    1876:true_left
    1877:sequentialToParalle
    1878:simplifyUpdate2
    1879:simplifyUpdate3
    1880:applyOnElementary
    1881:applyOnPV
    1882:exc=e_1;
    1883:sequentialToParalle
    1884:simplifyUpdate2
    1885:simplifyUpdate3
    1886:applyOnElementary
    1887:applyOnPV
    1888:{}
    1889:applyOnRigidFormula
    1890:simplifyUpdate2
    1891:simplifyUpdate2
    1892:applyOnRigidFormula
    1893:applyOnRigidFormula
    1894:simplifyUpdate2
    1895:simplifyUpdate2
    1896:applyOnRigidFormula
    1897:applyOnRigidFormula
    1898:simplifyUpdate2
    1899:applyOnRigidFormula
    1900:applyOnPV
    1901:simplifyUpdate1
    1902:replace_known_right
    1903:concrete_and_4
    1904:concrete_and_2
    1905:false_right
    1906:OPEN GOAL
  ClassCastException (!(thr
```

```
 java.lang.ArrayIndexOutOfBoundsException::exactInstance(a_4) = TRUE,
 x_arr_2.length = i_0,
 int[]::exactInstance(x_arr_2) = TRUE,
 wellFormed(heap),
 self.<created> = TRUE,
 Split::exactInstance(self) = TRUE,
 a.<created> = TRUE,
 measuredByEmpty,
 i_0 >= 1,
 a.length >= i_0,
 a.length >= 1,
 wellFormed(anon_heap_loop),
 len = i_0,
 \forall int j;
   (  j <= -1
    | j >= i_0
    |   x_arr_2[j]@heap[create(x_arr_2)]
                       [x_arr_2.<transient> := 0]
                       [x_arr_2.<transactionConditionallyUpdated> := FALSE]
                       [x_arr_2.<initialized> := FALSE]
                       [memset(arrayRange(x_arr_2, 0, -1 + i_0), 0)]
                       [x_arr_2.<initialized> := TRUE]
                       [anon(x_arr_2.*)]
      = a[j]@heap[create(x_arr_2)]
                 [x_arr_2.<transient> := 0]
                 [x_arr_2.<transactionConditionallyUpdated> := FALSE]
                 [x_arr_2.<initialized> := FALSE]
                 [memset(arrayRange(x_arr_2, 0, -1 + i_0), 0)]
                 [x_arr_2.<initialized> := TRUE]
                 [anon(x_arr_2.*)])
==>
 a_4.<created> = TRUE,
 a_4.<created>@anon_heap_loop = TRUE,
 a_4 = null,
 x_arr_2.<created> = TRUE,
 x_arr_2 = null,
 self = null,
 a = null
```

Integrated Deductive Software Design: Ready (Hint: Proofs can be quick-saved by pressing F5.)

Figure 5.3.: Proof state of method `split`

In this example the specification contains two mistakes: in the loop invariant it was forgotten to state that the variable `res` stores whether a sequence of elements was already found that show that there is no palindrome in the integer array. The same mistake is made in the post condition, where there was forgotten to state that the result of the method is true iff the array contains a palindrome. Listing 5.3 shows the annotated program, where the parts of the specification that were missing during the user study are highlighted. Figure 5.4 depicts the state shown to the participants.

FOL Proof States We also provided two tasks that contained a proof state from a pure first-order proof. With these proof states we wanted to explore whether the course of actions for gaining the orientation is significantly different from the course of actions when gaining the orientation in a program verification proof. The difference in first-order proof states is that there is no program to relate proof state to. Both tasks did not contain any function or predicate symbols that would have a clear meaning to the participants, i.e., we used symbols such as “m” or “f” for function symbols.

Listing 5.2: Proof task: split()

```
/*@ public normal_behaviour
  @ requires a!= null && len >= 0 && a.length > len && a.length>=0;
  @ ensures \result != null && \result.length == len;
  @ ensures (\forall int i; 0 <= i && i < len; \result[i]== a[i]);
  @ assignable \nothing;
  @*/
public int[] split(int len, int[] a) {
   int[] res;

   if(len == 0  a.length == 0) {
    res = new int[0];
    return res;
   }

   res = new int[len];

/*@ loop_invariant res.length <= a.length && a.length > 0 &&
  @  len > 0 && 0 <= i && i <= len
  @    && (\forall int j; 0 <= j && j < i; res[j] == a[j]);
  @  assignable  res[*];
  @  decreases len - i;
  @*/
  for(int i = 0; i <= len; i++) {
    res[i] = a[i];
  }
  return res;
}
```

FOL: `m(a, d)` For group A the subtask about the origin should be solved without the new feature, for group B by using feature.

> Our next step is to load the problem `hist2_wo.key.proof`. The play button has been pressed again.
>
> 3.4.1 What has happened? Please describe the proof situation to me. Where are we in the proof? What is displayed on the screen?
>
> 3.4.2 How would you proceed?
>
> 3.4.3 Please explain to me, why does `m(a,d)` needs to be proven? (Where does it come from?)

FOL: `g(1, h8_0)` The second task for a proof state from a first-order logic proof is briefly introduced in the following. For group A this task should be solved without the new feature, for group B with feature.

> Next we load the example Hist1.key.proof. The play button has been pressed again.

Listing 5.3: Proof task: palindrome()

```
public class Palindrome {

/*@ public normal_behaviour
  @ requires a!= null;
  @ ensures \result ==
  @   (\forall int i;  0 <= i && i < a.length; a[i]== a[a.length-i-1]);
  @ assignable \nothing;
  @*/
public /*@ helper @*/  boolean isPalin(int[] a) {
      boolean res = true;

    /*@ loop_invariant a != null && 0 <= i && i <= a.length &&
      @  res == (\forall int x; 0 <= x && x < i; a[x] == a[a.length-x-1]);
      @ assignable \nothing;
      @ decreases a.length - i;
      @*/
      for(int i = 0; i < (a.length/2); i++) {
        if(a[i] != a[a.length-i-1]) {
          res = false;
        }
      }
      return res;
 }
}
```

3.5.1 What has happened? Please describe the proof situation to me. Where are we in the proof? What is displayed on the screen?

3.5.2 How would you proceed?

3.5.3 Please explain to me, why does `g(l, h8_0)` needs to be proven? (Where does it come from?)

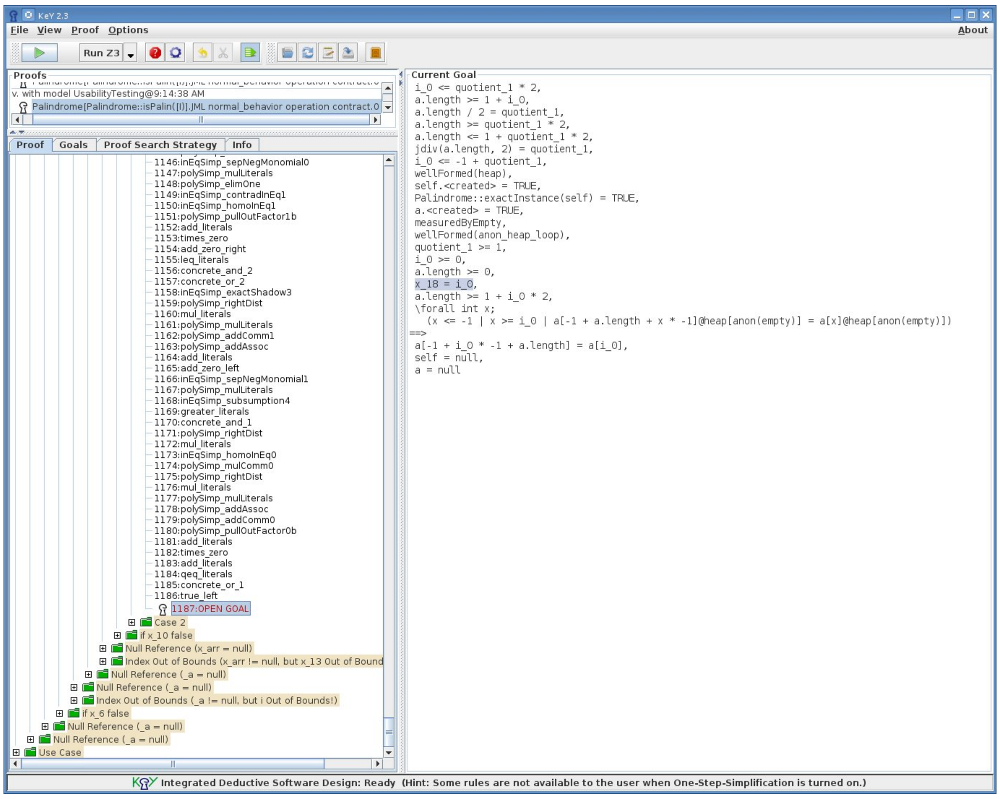
Script: Pre- and Post-Inquiry History Mechanism

After two situations using the KeY system as the participants were used to it, we introduced the new functionality that allows to change the view on the proof state but does not change the state itself. For introduction, we wanted to explore the first impression of the participants when they encounter that there is a new functionality and how it is called.

4.1 There is a new functionality in the KeY system which is called *Show History*. What do you envisage under this?

You are now able to use this functionality for the next tasks.

After performing tasks with this functionality we again explored the opinion of our participants about the new functionality.



```
v. with model UsabilityTesting@9:14:38 AM
Palindrome[Palindrome::isPalin([I)].JML normal_behavior operation contract.0
```

```
1146:inEqSimp_sepNegMonomial0
1147:polySimp_mulLiterals
1148:polySimp_elimOne
1149:inEqSimp_contradInEq1
1150:inEqSimp_homoInEq1
1151:polySimp_pullOutFactor1b
1152:add_literals
1153:times_zero
1154:add_zero_right
1155:leq_literals
1156:concrete_and_2
1157:concrete_or_2
1158:inEqSimp_exactShadow3
1159:polySimp_rightDist
1160:mul_literals
1161:polySimp_mulLiterals
1162:polySimp_addComm1
1163:polySimp_addAssoc
1164:add_literals
1165:add_zero_left
1166:inEqSimp_sepNegMonomial1
1167:polySimp_mulLiterals
1168:inEqSimp_subsumption4
1169:greater_literals
1170:concrete_and_1
1171:polySimp_rightDist
1172:mul_literals
1173:inEqSimp_homoInEq0
1174:polySimp_mulComm0
1175:polySimp_rightDist
1176:mul_literals
1177:polySimp_mulLiterals
1178:polySimp_addAssoc
1179:polySimp_addComm0
1180:polySimp_pullOutFactor0b
1181:add_literals
1182:times_zero
1183:add_literals
1184:qeq_literals
1185:concrete_or_1
1186:true_left
  1187:OPEN GOAL
          Case 2
        if x_10 false
      Null Reference (x_arr = null)
      Index Out of Bounds (x_arr != null, but x_13 Out of Bound
    Null Reference (_a = null)
   Null Reference (_a = null)
   Index Out of Bounds (_a != null, but i Out of Bounds!)
  if x_6 false
 Null Reference (_a = null)
Null Reference (_a = null)
Use Case
```

```
 i_0 <= quotient_1 * 2,
 a.length >= 1 + i_0,
 a.length / 2 = quotient_1,
 a.length >= quotient_1 * 2,
 a.length <= 1 + quotient_1 * 2,
 jdiv(a.length, 2) = quotient_1,
 i_0 <= -1 + quotient_1,
 wellFormed(heap),
 self.<created> = TRUE,
 Palindrome::exactInstance(self) = TRUE,
 a.<created> = TRUE,
 measuredByEmpty,
 wellFormed(anon_heap_loop),
 quotient_1 >= 1,
 i_0 >= 0,
 a.length >= 0,
 x_18 = i_0,
 a.length >= 1 + i_0 * 2,
 \forall int x;
   (x <= -1 | x >= i_0 | a[-1 + a.length + x * -1]@heap[anon(empty)] = a[x]@heap[anon(empty)])
==>
 a[-1 + i_0 * -1 + a.length] = a[i_0],
 self = null,
 a = null
```

```
Integrated Deductive Software Design: Ready  (Hint: Some rules are not available to the user when One-Step-Simplification is turned on.)
```

Figure 5.4.: Proof state of method `isPalin`

We have some questions about the new mechanism:

4.2 How do you like the new mechanism?

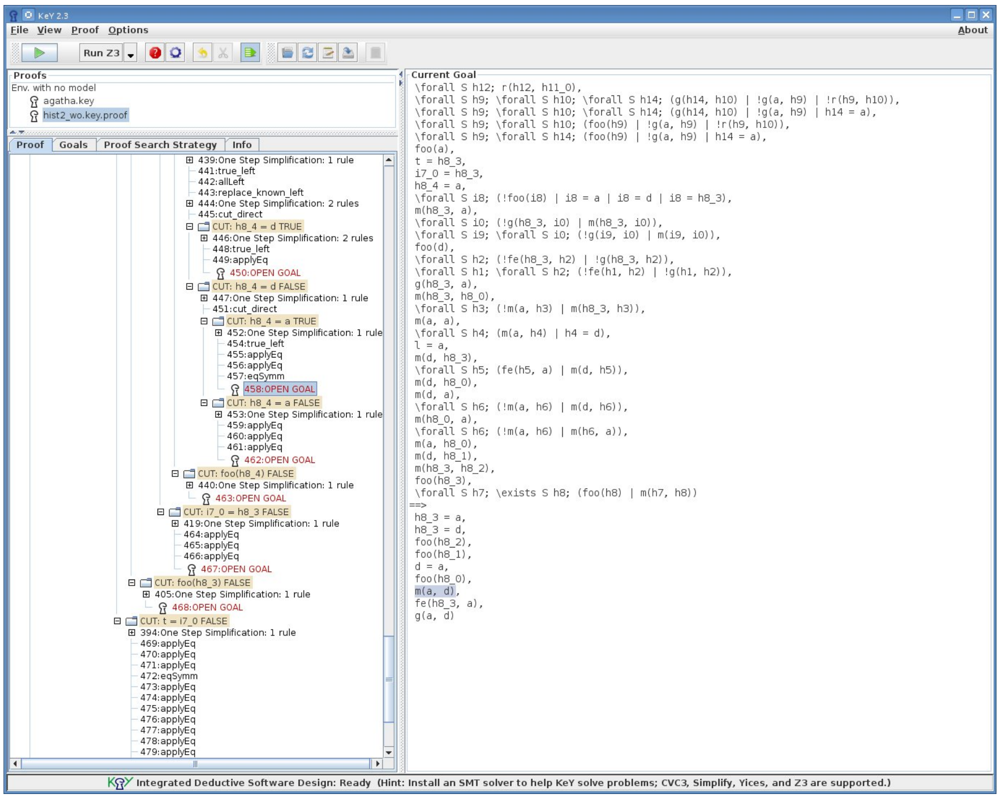
4.3 Did the mechanism show you what you expected? What did you expect? What was different?

4.4 What do you expect if a formula/a term has more than one parent? Up to which depth do you want to see the parents of the parent formulas?

4.5 Do you have suggestions for improving the mechanism?

The key idea behind the pre-enquiry was to explore the first impression about the functionality and to match the propositions to the opinions in the post-enquiry.

This part of the study focuses on the idea to provide different views onto the proof state in order to support the user in comprehending the state without changing the proof state. Our hypothesis is that changing the proof state often requires the user to start the orientation in the proof process again which can be a time-consuming task.



```
439:One Step Simplification: 1 rule
441:true_left
442:allLeft
443:replace_known_left
444:One Step Simplification: 2 rules
445:cut_direct
CUT: h8_4 = d TRUE
  446:One Step Simplification: 2 rules
  448:true_left
  449:applyEq
  450:OPEN GOAL
CUT: h8_4 = d FALSE
  447:One Step Simplification: 1 rule
  451:cut_direct
  CUT: h8_4 = a TRUE
    452:One Step Simplification: 1 rule
    454:true_left
    455:applyEq
    456:applyEq
    457:eqSymm
    458:OPEN GOAL
  CUT: h8_4 = a FALSE
    453:One Step Simplification: 1 rule
    459:applyEq
    460:applyEq
    461:applyEq
    462:OPEN GOAL
CUT: foo(h8_4) FALSE
  440:One Step Simplification: 1 rule
  463:OPEN GOAL
CUT: i7_0 = h8_3 FALSE
  419:One Step Simplification: 1 rule
  464:applyEq
  465:applyEq
  466:applyEq
  467:OPEN GOAL
CUT: foo(h8_3) FALSE
  405:One Step Simplification: 1 rule
  468:OPEN GOAL
CUT: t = i7_0 FALSE
  394:One Step Simplification: 1 rule
  469:applyEq
  470:applyEq
  471:applyEq
  472:eqSymm
  473:applyEq
  474:applyEq
  475:applyEq
  476:applyEq
  477:applyEq
  478:applyEq
  479:applyEq
```

```
 \forall S h12; r(h12, h11_0),
 \forall S h9; \forall S h10; \forall S h14; (g(h14, h10) | !g(a, h9) | !r(h9, h10)),
 \forall S h9; \forall S h10; \forall S h14; (g(h14, h10) | !g(a, h9) | h14 = a),
 \forall S h9; \forall S h10; (foo(h9) | !g(a, h9) | !r(h9, h10)),
 \forall S h9; \forall S h14; (foo(h9) | !g(a, h9) | h14 = a),
 foo(a),
 t = h8_3,
 i7_0 = h8_3,
 h8_4 = a,
 \forall S i8; (!foo(i8) | i8 = a | i8 = d | i8 = h8_3),
 m(h8_3, a),
 \forall S i0; (!g(h8_3, i0) | m(h8_3, i0)),
 \forall S i9; \forall S i0; (!g(i9, i0) | m(i9, i0)),
 foo(d),
 \forall S h2; (!fe(h8_3, h2) | !g(h8_3, h2)),
 \forall S h1; \forall S h2; (!fe(h1, h2) | !g(h1, h2)),
 g(h8_3, a),
 m(h8_3, h8_0),
 \forall S h3; (!m(a, h3) | m(h8_3, h3)),
 m(a, a),
 \forall S h4; (m(a, h4) | h4 = d),
 l = a,
 m(d, h8_3),
 \forall S h5; (fe(h5, a) | m(d, h5)),
 m(d, h8_0),
 m(d, a),
 \forall S h6; (!m(a, h6) | m(d, h6)),
 m(h8_0, a),
 \forall S h6; (!m(a, h6) | m(h6, a)),
 m(a, h8_0),
 m(d, h8_1),
 m(h8_3, h8_2),
 foo(h8_3),
 \forall S h7; \exists S h8; (foo(h8) | m(h7, h8))
==>
 h8_3 = a,
 h8_3 = d,
 foo(h8_2),
 foo(h8_1),
 d = a,
 foo(h8_0),
 m(a, d),
 fe(h8_3, a),
 g(a, d)
```

Integrated Deductive Software Design: Ready (Hint: Install an SMT solver to help KeY solve problems; CVC3, Simplify, Yices, and Z3 are supported.)

Figure 5.5.: State for task `m(a, d)`

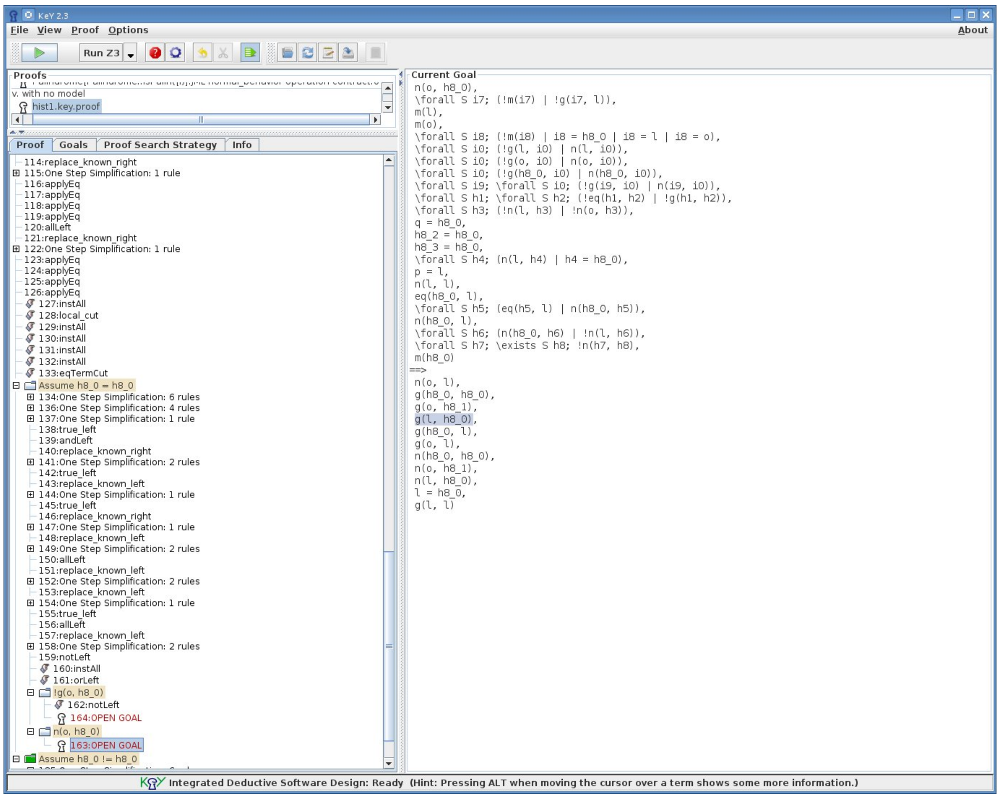
Script: Origin of specific terms and formulas

We wanted to explore the intuitive meaning of the origin of specific terms or formulas of our participants. Do KeY users consider the origin of a formula or term more globally – from the original proof problem formulation or from user input, or do they rather see an origin more locally – that the origin is determined by a single calculus rule application.

For this we chose to show a proof state and ask our participants what they consider to be the parent of a term or formula. We tried to cover formulas that resulted from different kinds of rule applications and different parts of the specification. It was not possible to cover all available combinations as we needed to limit the time for this task to a reasonable amount. As this kind of task requires the participants to scroll through the proof tree and find the parent formulas on the sequent, it may be a tiresome task.

We introduced the task to the participants:

> We would like to carry out a small task block. We will show you a proof state and we will ask questions about this state. It is important that you use the mouse-pointer to show and that you think-aloud.



```
v. with no model
hist1.key.proof

114:replace_known_right
115:One Step Simplification: 1 rule
116:applyEq
117:applyEq
118:applyEq
119:applyEq
120:allLeft
121:replace_known_right
122:One Step Simplification: 1 rule
123:applyEq
124:applyEq
125:applyEq
126:applyEq
127:instAll
128:local_cut
129:instAll
130:instAll
131:instAll
132:instAll
133:eqTermCut
Assume h8_0 = h8_0
  134:One Step Simplification: 6 rules
  136:One Step Simplification: 4 rules
  137:One Step Simplification: 1 rule
  138:true_left
  139:andLeft
  140:replace_known_right
  141:One Step Simplification: 2 rules
  142:true_left
  143:replace_known_left
  144:One Step Simplification: 1 rule
  145:true_left
  146:replace_known_right
  147:One Step Simplification: 1 rule
  148:replace_known_left
  149:One Step Simplification: 2 rules
  150:allLeft
  151:replace_known_left
  152:One Step Simplification: 2 rules
  153:replace_known_left
  154:One Step Simplification: 1 rule
  155:true_left
  156:allLeft
  157:replace_known_left
  158:One Step Simplification: 2 rules
  159:notLeft
  160:instAll
  161:orLeft
  !g(o, h8_0)
    162:notLeft
    164:OPEN GOAL
  n(o, h8_0)
    163:OPEN GOAL
Assume h8_0 != h8_0
```

```
 n(o, h8_0),
 \forall S i7; (!m(i7) | !g(i7, l)),
 m(l),
 m(o),
 \forall S i8; (!m(i8) | i8 = h8_0 | i8 = l | i8 = o),
 \forall S i0; (!g(l, i0) | n(l, i0)),
 \forall S i0; (!g(o, i0) | n(o, i0)),
 \forall S i0; (!g(h8_0, i0) | n(h8_0, i0)),
 \forall S i9; \forall S i0; (!g(i9, i0) | n(i9, i0)),
 \forall S h1; \forall S h2; (!eq(h1, h2) | !g(h1, h2)),
 \forall S h3; (!n(l, h3) | !n(o, h3)),
 q = h8_0,
 h8_2 = h8_0,
 h8_3 = h8_0,
 \forall S h4; (n(l, h4) | h4 = h8_0),
 p = l,
 n(l, l),
 eq(h8_0, l),
 \forall S h5; (eq(h5, l) | n(h8_0, h5)),
 n(h8_0, l),
 \forall S h6; (n(h8_0, h6) | !n(l, h6)),
 \forall S h7; \exists S h8; !n(h7, h8),
 m(h8_0)
==>
 n(o, l),
 g(h8_0, h8_0),
 g(o, h8_1),
 g(l, h8_0),
 g(h8_0, l),
 g(o, l),
 n(h8_0, h8_0),
 n(o, h8_1),
 n(l, h8_0),
 l = h8_0,
 g(l, l)
```

Integrated Deductive Software Design: Ready (Hint: Pressing ALT when moving the cursor over a term shows some more information.)

Figure 5.6.: State for task `g(l, h8_0)`

For each term or formula that we have shown we asked the same questions:

5.1 Please show me, starting from proof node number `NodeNumber` the parent of the formula `SpecificFormula`.

5.2 Why would you see this formula as parent formula?

For all formulas we used the `sumAndMax` proof from the examples shipped with KeY. We covered the following types of formulas/origin:

1. formula resulting from a rewrite rule and implicit assumption in the specification (Proof node 14: Formula `a = null`)
2. formula resulting from simplifying a subformula of the loop invariant (Proof node 116: Formula `i_0 <= a.length`)
3. formula resulting from the loop condition in the use case branch (Proof node 899: Formula `a.length >= 1 + k_0`)

4. formula resulting from the frame condition
   (Proof node 987: Formula `f_0_0 = o_0.f_0`)

Script: Cool-Down Tasks

In the cool-down part of our study we asked more general questions about the *Show History* functionality that the participants have seen during the study and about improvements of the KeY system.

> Thank you that you have taken part in our study. To the end we have some more general questions.

6.1 Can you imagine for which tasks in the general proof context *Show History* could be used for?

6.2 Would you use this functionality? Is so, do you have a small example for the usage?

6.3 Do you have other suggestions for improving the KeY system?

6.4 Would it help, that terms are marked with the information from which part of the specification they originated from?

This cool-down questioning should give the participants the possibility to mention topics they have missed mentioning during the study.

5.5. Running the User Study

We conducted our user study in 2014 with nine participants that used the KeY system. The setup of our study including a more detailed description of our participants is described in the following.

5.5.1. Moderator

A moderator which was not part of the test-design team and not an expert in software verification posed the questions during the test sessions. As with the focus group study, also in this study design the moderator must not be one of the stake holders and must be neutral in his or her opinion about the evaluated software. The moderator was a computer scientist who has seen the KeY system in action beforehand, but was not an experienced user of the system. She was not an expert moderator, so an extensive briefing prior to the study was necessary. Notes have been taken by the leader of the study team. The study leader was not allowed to pose questions by herself but was able to instruct the moderator to ask further questions if needed.

5.5.2. Technical Setup

We performed this user study using a computer with a special version of the KeY system and an open-source screencast software (SimpleScreenRecorder) installed. An external microphone attached to the computer was used to record the voices for the screencast recording. An additional recording device was used to record the voices without connection to the screencast software for back up purposes and in case the primary technical setup fails.

5.5.3. Recordings and Transcription

We have recorded all test sessions using the screencast software and a microphone. To be on the safe side in case technical issues arise we also have recorded the voices of the sessions with a second voice recording device. In total during the study around 13 hours of video respectively audio recorded material has been collected.

All sessions have been manually transcribed and names have been anonymized using pseudonyms. Each participant was assigned a number and either the character “A” or “B”. the characters encode which order of tasks the participants were given during the study. However, from the introduction part of the study it may still possible to identify individual participants, therefore we chose to only present a list of answers or ranges for the knowledge of the system and not the link to the individual participants. Especially we abstracted the time span to three categories

The voice recordings were transcribed using very basic transcription rules (adapted from [Kuc14]). In the following, we summarized the main rules we followed:

1. If the participant has a longer break it is indicated by a note in brackets. Shorter breaks are indicated by “...”. The exact duration of the break was not transcribed.
2. The language is polished to a certain extent, such that dialects are not visible anymore.
3. The language and punctuation is approximated towards written language and minor grammatical errors are corrected during transcription.
4. Speaker changes are indicated by starting the paragraph with the alias for the participant or the moderator.
5. Vocalizations such as laughing or groaning are transcribed using parentheses, e.g., “(laughing)”
6. Incomprehensible words or phrases are indicated using “(unv.)”.
7. All references that allow to determine a certain identity are anonymized during transcription.
8. Disturbances or non-verbal actions (e.g., on the video) are noted in brackets.

During the test session of one participant (TN2B) the screencast software was accidentally turned off after approx. 30 minutes of recording. For this participant we were therefore only able to use the 30 minutes of video-taped session and for the rest of the test session we needed to use the voice recordings. This resulted in the decision to only use the responses to the interview questions after the first practical task and to omit the rest of the transcribed material.

5.5.4. Participants

We conducted our experiment with KeY users and KeY developers. As the KeY system is developed in academia it was difficult to get users of the system that were not also developers of the system in any way. For the introductory part of the sessions we included the questions about the development tasks of the participants in case they also develop the KeY system.

Nine KeY users have agreed to take part in the experiment, from which six participants rated their experience level with the KeY system as being intermediate[4]. Three participants rated themselves as being expert users. We have summarized the distribution of the participants in Table 5.7.

Moreover, we asked about the time the participants know the KeY system and about the hours they spend per week in using the KeY system. The results of the participants did not support the categorization of the participants into different experience levels: On the one hand, e.g., intermittent, extensive usage of the system is not accurately reflected by the average usage time per week alone. On the other hand the difference between merely knowing the system compared to actually using it is not captured precisely by our questions.

Our observations have been that the self-rating together with a description why the participants rate themselves into the respective experience level revealed the best information to be able to categorize the experience levels.

5.6. Results of the User Study – Proof Process

For the data analysis of the study we first summarized the responses of our participants and assigned labels to the transcribed material that captured the content of the responses. Furthermore, if general statements about problems or room for improvement have been made, we labeled them accordingly.

We then analyzed the transcribed and labeled data in two ways: the interview questions already provided top-level categories; the responses to the different questions were inductively coded by building the (sub)-categories from the participant's responses and the labels. For the interactive interview we extracted sequence models

[4]One participant rated him/herself as being somewhere between novice and intermediate. We counted this participant to the group of intermediate leveled-users. Another participant rated himself as being somewhere between expert and intermediate-leveled. This participant was also counted as being intermediate-leveled, because the description about why this categorization was chosen revealed that the participant stated that some parts of the system are not known fully to him.

Participant	Time knowing KeY (yr)	Self-rating	Our Rating
TN1A	≥ 7	expert	expert
TN2A*	≥ 3 and < 7	novice–intermediate	intermediate
TN3A	≥ 7	expert	expert
TN4A	≥ 3 and < 7	intermediate – expert	intermediate
TN1B	≥ 7	intermediate – expert	intermediate
TN2B	≥ 3 and < 7	intermediate	intermediate
TN3B	≥ 7	expert	expert
TN4B	≥ 7	intermediate	intermediate
TN5B*	≥ 7	intermediate	intermediate

Figure 5.7.: Participants of our user study. Participants marked with an asterisk were not developers of the system.

(see Section 2.5.1) from the screencasts and added notes to these models. If statements were made that may also be relevant for other parts of the study, we coded those statements and assigned them to the respective categories.

In the following, we will present the detailed analysis results of the different parts of the study. Although the user study was conducted in German, we performed the coding and analysis directly in English. If it suited the explanation, we quoted parts of the voices of participants. In this case we have translated the voices to the best of our knowledge.

The results of the user study divided into the different topics for which we wanted to gain insight into. The section is structured as follows: We briefly introduce the topics and recapitulate which parts of the study were intended to provide insights into the topics. We follow with the presentation of our analysis and for each topic conclude with a discussion of the data.

5.6.1. The Proof Process

To cover different situations in the proof process, this study included tasks that required the participants to use the KeY system. To be more precise we covered the beginning of the process with task `removeElem()` and with tasks `palindrome()`, respectively `split()` we aimed to cover the situation where users need to gain the orientation in the proof process after the application of automatic strategies. In these tasks we focused on the actions the participants performed, their goals, their expectations about the proof situation and the information they need to perform the actions and assess the situation as well as the way this information was obtained. In addition to the aforementioned tasks, also the questions from the warm-up phase from the interview covered the proof process concerning the reasons for an open proof in KeY, the approaches to find out what has happened in the proof and to find the next promising step and the used functionalities provided by KeY for this task (Questions 2.1, 2.2 and 2.3 on page 96).

5.6.2. Expectations if a Proof Attempt in KeY is Unfinished

Central for the interview question about the expectations if a proof attempt is unfinished, is the *interaction point* a user reaches in the proof process (according to the model we have depicted for the hypothesis of the focus group discussions in Sect. 4.1). At such interaction points the user needs to make a decision on how to proceed.

In the verification process, interaction points are at the beginning of the proof process, where the user provides the annotated program to the verification system and at points where the automatic proof search strategies of the system stops and provides an unfinished proof to the user. To gain information about the context where users have to decide which activities to perform we chose to use the interaction point after the application of automatic strategies. We asked about the participants' expectations in case a proof attempt is unfinished and further application of automatic strategies does not change the proof state. We did give some more information about the proof state, i.e., that the sequents are large and the proof contains many open branches.

We have to note beforehand that this information also misled some participants, such that not only answers to the general expectations where given but also answers to the reasons for large sequents and splits in the proof. We did not use these answers for exploring what users expect when they encounter an unfinished proof attempt.

We categorized the answers of the participants into two main categories: answers that were related to the *Semantics of the Formalization* and answers that were related to the category *Calculus and Tool*. The category *Calculus and Tool* was further divided into the sub-categories *calculus* and *automatic proof search and implementation*.

Some answers were not detailed enough or did not fit in one of the two categories, such that we formed a third category *unspecific* to categorize the remaining answers.

Semantics of the Formalization The category semantics of the formalization includes all answers related to the formal specification and the program. Participants gave as reasons for an unfinished proof attempt that the program or the specification contains defects such that the program does not meet its specification and thus the problem is not provable. One particular example is that the loop invariant is either not provided at all or contains a defect. Also, a reason mentioned by one participant is that a counterexample exists. This reason is also concerned with the semantics of the formalization, as a counterexample exists if the current proof state is not provable, e.g., because of contradiction in the propositions in the proof state.

In detail the participants named the following reasons for an unfinished proof attempt that are concerned with the semantics of the formalization:

- the specification contains a defect
- the program contains a defect
- the loop invariant is missing or contains mistakes
- the proof problem is not provable at all
- a counterexample exists.

Calculus and Tool This category contains all subcategories that have to do with the syntactic level of the proof problem, so the calculus that is used to perform the proof by syntactically transforming a proof obligation, as well as the automatic proof search strategies and their implementation. The participants mentioned expectations that relate to the incompleteness of the calculus itself, e.g., that rules for a specific proof problem may be missing in the system, and therefore the proof can not progress, or that the automatic application of a rule is disabled, and needs to be applied interactively.

Also expectations about the concrete implementation of the automatic proof search have been mentioned. Here we can distinguish between statements concerning the user-defined parametrization of the automatic proof search and the implementation of the proof search strategies that can not be controlled by the user.

Regarding the user controlled parameters, reasons for an unfinished proof attempt were that the automatic proof search did not have enough resources, i.e., number of proof steps, to find a proof. Also that rule sets are disabled or the proof splitting was turned off was mentioned as reasons.

Reasons for an unfinished proof attempt that have their origin in the *implementation of the proof search strategies* included that the proof search strategies used an adverse prioritization of certain rules (e.g., used splitting rules too early in the proof), such that the proof cannot not be found or structures necessary for the application of constructive rules have not arisen from the formulas on the sequent. Also possible loops in rule applications were mentioned as reasons that KeY was not able to find a proof. More specifically, it was mentioned that quantifier instantiations could not be found or that the automatic strategies "went too far".

Reasons that KeY was not able to find a proof that were also mentioned also included the problem formulation, e.g., that the formulation of the proof problem is too complex (i.e, the program and the specification) or that just the specification was formulated too complicated for the automatic strategies to find a proof. Also that the proof problem itself is too complex for KeY or the automatic strategies to find a proof was mentioned as reason. One participant mentioned in his answer that deciding which of the two possibilities (syntactic or semantic) occurred is not easy.

Unspecific Some answers given by the participants were not detailed enough or unspecific in the form that they may fit in more than one category depending on their interpretation. So we have decided to categorize those answers in one extra category. Answers were for example that the reasons for an open proof are "dependent on the problem" or that a "proof could not be found". Depending on the interpretation of these answers they may have been categorized into either calculus and tool or semantics of the formalization, e.g., dependent on the problem there may be either a defect in the specification or the program or the proof system needs user guidance in some form.

Discussion The question we have posed to gain insight into the expectations of users of the KeY system if they encounter an unfinished proof attempt was:

> What could have happened? What could be the reasons that KeY opened a lot of proof branches and was not able to close them?

The information we included about the proof state (many branches and large sequents) in this question may have drawn the attention of some participants to this insignificant detail, such that in some cases answers contained concrete references to the shape of the proof state. One participant even interpreted this information to conclude that all goals stay open and said this situation is unusual. This however, showed that the shape of an unfinished proof may also give users hints about the reasons and may raise expectations about what happened in the proof.

Although the wording of the question may have lead participants to concentrate on an immaterial detail, we were able to gain insight into expectations our participants have when encountering an unfinished proof attempt. We have learned that the participants consider reasons for unfinished proof attempts on different "levels": the proof input artifacts as well as the proof calculus, the automatic strategies with their settings and the implementation of the calculus. Users should be supported in the task of finding out on which level to interact with the system in case they encounter an unfinished proof attempt. Users are not able to interact on all of these levels, e.g., the implementation of the strategies cannot be easily changed by the user, but only adjusted by the strategy settings. We were also able to encounter that the settings of strategies that were considered, included implementation decisions, such as that splitting rules were used too early, as well as user-controlled settings, such as the settings of the maximal resources the prover is allowed to use.

One expectation about the underlying strategies why the proof attempt may be unfinished was that specific rules were not applied automatically, because they were disabled for the strategies. In such a case it may be a supporting feature for users to make this information explicit, i.e., that the proof system informs users about such applicable rules for open goals.

An expected reason for the automatic strategies to stop is that the user-defined limit for resources is reached. In KeY the resource limit is expressed in terms of number of rule applications. We assume this limit may be supportive if users want to control the proof search and prevent the prover from constructing complex or incomprehensible proof states, which the user may want to inspect afterward. However, we assume that users may have the intent to let the prover try to prove the proof problem without wanting to inspect the proof state afterward. In such a case another resource limit may be advisable: the usage of a time limit that specifies the time span users are willing to wait until they try to prove the problem interactively. Such time limits are often present in proof systems where the user is only interested in the answer of a prover, such as SMT or SAT solvers.

Concerning decisions made in the strategies, a participant mentioned that possibly splitting rules have been applied too early in the proof. Users of KeY can control the application of splitting rules, however, maybe more fine-grained control options for splitting could be supportive.

It was furthermore expressed that the (formalization of the) proof problem may be "too complex" for the prover or that the strategies "went too far" as expectations. These statements were quite general and it has to be further investigated what users considered with these statements.

Assuming that the users' intention for the statement "strategies went too far" is that the produced proof state is too complex for the user to trace and comprehend, a further limit for the strategy application may be a user-defined complexity metric for the proof state. For instance, the strategies should stop if a certain the number of (open) proof branches has been created or the size of the sequents of the open goals exceeded a user-defined limit.

For the complexity of the proof problem itself, the program as well as the specification have been considered as being too complex. For both cases it could also be a support for the user to get feedback on how complex the proof input artifacts are and which complexity can still be handled by the prover. In software engineering, already complexity metrics for software exist, such as the cyclomatic complexity. For specifications, it could also be the case that the property itself is not too complex, but the formulation by the user. For such a case it could be a user support to provide a set of common specification patterns to avoid adversely formulated specifications.

5.6.3. Approaches to Proceed in the Verification Process

As a follow-up question we wanted to know the strategies and the features of KeY the participants use to discover the reason for an unfinished proof attempt. We asked how the participants approach the problem of finding out what has happened and we also asked which possibilities they have for this task. The answers given by the participants ranged from very high-level descriptions and intents down to single rule applications. If possible we have derived sequence models of the descriptions.

In the following we will summarize the different approaches of our participants for the different proof situations and, if named, also the features offered by the KeY system they use. This includes answers to Questions 2.2 and 2.3 of the warm-up phase on page 96. When analyzing the collected data, we sketched sequence models for each participant. After developing the sequence models we identified commonalities and differences in the approaches.

We were able to identify different activities the users mention they would perform for finding out the next step and to proceed in the proof process. The order in which these activities are performed and the triggers for the activities differ between the participants. It was visible that each participant has his own strategy for the posed task, however, the activities they perform are comparable.

Place proof in a defined situation. To place the proof in a defined situation the participants mentioned to restart the whole proof or prune the proof tree from a point from which they have an idea about the situation, e.g., after the symbolic execution has ended or at the root of the proof tree.

One participant mentioned he prunes the proof when the automatic strategies "went too far", another participant mentioned he prunes when he has "lost the overview". We also assigned the use of the macro `Propositional Expansion with Splits` to this activity, as we consider the result of this macro to be a defined situation where users know the general shape of the sequent and proof. A goal for this activity mentioned by a participant was to “gain clarity about the situation” respectively “localize what is necessary in the respective situation”.

Controlled proof construction. One participant mentioned explicitly that he constructs the proof “in a controlled way”. We used this description for the manual activities used to construct a proof. According to our observations, manual proof construction is done using macros, manual rule applications or short sequences of automatic proof steps. We also added the simplification of formulas using macros to this activity, as well as the manual rule applications of the cut- and instantiate-rule. We considered the usage of the macro `Close Provable Goals Below` as controlled proof construction, as this macro leaves the proof goal untouched if not closing and therefore does not require that the user needs to find the orientation in the proof after the application of the macro.

Continue automatic proof search Many participants would continue the automatic proof search, often after increasing the number of proof steps. One intent mentioned by a participant was to see what KeY was able to prove. We assume another intent for this activity: to omit the cause of not applying enough proof rules. We also assigned actions to this activity where participants stated to try to close open goals. This can be done in KeY either by using macros (e.g., the macro `Close Provable Goals Below`) or the general automatic strategies. As it was not always mentioned whether the macro `Close Provable Goals Below` would be used (which would fall into the activity *controlled proof construction*) for this purpose, we chose the more general strategy, which requires users to gain orientation after application.

Isolation/deliberate selection of an open goal/proof branch Some participants explained that they select proof branches or open goals according to different selection criteria, amongst others: select an arbitrary or the first open goal, select a proof goal or proof obligation from which the participant knows what to prove or what is likely to go wrong and find specific exceptions on the proof branch and select this goal because it is easier to deal with such cases.

Inspection of the proof state An activity present in many sequence models was to inspect the proof state. We divided this activity into different parts: (a) the preparation for inspection, (b) the inspection of the artifacts, (c) the reconsideration of the mathematical argumentation of the (abstract) proof problem. As preparation for inspection (a) it was mentioned that the view option `Hide Intermediate Proof Steps` for the proof tree was used to hide nodes in the proof tree. Another possibility mentioned was to use model search for integer problems and obtain an assignment for the input parameters of the program, which can be used for task (b). The different artifacts the participants mentioned to inspect are the proof branches and the branching labels, the paths to the open goals, the open goals, the formulas on the sequents and the different proof input artifacts.

When inspecting the proof artifacts it was mentioned that the abstract mathematical argumentation is reconsidered (c). While not explicitly expressed by the participants, we assume that the reconsideration happens mentally.

The intents of the activities in this category have been to gain an overview over the proof or localize/understand what has to be proven (respectively what could not be proven) and to find errors in the formalization. Also to gain orientation over the proof relative to the program, respectively find the relation between the proof branch and the execution path of the program was one intent. One participant inspects the proof tree also by stepping through the tree and matching the steps with his own expectation. Furthermore, when navigating through the proof tree it was mentioned that it is inspected when and how different calculus rules were applied.

Modify input According to what the participants have seen during the activities performed before, some participants mentioned that they may modify the proof input artifacts. Concretely, it was mentioned to adjust or provide the (auxiliary) specification. The intents for modification result from the assessment of the proof artifacts. Intents that were explicitly mentioned have been to correct an error which was observed on the logical representation of the proof obligation in the proof or the specification.

Generate a counterexample Two participants mentioned to generate a counterexample to determine the next step. The counterexample generator can generate a counterexample if one exists and a predefined timeout was not reached yet for the current proof goal. A counterexample is a hint for the user that the proof problem is not provable.

Discussion

In the retrospection participants mentioned activities they perform to successfully proceed the verification task. We were able to identify details to seven activities. In the following we will discuss possibilities to improve KeY for these activities based on the details mentioned by the participants.

Participants mentioned to prune the proof to place the current proof into a defined situation. Pruning a proof is a destructive operation on the proof state that may be a costly action. Preceding a pruning steps we assume that, proof construction steps as well as a step to find the goal node from where the proof should pruned had been performed. We assume if users are willing to destroy a proof state to place the proof in a defined situation, the comprehension of a proof state instead of pruning it may even require more effort for users. To improve KeY such that users do not need to use destructive actions when trying to orientate themselves in the proof, one possibility is to investigate how to improve the presentation of the proof state. Instead of improving the presentation of the proof state also mechanisms may be developed that support users in orientating themselves, for example by supporting effective and efficient retracing of proofs. Further targets for investigation may be the rules or operations that lead to incomprehensible proof states. It needs to be further investigated why users lose the overview in the proof, e.g., between the end of the symbolic execution and the open proof goals. For manual proof construction, participants mentioned to construct (parts of) a proof manually, in a controlled way by for example using macros and then manual proof rules. One way of contributing to this course of actions would be to further develop proof macros or proof search strategies that performs proof steps that have a clear intuitive meaning for the user, e.g., a strategy that only applies arithmetic normalization of selected terms, similar to the macro that simplifies heap structures. To develop such macros it needs to be investigated which more coarse-grained proof steps users need. Furthermore, maybe also a strategy could be implemented that stops after a certain number of proof steps (or after the current proof state differs from the one the user has seen last according to a distance metric). By having such a proof strategy the user may inspect the intermediate proof state shown after the strategy stopped to decide whether to continue the proof search strategy or adjust the proof strategy before continuing the proof search. This corresponds to a certain extent to a "step-wise application of proof strategies".

Proof construction in KeY is in many cases an alternation between automatic and manual proof construction. Participants mentioned to continue in the proof process with the automatic proof search often after the modification of strategy settings, e.g., after the modification of the maximal number of proof steps. Instead of letting the user apply the proof search, one possibility for improvement would be to make use of modern multi-core CPUs: after the application of a proof strategy, when the user inspects the current proof state, the verification system tries to close the currently open proof branch in the background. Important is that the proof state presented to user is not changed, such that the user is not disturbed while inspecting the state. This functionality could be achieved by constructing a new side proof in the background and the user only retrieves an indication if a branch could be closed by this technique (e.g., by changing the icon of a node in the proof tree or adding a notification bar at the top of the current sequent). The background computation of a proof can also be accompanied by trying to generate counterexamples in the background, not only after the termination of the automatic strategies, but even while the proof is progressing.

To be able to apply proof rules or strategies some participants select specific branches or goals. To find the desired node in the proof tree users have to navigate in the proof tree and search for the node. To add further support for the selection of proof branches or nodes one possibility to add additional information to the list of open goals that is available for fast access to open goals. One example is to add path information to the list of open goals. Further support could be provided by allowing users to filter the list according to selection criteria or predefined properties. To enable such support it needs to be investigated further how users select promising goals. Gaining information on the users selection criteria would allow KeY to preselect goals for the user after the termination of automatic strategies instead of selecting the last goal the strategy was working on before termination.

To decide how to proceed in the proof process, users inspect the proof state of unfinished proof attempts. The answers of participants showed that many sources of information are needed (e.g., the proof tree with its labels and branching nodes or the annotated program). To support the user in the inspection task it needs to be investigated how to present the right amount of information and how to present relation between the different information sources appropriately to the user for an effective and efficient proof state inspection task. One example for an improvement of the information presentation can be found in chapters 8 and 10, where the contents of the proof state are shown in different views and users can shift their focus from one presentation to another without losing the relation between the contents of the views. In KeY, support for proof state inspection is present for example in the form of counterexamples or test case generation, where users can generate a test case for the program under verification and trace the program with the test case using a software debugging system. Users can also step through the proof tree to mentally retrace the proof steps. This stepping may be accompanied by a possibility to step over proof parts that the user considers as uninteresting (e.g., normalization of arithmetic expressions), to be able to speed up the tracing process.

During the proof process users need to modify the proof input artifacts according to their knowledge gained from inspecting and advancing the proof. To modify the proof input artifacts users need to switch to the annotated program, perform the modification and redo the proof. We assume that users are not always completely certain whether their modifications would close the proof goal. To ease this activity and avoid switching the artifacts one possibility would be to let users change parts of the proof state within KeY and determine whether the changes would close the goal. In Chapter 9 we have presented a concept for proof exploration in KeY that aims to reduce the number of unfinished proof attempts by allowing users to change the proof state and proceed with modified state instead of performing the changes directly on the proof input artifacts and restart the proof attempts. If the changes led to a closed goal, we show one way of supporting users in transferring the modifications to the proof input artifacts.

5.6.4. Improvements for User Support in the Proof Process

We explicitly asked for alternatives used or missing mechanisms in the KeY system for finding out what has happened and to determine which next steps seem promising. Furthermore, we also asked for general room for improvement in the KeY system in Question 6.3 of the cool-down phase on page 107. Some answers of the general room for improvement were related to or matched the answers given to the more detailed question about the support in the proof process. We have included these answers here as well. We have added the aspect *general room for improvement*, if the answers did not fit any aspect found for the more detailed question. The participants mentioned room for improvement for the KeY system in the following aspects.

Traceability Almost all participants mentioned that they miss the relation between components of the proof state presented in KeY and the proof input artifacts.

The answers given by the participants were of different granularity and concerned different parts of the proof state of KeY and the proof input artifacts. Besides the relation between the proof state of KeY and the proof input artifacts, also the relation among the contents of the different proof nodes as well as elements within a single proof state was mentioned to be insufficiently supported. In Table 5.1 we have summarized the relations between the different components of the proof state mentioned by the participants.

Components of the proof state in KeY		Proof input artifacts
Proof state	$\rightarrow$	Annotated program
Proof state	$\rightarrow$	Source code location
Proof state	$\rightarrow$	Path through program
Sequent	$\rightarrow$	JML annotation
Formulas	$\rightarrow$	Origin in specification
–		Condition that is not provable underlined in the program
Origin of terms in the proof process		–
Different heaps and their contract applications		–
Origin of renamed variables		–
Different views (levels of abstraction) from original form of formulas to representation as it is shown in KeY		–

Table 5.1.: Relations between the different artifacts as mentioned by the participants. The arrow indicates source-target relations. If an entry is missing on one side, the relation is within the artifact.

Furthermore, some participants suggested ideas for improving KeY in the tracing aspect. One idea for improvement was to provide different views (levels of abstraction) from the original form of formulas (e.g., as given in JML) to the representation as it is shown in the sequents in KeY. Another idea that was mentioned is to underline the conditions that could not be proven in the proof input artifacts. This feature is already present in autoactive systems, such as Dafny.

A further suggestion for improvement mentioned by a participant was to develop a tracing functionality dual to *Show History* which allows to select a term and trace how this term develops in the course of the proof process.

To improve the tracing aspect, it was mentioned to allow for a rough estimation of the origin of formulas by indicating this origin information in the branching labels. We assume, from the context of this answer, that it is an improvement for the *Show History* functionality.

One participant summarized as answer to the more general question that the presentation of the requirement/auxiliary specification on the logical level in the KeY system is too different from the representation in the proof input artifacts the user has provided and that these representations need to be brought closer together.

Graphical Representation of the Proof State in KeY For the graphical representation of the proof state in KeY room for improvement was mentioned concerning the *proof tree* and the *sequent*.

The view resulting from the view option `Hide Intermediate Proof Steps` for the presentation of the *proof tree* was one aspect to have room for improvement. In addition to the current representation in the view where the branching nodes are presented to the user, the participant also wants to see the node before the branching node, to not have too large differences between the sequents when navigating through the proof tree.

For the *sequent*, different aspects were mentioned to be missing, which included possibilities to structure the contents of the sequent by means for abstraction (e.g., folding/unfolding of subterms), as well as visualization aspects, such as syntax highlighting on the sequent or an improvement in the visualization of heap structures. Some participants pointed out that formulas containing heap structures are often rather lengthy.

A further suggestion for improving the sequent presentation was to allow for different views onto the sequent that only show comprehensible formulas to the user, as one participant mentioned that the sequent contains a lot of unnecessary symbols and representation of specific formulas is considered to be difficult to understand.

Proof Search To minimize the number of proof goals a user has to inspect it was mentioned that the automatic proof search should be improved to be more powerful and therefore to be able to prove more proof problems automatically. Furthermore, the idea to try out different proof strategies in parallel on the proof problem and to discover which strategy performed best was expressed by one participant.

Furthermore, suggestions for improving the proof search included to automatically perform brute-force quantifier instantiations and let the system perform an educated guess about which formula needs to be true in order to close the open goal.

Interaction in the Proof Process Support for the interaction in the proof process was also mentioned to be worthy of improvement. Specifically, support to reproduce a proof situation after a change in the proof input artifacts was mentioned. Furthermore, two participants wished for another way of interaction, namely a textual language for interacting with the proof system.

To minimize similar interactions a participant mentioned it would help to be able to reuse special cuts as lemmas in the proof.

A more general suggestion for improvement was to allow for more coarse blocks for proof control, as it is currently the case.

Other Issues In our study also other more specific issues needing improvement were mentioned, such as an improved loading and saving functionality for information-flow proofs, as well as an improvement of the counterexample generator. Also the reloading of proofs after a version change of KeY is considered to be room for improvement.

One participant also mentioned the idea to provide another documentation of the proof rules in the KeY system. He wishes for a description of the rationale of a proof rule in natural language. The currently given information in the taclet's pop-up view is considered to be difficult to understand.

General Room For Improvement More general aspects mentioned have been that KeY should be improved in the sense that more Java features are supported and that KeY should be used for more "real-world" case studies. Concerning the case studies the participant identified the requirement that more Java APIs should be specified and shipped with KeY.

Discussion

We specifically asked for general room for improvement, as well as collected suggestions for improvement mentioned throughout our user study. The fact that almost all participants were missing the relation between the proof state and proof input artifacts has led to the development of the user interface design presented in Chapter 10. Also, our mocked-up mechanism which we presented in the user study addresses the issue of relating terms or formulas in a sequent to their ancestors as introduced in Section 5.2.2. The answers of the participants further allowed to identify improvements for future versions of the show history mechanism, e.g., to include origin information in the branching labels of the proof tree or to add the relation to the elements of the proof input artifacts. One option to facilitate the user in relating the terms and formulas on the sequent to the annotated program would be to color code the terms on the sequent according to their origin in the proof input artifact (providing different colors for pre- or postcondition, as well as various auxiliary annotations).

One reoccurring issue during the user study was the proof tree representation and navigation. One difficulty was the amount of information shown to the user: an indicator for this issue was that users often used the feature to hide intermediate proof steps during proof navigation – at the same time, hiding all intermediate proof steps also removes context information about surrounding rule applications that some users would be interested in. One participant expressed this interest in surrounding proof nodes in the room for improvement for the history mechanism where he stated that he would also like to see the node before a branching node when using the option `Hide Intermediate Proof Steps`.

It is unlikely that a general solution for the proof tree presentation exists that shows the right amount and kind of information to the user for all proof situations. In this direction, further investigation by, e.g., user studies would be needed to determine how to improve the proof tree presentation.

Another repeatedly mentioned issue concerns the presentation of a single proof node, i.e., the sequent. After the application of automated proof search strategies the resulting sequents may become large and complex, hence users asked for functionalities to handle this complexity in the user study.

Partially, the requests for improvement of the interaction in the proof process are also consequences of the direct manipulation style KeY uses, e.g., reproducing previous proof states, possibly including manual interaction. Compared to systems with a script based interaction, reproducing previous proof states or also asking for more coarse blocks for proof control is more difficult in systems offering direct manipulation interaction. Some of these issues are in part addressed by the proof scripting language for KeY presented in Chapter 7. One of the issues mentioned was the stability of saved proofs after a version change of the verification system. This is an important feature, especially for direct manipulation systems where parts of the abstract proof argument, given by the user when interacting with the system, are not prominently stored for future inspection in case a proof cannot be reloaded anymore. In this regard, textual proof construction mechanisms are more robust (cf. Chapter 8 for an implementation of a proof scripting language in KeY).

One of the general topics to improve was that KeY should be applied to additional real-world case studies (examples for recent existing case studies are the verification of JDK's Dual Pivot Quicksort implementation [Bec+17] and OpenJDK's sort method for generic collections [Gou+17]). One of the obstacles identified by a participant is the missing requirement specification for some Java APIs. The common specification language enables to be able to share specifications between different verification systems – although even for JML, different viewpoints exist on the semantics of the specification constructs. Therefore, attempts to enhance the API are possible and verification systems that allow JML as specification language can benefit from such an attempt.

5.6.5. Practical Task: Proof Process

With the task `removeElem()` we focused on gaining insight into how KeY users *start* and *proceed* in the proof process.

We were able to identify abstract activities the participants performed while solving the task. The performed activities and the transitions between them are depicted in Figure 5.8 as aggregated process, where we aggregated the sequence models of all participants. The nodes in the figure are activities, the arrows between the nodes depict the transitions between the activities which participants have taken. The color and the line strength indicate the frequency that a transition was taken, i.e., darker and thicker lines indicate that transitions were taken more often. The dashed transition in the diagram are either *undo* or *cancel* operations, i.e., participants either canceled the automatic proof search or reverted actions such as rule applications. The start of a task is shown using the symbol ○→. The symbol 🕓 indicates that the task of a participant was interrupted due to the time constraints of our study, the symbol ◉ indicates that a proof attempt was successful and the symbol ⚡ indicates that the video recording was interrupted due to technical difficulties.

In the following we will first present the activities we have identified from our observations and which actions we have grouped to the respective activity. Following this description we will continue with the characteristics visible in the aggregated process. We will focus on the characteristics of the process abstracting from the single sequences. We will proceed with a description of the intents which were either directly expressed by the participants, observable during the study or added in a later stage based on context or domain knowledge.

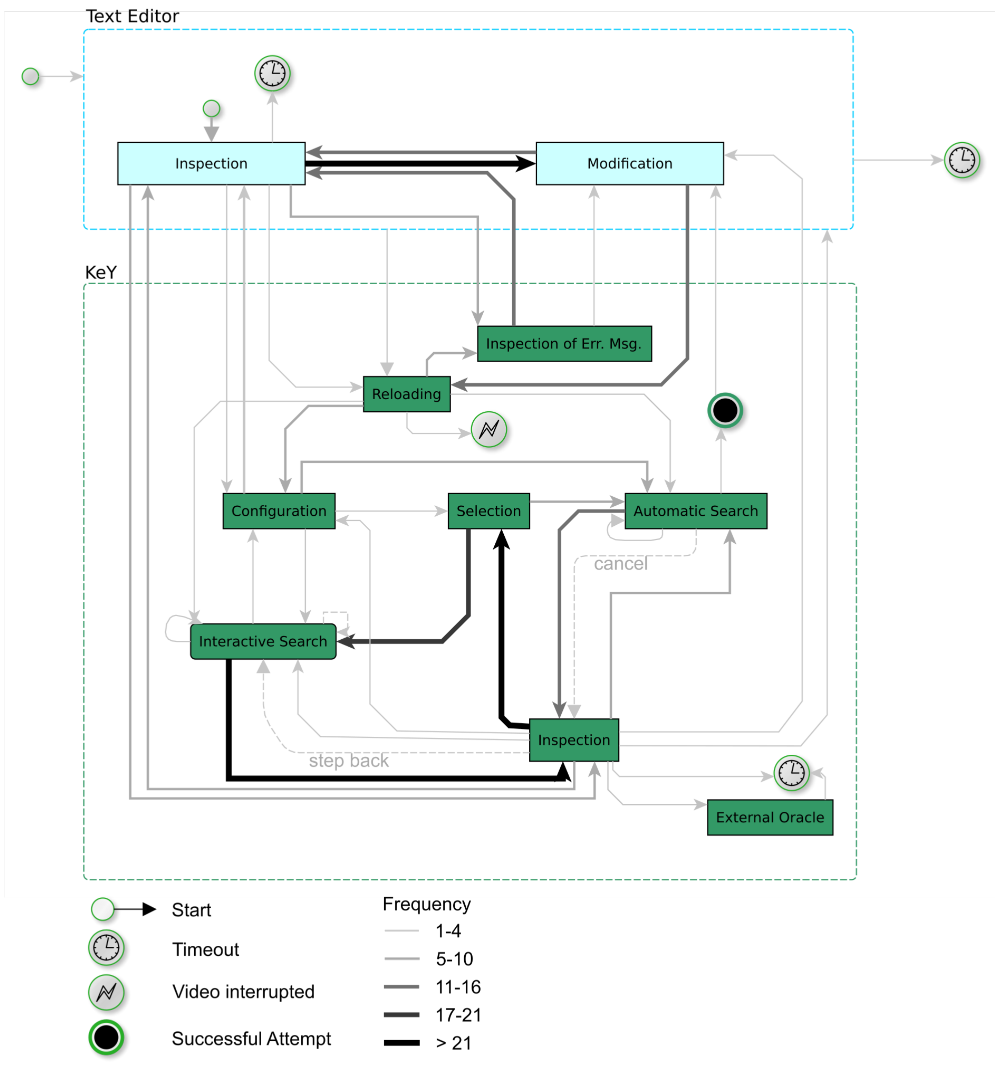


Figure 5.8.: Abstract proof process from observing the participants solving the proof task `removeElem()`

Activities in the Text Editor

Inspection of the Proof Input Artifacts. We considered the participants to perform inspection of the proof input artifacts if they read or analyzed the program or the annotations in the text editor.[5]

Modification of the Proof Input Artifacts. All modifications to the program and its annotations were considered to be actions of the activity *Modification of the Proof Input Artifacts*. More precisely the actions have been adding, deleting or modifying parts of the specification and the source code.

Activities in KeY

(Re)loading the Proof Input Artifacts. After a change of an annotation or the program, the proof input artifacts need to be loaded or reloaded into the KeY system, where they get transformed into the proof obligation that needs to be proven valid.

The loading activity may have two results: either the KeY system responds with an error message, because it could not parse or transform the input or with a dialog that allows the user to select the proof obligation he wants to prove.

Configuration. Some participants performed configuration activities during the proof process. These activities included checking and adjusting the proof search strategy settings, as well as adjusting general view settings, e.g., switching the presentation of the logical symbols to unicode symbols instead of an ASCII presentation. Adjusting proof search strategy settings included changing the number of proof steps, setting single proof search strategy settings, e.g., the option for loop treatment and setting all proof search strategy settings back to the default values.

Selection. Selection of proof nodes has been performed not only to apply strategies or rules but also to inspect the proof state. In the abstracted process we have only included the explicit selection of goals or (branching) nodes before the modification of the proof state. Furthermore, participants may have selected a goal node to inspect and immediately operate on that goal. In these situations we have also added the actions to the selection activity, as they operated on a selected goal.

There have been situations where the KeY system has pre-selected goal nodes on which the participants worked on. We have not included these situations to the selection activity, we rather added a transition directly to the state changing activity.

[5]One participant misunderstood the task and performed the same action by applying some proof rules manually to the root sequent until the program became visible. He stated that "[...] normally he would read the method's implementation [...]". We therefore added his statement and actions to the activity in the text editor, although it was performed within the KeY system. Later in the course of solving the task, he wanted to see the annotated program respectively the loop invariant and changed to the text editor.

Interactive Search/Controlled Proof Construction. We included all those proof state changing activities to the interactive search where the user either locally performed actions on a single proof goal or other actions where the user may still be able to quickly relate the newly created proof state to the one seen before the action.

This includes the use of the macro `Propositional Expansion with Splits`, the strategy `One Step Simplification`, as well as the use of single manual rule applications and the use of the more general strategy `Apply Rules Automatically here`, which only applies rules to the selected formula on the sequent.

Furthermore, we considered the use of the macro `Finish Symbolic Execution` and `Close Provable Goals Below` also to be more on the interactive side. The macro `Finish Symbolic Execution` symbolically executes the program to be proven and presents all resulting proof states to the user. Mainly, the parts of the proof state structure correlate with the program's structure and furthermore the user knows that the symbolic execution has been performed and only certain other proof steps were applied by the proof system in order to allow the symbolic execution to proceed. So we consider that the user has an idea about the proof state after symbolic execution and he is able to relate the proof state to the program. We added the macro `Close Provable Goals Below` to the interactive search as well, as applying this macro has only two possible outcomes, either it closes the goals or it leaves the proof state untouched if the goals are not closeable. So the user only has to inspect whether the goal was closed or not to gain orientation in the proof.

Automatic Search. Some participants used the automatic proof search strategies offered by the KeY system. Specifically, we assigned the usage of the general strategy `auto`, which is also accessible using the *play*-button in KeY, as well as the more structured proof search `Autopilot` (see Fig. 3.5 on page 53), accessible through the macro context menu, to the activity *Automatic Search*.

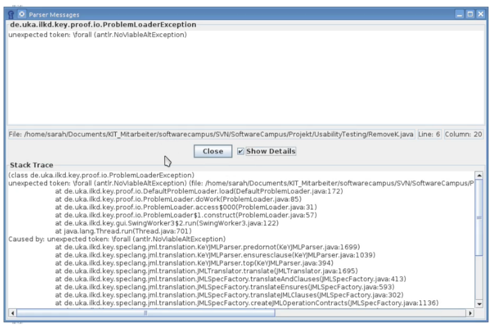
Our intention was that these strategies perform a lot of proof steps automatically, resulting in a proof state where the user often has to perform an inspection step to orientate himself. While all other strategies or macros in KeY follow a specific goal, such as finishing the symbolic execution of a program. One could argue to also include actions using the macro `Close Provable Goals Below` to this activity, as the goal of all these strategies is to close the proof. However, the macro `Close Provable Goals Below` is different as it leaves a goal untouched if it could not close the goal and therefore, in our opinion, it does not require the user to orientate himself after its application such as it is the case with all other strategies covered by this activity.

Inspection of the Proof State. All actions that did not change the proof state or were related to the inspection or modification of the proof state view were included into the activity of *inspection of the proof state*. We included actions such as navigation in the proof tree, reading the sequent or reading the branching labels in this category. Furthermore, also actions that changed the view onto the proof tree, e.g., `Hide Intermediate Proof Steps` were considered to be part of the *inspection* actions.

The inspection activities in the study were all performed either after the user inspected the annotated program, or after the application of state changing activities.

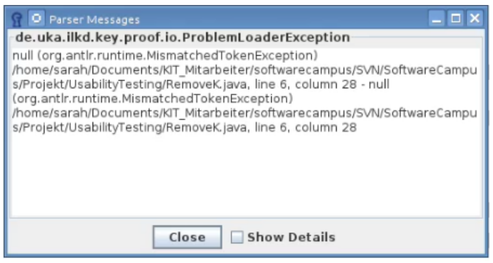
Usage of an External Oracle During the proof process users can also access two external tools in the KeY system which serve as a kind of oracle: the counterexample generator, which can find a counterexample for the proof goal and SMT solvers, which can close a proof goal without the possibility to see the proof.[6]

Inspection of Error Messages Loading an annotated program or a problem formulated in JavaDL into the KeY system may result in an error. Most of the errors are parsing errors at this stage. In such a case the KeY system responds with a message dialog that contains details about the error, and if possible also contains information about the location of the error. Examples for error messages which occurred during the study are shown in Figure 5.9. In these cases the participants inspected the error message and acted according to their understanding of the presented message.



```
de.uka.ilkd.key.proof.io.ProblemLoaderException
unexpected token: \forall (antlr.NoViableAltException)

File: /home/sarah/Documents/KIT_Mitarbeiter/softwarecampus/SVN/SoftwareCampus/Projekt/UsabilityTesting/RemoveK.java  Line: 6  Column: 20

Stack Trace
(class de.uka.ilkd.key.proof.io.ProblemLoaderException)
unexpected token: \forall (antlr.NoViableAltException) (file: /home/sarah/Documents/KIT_Mitarbeiter/softwarecampus/SVN/SoftwareCampus/P
            at de.uka.ilkd.key.proof.io.DefaultProblemLoader.load(DefaultProblemLoader.java:172)
            at de.uka.ilkd.key.proof.io.ProblemLoader.doWork(ProblemLoader.java:85)
            at de.uka.ilkd.key.proof.io.ProblemLoader.access$000(ProblemLoader.java:31)
            at de.uka.ilkd.key.proof.io.ProblemLoader$1.construct(ProblemLoader.java:57)
            at de.uka.ilkd.key.gui.SwingWorker3$2.run(SwingWorker3.java:122)
            at java.lang.Thread.run(Thread.java:701)
Caused by: unexpected token: \forall (antlr.NoViableAltException)
            at de.uka.ilkd.key.speclang.jml.translation.KeYJMLParser.predornot(KeYJMLParser.java:1699)
            at de.uka.ilkd.key.speclang.jml.translation.KeYJMLParser.ensuresclause(KeYJMLParser.java:1039)
            at de.uka.ilkd.key.speclang.jml.translation.KeYJMLParser.top(KeYJMLParser.java:394)
            at de.uka.ilkd.key.speclang.jml.translation.JMLTranslator.translate(JMLTranslator.java:1695)
            at de.uka.ilkd.key.speclang.jml.translation.JMLSpecFactory.translateAndClauses(JMLSpecFactory.java:413)
            at de.uka.ilkd.key.speclang.jml.translation.JMLSpecFactory.translateEnsures(JMLSpecFactory.java:593)
            at de.uka.ilkd.key.speclang.jml.translation.JMLSpecFactory.translateJMLClauses(JMLSpecFactory.java:302)
            at de.uka.ilkd.key.speclang.jml.translation.JMLSpecFactory.createJMLOperationContracts(JMLSpecFactory.java:1136)
```

(a)



```
de.uka.ilkd.key.proof.io.ProblemLoaderException
null (org.antlr.runtime.MismatchedTokenException)
/home/sarah/Documents/KIT_Mitarbeiter/softwarecampus/SVN/SoftwareCampu
s/Projekt/UsabilityTesting/RemoveK.java, line 6, column 28 - null
(org.antlr.runtime.MismatchedTokenException)
/home/sarah/Documents/KIT_Mitarbeiter/softwarecampus/SVN/SoftwareCampu
s/Projekt/UsabilityTesting/RemoveK.java, line 6, column 28
```

(b)

Figure 5.9.: Examples for error messages that occurred during the user study. In part (a) an error message with detailed stacktrace is depicted where the location information is presented in the middle of the dialog box. In part (b) an error message is depicted with the location information close in the white pane of the dialog box.

Characteristics of the Aggregated Proof Process

In Figure 5.8 we have depicted the aggregation of all processes which were observable during the task completion in the study. We abstracted from the individual, concrete actions by using the aforementioned activities.

We were able to observe that during the process of solving our task the participants switched between the KeY system and the text editor containing the annotated program.

[6]One participant used the counterexample generator during the completion of the posed task.

Interplay between KeY and the text editor. We were able to identify that the transition to the text editor was taken at different stages in the process. Participants started the process in the text editor to provide the annotated program or to inspect the annotated program to find out what should be proven.

In the course of the process the transition to the text editor was taken to either inspect parts of the annotated program, e.g., to find the relation to the currently visible proof state or to modify respectively enhance annotated program, e.g., by completing the specification or changing the source code. For finding the relation between the proof state and the annotated program it was visible that some participants used clues found either from inspection of the sequent or from the labels in the proof tree. Sources for hints from the sequent were for example the modalities, updates or specific formulas on the sequent, such as a formula formalizing that an exception was thrown. If participants used information from the proof tree to relate to the program, often the information resulted from the labels of branching nodes that seemed to indicate control-flow decisions (branching nodes containing an if condition) or loop invariant cases were used to guide the inspection of the program.

After modifying the proof input artifacts the participants (re)loaded the proof problem into the KeY system. At this stage of the process either KeY responded with a selection dialog, where the contract that should be proven can be selected or with error messages. These error messages were mainly parsing errors, i.e., the input artifacts contained syntax or typing errors.

In case loading of the problem caused KeY to respond with error messages, the participants inspected the error message and switched back to the text editor. Two participants spent a lot of the task time in the text editor and discovered error messages while loading (mainly syntax errors for their specification), such that the sequences of these participants did not contain any other interaction with the KeY system except for the loading procedure and the inspection of the error messages. We had to abort their tasks due time constraints. Additionally, two other participants experienced a problem loading error as well, however, these participants still had time to interact with KeY after error correction.

The inspection of the error messages included that the participants read the information given in the (upper) white box (in Fig. 5.9 (a), the upper box) and if the information did not fully help also the detailed stacktrace was inspected (in Fig. 5.9 (a), the lower box). During the user study two kinds of error dialogs appeared. One where the location information was presented in the user focus (see Fig. 5.9 (b)) and a second one, where the location information was presented below the user focus (see Fig. 5.9 (a)). In the latter case participants did not always discover the relevant information at first sight. If the information about the code location was discovered participants inspected the source file at the given location for errors.

Processes within KeY Within the KeY system two processes are visible. The first process is the transition between *reloading, configuration, automatic proof search* and the activity *inspection.* The second process that is prominently visible is the loop between *inspection, selection* and the activity *interactive/controlled proof construction.*

The first process is performed after successfully reloading a problem into the KeY system. The settings for the general proof search are either inspected or set, e.g., to personal default values or to specific values to control the automatic proof search to a certain extent. Succeeding this configuration activity the participants started the automatic proof search. The resulting proof state is inspected to gain an overview and knowledge about what happened during proof search. According to the gained insight during inspection the participants either continued automatic proof search, switched to the second loop to perform a more local controlled interactive proof search or completely changed to the text editor or an external oracle.

In case of the second process, where participants decided to perform a more local interactive proof search, a goal or subtree is selected to apply a rule or a specific macro to it. The decision which goal or node is selected and which next step to perform in the proof search is based on the knowledge from the inspection activities. For example, a participant expected that a goal should be closeable, but was not certain about his hypothesis. He then applied the macro `Close Provable Goals Below`, which tries to close the goal or leaves the goal untouched if his expectation was not met.

After the interactive application of proof rules or macros the resulting state is inspected and a decision about the next action is made according to the insight gained from the inspection activity.

Intents in the Proof Process

In the following we will summarize the intents the participants had when performing the observed activities. In some cases an intent covered a sequence of activities. In this case we have included the intent in multiple activities, if appropriate. For other activities of participants it was not possible to observe or infer an intent.

Inspection of the proof input artifacts. At the beginning of the proof task, participants inspected the proof input artifacts in the text editor to gain insight into the proof problem. More precisely, the inspection was performed to understand and analyze the program behavior, in addition to the already given specification via JML annotations.

After either modifying the proof input artifacts or after gaining information from a proof attempt the participants' intent to inspect the proof input artifacts was to *review* the specification. In more detail, the participants reviewed the specification to check (mentally) whether the specification holds or to make sure that they did not provide an inconsistent specification.

Throughout the proof attempt, one intent of the participants to inspect the proof input artifacts was to find the relation between the observed proof situation (respectively the current proof state) and the location in the source code. One particular instance of trying to relate the proof state to the proof input artifacts was to determine the cause for the currently open goal.

The participants also inspected the proof input artifacts to look for errors in the annotated program. The search can be undirected (e.g., by looking out for errors as part of reading and understanding the program and specification) or directed after feedback from the proof system (e.g., after a syntax error was signaled by KeY or after inspecting the currently open proof state).

Modification of the proof input artifacts. Besides inspecting the proof input artifacts, the second apparent activity of users in the text editor is to modify either the program or the specification. At the beginning and after a successful proof attempt the participants' intents were to develop the (full functional) requirement specification, together with the corresponding auxiliary specifications. In some cases, the specification was developed in a single step; in other instances, a step-wise modification was used, e.g., by gradually strengthening the loop invariant or postcondition.

Another intent to modify the proof input artifacts was to reformulate the program according to the user's preferences, without changing the behavior of the program (e.g., by rewriting boolean expressions in an if-statement). The goal in this case was not to simplify the program for the verification system, but to clarify the program behavior for the user. The same intent and goal is also applicable to the modification of the annotations.

The proof input artifacts can also be modified to control the proof search before starting the proof attempt. One exemplary secondary intent is to weaken the specification, to be able to simplify the first proof attempts.

After inspection of the proof input artifacts, e.g., because the KeY system signaled an error or the proof search resulted in an open proof, the participants modified the artifacts to correct syntax errors, the specification or fix the program.

(Re)Load proof problem. Besides the apparent intent of starting (or restarting) a proof attempt by loading the proof input artifacts into the KeY system, a secondary intent was to use KeY as a validation tool: in some cases, participants used the activity of loading proof artifacts to determine whether the specification could be parsed or whether possible syntax errors have been successfully fixed after a modification of the specification.

Inspection of error messages. During the user study the KeY system signaled syntax errors in some cases. If this was the case the participants inspected the error messages raised by KeY to determine the reason for the syntax error. In particular, the type of syntax error and also the location of the erroneous statement in the proof input artifacts were information the participants intended to find by inspecting the error messages.

Configuration. After reloading the proof problem, but also within a proof attempt some participants changed the configuration of the KeY system. One type of configuration change was concerned with the presentation of the proof obligation, e.g., one participant adjusted the sequent view settings to display unicode symbols instead of using ASCII notation in order to improve readability of the sequent. This type of change is typical for a configuration activity performed once at the beginning of a proof attempt.

Another more prevalent type of configuration that was performed both at the start of a proof attempt and during proof construction was to change the proof search strategy settings: These settings have been set by the participants to either the general default values, or to personal preferences, e.g., to fit the values the participant had good experiences with in previous proof situations. One instance of resetting the proof strategy configuration during the proof attempt was that either proof search with custom settings was performed so far, or the participant performed other activities in the proof attempt and wanted to continue with the default proof search.

Proof search strategy settings have also been changed during proof construction to give the prover more resources (i.e., by increasing the number of rule applications available for the automatic search) with the intent of being able to discharge parts of the proof problem the participant was confident the verification system could handle.

Another application of changing the strategy settings was to configure the proof search to not automatically apply certain rules for symbolic execution but to let the user manually apply rules at this point in the proof – with the resulting effect being similar to setting a breakpoint in symbolic execution. In the specific case in our user study, rule application was configured to stop before applying the loop invariant.

Selection. Besides the implicit default selection of proof nodes by the KeY system, one activity of the participants during the proof task was to select both inner nodes of the proof tree and open goals, depending on the underlying intent of the activity.

One category of selection activities had the intent of proceeding with the proof task, e.g., after previous inspection actions. In the next step after selection, participants would then continue the proof process by applying proof rules or macros to a single goal or a whole subtree. In this context, goals were selected either according to an estimation about their complexity, according to their occurrence order in the proof tree or because of a property that a participant considers important or interesting.

A second intent for selection activities was to select goals or inner nodes as part of further information retrieval, e.g., to determine why a certain goal could not be closed by the automatic proof search. In this case, selecting single nodes was also performed as part of navigating either towards a specific piece of information in the proof that the participant explicitly searched for, or as part of a sequence of actions looking for clues how to proceed.

Inspection of the proof state. Intents for the inspection of the proof state ranged from gaining a rough overview about the current proof situation to determining the exact meaning of single goals and formulas, together with how they were derived from the original proof obligation. Accordingly, the type and amount of information viewed by the participants varied from inspecting the structure of the whole proof tree, the (branching) nodes of the proof tree together with the labels of the nodes down to individual sequents of the nodes and open goals.

One goal for inspection of the proof state was to gain orientation in the proof. To gain an overview over the current proof situation, participants included information like the list of open goals, the structure of the proof tree (where often intermediate proof states have been hidden in order to obtain a better view of the overall shape and contents of the proof tree), as well as labels of the proof nodes.

A second intent for the inspection actions was that participants wanted to determine the reason for resp. origin of an open goal. The level of detail varied from gaining a rough estimation about the cause of an open goal up to understanding in detail why an open goal is formed during the proof construction activity. Depending on the aim, either more high-level information was inspected, e.g., branching nodes and their labels, narrowing down on specific properties of single nodes, e.g., one participant inspected open goals more thoroughly to observe the change of a manually applied rule.

The proof state was also inspected by the participants to localize specific goals that seemed interesting or important, or to relate the proof state or specific formulas of the proof state to the proof input artifacts, e.g., the node with the label `emptyModality` that indicates the end of the symbolic execution in the proof which in turn signals that a final state of the program has been reached.

Finally, as already mentioned in the previous paragraph on selection activities, participants also inspected the proof state to find clues for the next step in the proof process.

Automatic proof search. For proof construction the participants performed activities that we categorized as automatic proof search. Especially at the beginning of a proof attempt, the general automatic strategy of KeY was used to estimate the proof complexity, i.e., to identify whether a part and which part of the proof problem needs further attention. The participants used the strategies to find out whether the proof problem is provable or which part is not provable and either needs modification of the proof input artifacts or user interaction.

The strategies were also used by the participants to perform undirected proof search, i.e., to let the prover find a proof automatically. Instead of using a configuration activity to control the proof search resources (number of rule applications and depth) the participants also intervened the proof search during automatic rule application, either to stop the proof search before all steps were used or to repeat the proof search activity. These decisions were made based on previous experiences concerning the progression of the automatic search.

However, also for a more directed search the strategies were used, to perform proof search up to a certain point in the proof: a participant first configured the strategies, such that the proof search stops before the loop invariant can be applied, resulting in a proof state where symbolic execution had been performed just up to the loop in the program. After the termination of the symbolic execution strategy, automatic strategies were applied to the resulting state.

Interactive/controlled proof search. For proof construction and proof search, activities were performed that were applied in a more controlled way. We categorized these interactive proof search activities into (a) the resulting state after performing the activity is either completely known a priori or (b) the user has an idea about the shape of the result. As one example, to close the gap between the mental model about the proof state and the actual proof state a participant pruned a proof that was performed using the automatic strategy and used the macro `Finish Symbolic Execution`. The participant used interactive proof search activities to control that the proof search stops at a point from which he knows the relation to the proof input artifacts.

Interactive/controlled proof search activities have been used to perform a targeted review of expectations or hypotheses, often after inspecting the proof state and selecting a specific proof goal. To locally check whether single goals are closeable or need further attention the macro `Close Provable Goals Below` was used by some participants. Moreover, to verify whether the own expectation about the provability of a goal is correct, i.e., whether a goal should close, the macro `Close Provable Goals Below` was used. A third intent for participants to perform controlled proof search was to locally check whether changes in the proof input artifacts affected the validity of the proof obligation.

A further intent for the use of interactive proof search was to split the proof task into smaller tasks from which the participant knows the relation to the proof input artifacts. Here, either the macro `Autopilot Preparation` or a sequence of macro applications together with single rule applications (proof rule andRight) have been used by some participants. One further intent of the interactive/controlled proof search, more specifically for the use of the `One Step Simplification` step, was to determine the truth value of the postcondition after update application.

Manual rule applications were also applied to directly observe the effect of a rule application and check whether the expectations of participants are met concerning the resulting state.

Usage of an external oracle. We were able to observe that one participant used an external oracle, the counterexample generator of KeY, to determine the cause of the current open goal.

Discussion

In deductive program verification, users have many possibilities to choose the next promising step to complete a correctness proof. These degrees of freedom were also visible in the aggregated process model, where not one single path through the process was clearly visible, rather many ways to accomplish the task were observable. As the verification task is a creative task these degrees of freedom are necessary and need to be supported by the verification system.

In general, the proof task posed in this part of the user study has shown that inspection and selection are two important activities for proof search and construction that need to be supported in an interaction concept for a deductive program verification system, such as KeY.

From the aggregated process model it becomes apparent that inspection activities are not only performed in one system (either the text editor or KeY) but participants used both systems and switched between them during inspection. These transitions indicate that the participants try to relate the artifacts of both systems, which was also expressed as one of the intents during the study. In some cases participants also directly modified the proof input artifacts after inspecting the proof state in KeY. A switch between the text editor and the KeY system requires that users need to find the relation between the two different representations (proof input artifacts and proof state). Changing between the two systems requires a context switch which increases the costs for users in terms of time and mental effort.

Regarding the frequency of transition usages, one noticeable characteristic is the difference in the frequency between going through the loop between activities "automatic proof search" and "inspection" and the frequencies of the transitions in the loop for the interactive proof search. We assume that the reason why the interactive proof search loop is taken more often than the automatic proof search loop is, because it is necessary to apply more fine-grained steps interactively to achieve a desired effect, in contrast to using automatic proof search steps. In the case of applying the automatic proof search strategies the fine-grained steps are applied by the system which often resulted from only one single user interaction.

One hypothesis is that the effort for the user is increased in the case of automatic proof search steps as the user has to gain orientation in the proof afterward. In the interactive case, however, we assume the effort to be lower as the user was able to inspect intermediate results and the steps have been chosen by the user to achieve an intended effect on the proof state.

One information that does not become apparent from the aggregated model alone is how users correct (interaction) errors (the dashed arrows labeled "step back" in Fig. 4.1) or handle unfinished proof attempts. We only observed a few interaction errors, because the goal of the task was not to observe a full task completion (due to the limited time we had for the user study). In order to observe more error handling activities, one possibility would be to perform the user study with one single user task.

One of the error correction activities we noticed in the user study was that participants reverted single proof steps during the task after the resulting proof state was not as they expected. Another option to recover from errors was to prune the proof at selected proof nodes, which was mentioned during the interview questions. Intents for pruning the proof are that the user has lost the overview, especially after the application of the automatic proof search strategies, or that the resulting proof state is not as the user expected.

Errors that have been observable in the user study were syntax errors in the proof input artifacts which were reported by KeY after reloading the proof input artifacts. The participants inspected the error messages in KeY and switched back to the text editor to find and correct the error. The current feedback about syntactical mistakes in the proof input artifacts provided by KeY is given rather late in the specification process. One improvement would be to provide this feedback already while writing or saving the proof input artifacts. In addition, providing the information about (syntactical) errors in the proof input artifacts in a different system than the one the artifacts were written in, necessitates the user to perform a context switch. In Chapter 10, we will introduce a concept to integrate both processes, text editing and program verification, to reduce the effort needed when switching between the two activities.

Generally, the observations for this task provided insights into the interaction with KeY for proof construction and alongside showed opportunities to improve this interaction, by improving the connection between activities in the text editor. This connection would for instance support reviewing the specification, relating the proof state to locations in the annotated source code and writing (or completing) the specification.

One prominent observation in the user study was the variety and amount of information used by the participants during the inspection actions. It was, however, not always possible to gain insight into the intents of the participants when accessing proof artifacts. To be able to appropriately support all intents and improve the state presentation in a new interaction concept, it is important to determine the intents for all actions performed, possibly in a further user study, e.g., by conducting a user study only with expert users of the system.

During the user study, we were often only able to derive which exact information a user inspected from the actions a user performed (e.g., pointing with the mouse cursor) or from explanations the user gave. It may be the case that inspection actions were thus also performed that could not be noticed in the study – here, a further experiment with other techniques such as eye tracking could be worthwhile to determine the exact kind of information observed by the participants in each step.

Participants mentioned or performed controlled proof search and proof exploration. When developing a new interaction concept the explorative nature of proof construction has to be supported. Also, proof search strategy settings were changed in the middle of a proof during the user study. This action may be an indicator that users may benefit form a proof scripting language that offers a more fine-grained control over the application of the strategies in KeY.

We also observed that one proof construction strategy was to successively strengthen a specification to prove parts of a property step-by-step. This approach may be replaced by generating single verification conditions for each of the properties similar to systems such as Why3.

5.6.6. Orientation After Applying Automatic Strategies

In the study we included tasks in which we showed an open proof to the participants and asked them to explain the proof situation by explaining what has happened and how they would proceed in that situation. Our intention with this task was to gain insight into how users of the KeY system orientate themselves in the proof situation. The explanation of an open proof state may also contain the information about the cause for the unfinished proof attempt. One way of determining this cause in the KeY system is to change the proof state, either by pruning the proof and performing the proof in the user's favored way or by again using the automatic strategies to determine whether an insufficient number of proof steps was the cause.

In this task we wanted to gain insight into the possibilities KeY users have for the comprehension of the proof state *without* changing the state itself. This restriction was communicated to the participants if they wanted to change the state.

The outcome of this task is a more detailed insight into both, the activities *inspection* in the KeY system and *inspection* in the text editor as well as their interplay.

While, from a high-level overview it was observable that the individual actions differed between participants, we were also able to observe that participants used common sequences of abstracting from and focusing on individual proof nodes, so to obtain the more *global* view onto the whole proof and to focus on *local* aspects of the proof (state).

In many of the observed sequences it was visible that the option `Hide Intermediate Proof Steps` was used to gain an overview of the proof state. Some participants reverted this view action when focusing on details of the proof. We assume that the option was also used when the participants did not find the information they were searching for on the filtered view, which did not contain the intermediate proof steps.

In the following we will give a detailed insight into our analysis of the sequence models we have obtained during the analysis of the user study. For this, we have created sequence models for the tasks `split()` and `palindrome()` based on the observation in the user study when observing how the participants gained orientation in the proof after the application of the general automatic strategy of KeY. To be able to observe the orientation process we asked the participants to explain what has happened in the proof respectively to explain the proof situation. Similar to the analysis of the sequence models for the proof process (see Sec. 5.6.5) we have abstracted from the concrete actions to obtain activities capturing the intents of our participants. We also included those activities into the sequence models for data analysis that were actions to further gain orientation in the proof, from the succeeding question on how the participants would proceed after the orientation task. We then aggregated all sequence models into one large model (see Fig. B.2 in the appendix).

As the whole model captured the large amount of degrees of freedom KeY users have in orientating themselves in the proof, we have filtered out those activities and their transitions which were taken at most twice by all participants together in the aggregated model. The resulting model is depicted in Fig. 5.10. It is important to note that in the complete aggregated model it was not possible to identify a uniform strategy across all participants.

A more detailed insight into orientation in the proof process.

In Sec. 5.6.5 we have already briefly introduced the intents of the participants to inspect the proof state. In the following we will focus on the (secondary) intents which were either mentioned explicitly or which we added in the analysis stage of the sequence models for the tasks `split()` and `palindrome()`.

The abstract activities in the sequence models which we have identified are summarized in Table 5.2. We will give details about those abstract activities that are present in the filtered model in the following.

Gain insight into the proof problem We have categorized the actions that participants analyzed the proof input artifacts and retraced the program's execution (especially at the beginning of the task) into the abstract activity of gaining insight into the proof problem. In some cases the participants even expressed that they want to observe what should be proven in the proof task respectively to understand the specification.

Text editor	• gain insight into the proof problem • review specification • find relation to the current/observed proof situation • localize errors/mistakes/issues in the proof input artifacts
KeY system	• find clues/hints for the next action/step • gain overview • gain (more detailed) information about open goals (from labels, branching nodes) • determine reason for open goal/rule out a reason for an open goal • gain rough estimation about open goal • localize/select (specific) open goals/nodes • relate proof state to proof input artifacts • retrace proof/observe changes by rule applications • analyze specific node/sequent for details • inspect proof search strategy settings • change view settings • perform mental deduction steps

Table 5.2.: Abstract activities we have identified in the sequence model, distinguished whether performed in the KeY system or the text editor.

Find relation to the current/observed proof situation The actions to analyze the program as well as retracing the program's execution, especially after inspecting the proof state were categorized in the activity of "Find the relation to the current proof situation". In addition, if participants searched for specific statements or annotations that they have explored in the KeY system before, we also categorized these actions as finding the relation to the current proof situation.

Gain overview If participants expressed that they want to find out "(roughly) where they are", "comprehend the situation or the proof" or "find out which branches are in the proof" especially towards the beginning of the task or after they expressed that they do not know where they are at the moment, we included their actions into the activity *gain overview*. Typical actions that we observed have been to use the view option `Hide Intermediate Proof Steps` to change the view onto the proof tree to a reduced proof tree only containing branching nodes. After changing the view participants often analyzed the shape of the proof tree or scrolled in the reduced proof tree. Another example we have observed for gaining an overview was the use of the goals tab in KeY where a list of all current open goals is shown to the user.

Gain (more detailed) information about open goals (from labels, branching nodes) Often after gaining an overview participants started to analyze the proof tree more thoroughly, for example by navigating (upwards and downwards) and inspecting the branching labels in more detail, expanding subtrees or unselecting the option `Hide Intermediate Proof Steps`. In some cases at the end of this activity specific nodes have been selected for further detailed inspection.

Localize/select (specific) open goals/nodes During the orientation process participants selected or localized specific goal nodes or inner nodes. Selection or localization criteria could have been gained from inspecting the program or a predecessor or successor node. The selection was also done in cases where participants observed a branching label that raised their attention, for example a label indicating that an exception was thrown.

Relate proof state to proof input artifacts Based on earlier inspection activities of the annotated program and from participants' experiences in verifying programs with KeY, the relation to the proof input artifacts was searched. The precision of determining the relation between proof state and proof input artifacts reached from a rough localization for example relative to the loop and loop invariant, to specific control flow statements, such as if-conditions. One example of our study was that a participant navigated to specific nodes that indicated the end of the symbolic execution and from there on to symbolic execution nodes that corresponded to assignments in the program to see which values changed during a loop execution. In some cases it was observable that participants used specific nodes, such as the different loop invariant cases, the `emptyModality` node or specific symbolic execution or branching nodes (for instance if-conditions) to relate the proof state to the program's execution.

Analyze specific node/sequent for details After activities to gain overview over the proof (state), we were able to observe that participants inspected individual sequents more thoroughly. One reason was to gain a detailed insight into the current proof obligation, i.e., participants wanted to know what exactly has to be proven in the current state. Participants also searched for specific formulas on the sequent while inspecting the individual sequents. To gain insight into the cause of an open goal and to form a hypothesis about this cause, the individual sequents were inspected in detail.

Characteristics of the aggregated model

In Fig. 5.10 we have depicted the filtered orientation process, that is a filtered view of the aggregation of all sequence models into one large model (see Fig. B.2 in Appendix B). To obtain the filtered model from the full model, we have filtered out those activities and their transitions which were taken at most twice by all participants together in the aggregated model. In this filtered model we were able to recognize relevant usage patterns and regularly used transitions.

In this process one significant observation is a strategy that corresponds to a stepwise focusing on details of the proof state. First an overview over the proof is gained, by for instance filtering the proof tree view to show only branching nodes and inspecting

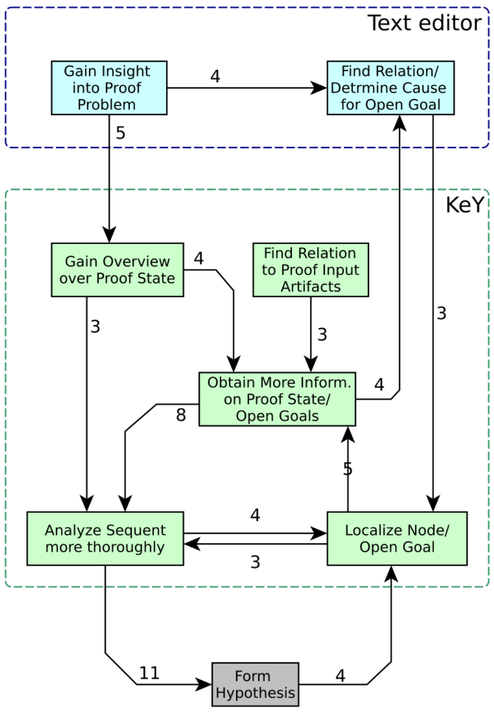

Text editor
Gain Insight into Proof Problem
4
Find Relation/ Detrmine Cause for Open Goal
5
KeY
Gain Overview over Proof State
4
Find Relation to Proof Input Artifacts
3
3
3
Obtain More Inform. on Proof State/ Open Goals
4
8
5
Analyze Sequent more thoroughly
4
3
Localize Node/ Open Goal
11
Form Hypothesis
4

Figure 5.10.: Filtered model for the activities of the task `palindrome()` and `split()` in the user study containing only those transitions that were taken at least three times by all participants together.

the filtered tree. Succeeding this step, more details are inspected, by for example expanding subtrees, displaying intermediate proof nodes and scrolling in the proof tree to inspect the labels of the proof tree more thoroughly. In some cases the inspection for details was targeted towards gaining information about the path to the open goal. As the most detailed zooming step the sequents of individual proof nodes were inspected for details. One focus was to inspect whether formulas such as the modality are still present on the open goal or to inspect the modality and its preceding update. Another focus was on specific formulas that contained terms that were also observed during the analysis of the branching labels or formulas which were familiar to the participant from, e.g., former experiences in using KeY. From the detailed analysis of the sequents we could observe that hypotheses were formed about the cause for the open proof.

A further strategy that became visible after filtering the aggregated model was that when participants inspected a sequent for details it was sometimes also observable that a zooming out from the detailed inspection was performed by localizing and selecting other nodes to gain more context, e.g., by (not necessary directly) neighboring nodes or branching nodes above, and then either inspecting the selected node more thoroughly or zoom out further to scroll in the proof tree. In some cases when a hypothesis was formed about the reason for the open goal, participants localized certain proof nodes to further inspect them. We assume this was performed to either sharpen respectively change the hypothesis or to confirm the formed hypothesis.

We were also able to observe that participants searched for the relation between the proof state in KeY and the program's execution and annotations relative to the information gained from inspecting the proof tree in more detail. This observation is not surprising as the information presented in the labels may have a correspondence to the program, e.g., case splits in the proof resulting from symbolically executing an if-condition of the program. In the other direction participants related the proof state to the proof input artifacts in the text editor by switching back to the KeY system. In this case specific nodes that corresponded to the previously observed program execution were localized and selected.

In some cases participants related the proof state to the proof input artifacts by using knowledge about the proof structure, e.g., by the distinctive nodes such as the end of the symbolic execution or the split between normal and exceptional execution. Based on the position in the proof obtained by the relation actions, participants navigated in the proof tree to gain more detailed information about the proof. This strategy points out that participants may use distinctive features to start the search for more detailed information.

Artifacts and actions used to inspect the proof state. As a more general insight we collected the artifacts the participants inspected to gain orientation in the proof after applying automatic strategies and to find out the next promising steps. The participants used or named the following sources for information:

- source code and specification
- proof tree
- proof tree labels
- search for terms or formulas in the proof tree and sequent

Participants also mentioned activities they would perform to gain orientation and determine the next promising step:

- retry proof attempt
- macro `Close Provable Goals Below`
- use the strategies again
- counterexample generator

Proceeding step after orientation. We asked the participants at the end of the orientation task how they would proceed. Unsurprisingly there was no uniform answer, rather the answers were related to the observation and the knowledge gained during the orientation task. In the following we have briefly summarized the answers. Some participants would first try to use the automatic strategies, to see whether KeY is able to close the proof. If that attempt is not successful, participants would try to gain a deeper comprehension of the proof problem and proof state or would manually explore possible rule applications on the open goal, e.g., quantifier instantiations or to inspect which applicable rule may be meaningful to the participant. One possibility to gain a deeper understanding is to use pen-and-paper to clarify the situation.

Some participants would either start a new proof attempt (e.g., because they have lost the overview) and use the macro `Autopilot` together with manual proof control, or perform the whole proof interactively.

Other participants would search for mistakes in the source code or the annotations, probably also because they already suspect an error in the proof input artifacts after the orientation activities.

Changing specific parts of the proof input artifacts was also mentioned as a step to proceed. These participants seemed more convinced that there is an error in the program at specific locations.

One participant mentioned to use the counterexample generator to generate a counterexample for the proof goal. In case no counterexample could be generated, he would try to close the goal using an SMT solver. If this action was also unsuccessful, he would generate a test case and use a program debugging system to step through the programs execution using the test case.

Discussion of the Results

Unsurprisingly we were able to observe the many degrees of freedom that the participants have when gaining orientation in the proof using KeY. A central observation was way information is presented in the KeY system and the way participants retrieve information. For gaining the orientation all information is presented, however, the user is overwhelmed by the amount of information. The relevant information is not clearly visible at first sight, rather participants used a non-linear process to identify the relevant parts. The process for identifying relevant information was performed by relating parts of the proof state to the mental model, e.g., to the annotated program. One reoccurring pattern to identify relevant information was to zoom into the proof from an overview over the proof tree to single sequents of the proof nodes and open goals. Interesting or relevant information was often discovered unanticipated while, e.g., zooming in for details. One example where this becomes apparent is when looking into the activity *find clues/hints for the next action/step (in KeY)*, where participants, e.g., scrolled in the proof to look for hints where to search more thoroughly.

We also assume that participants selected specific nodes to have a reference point from which on they searched for more details and with which they are able to relate a newly inspected goal. These reference points are often nodes that have a direct relation to the proof input artifacts.

The information retrieval was a time-consuming task and has room for improvement. Different types of information that should be presented separately to support the users' mental model are presented interwoven with each other. Steps in the proof tree are logical rule applications, as well as steps that transform the program using symbolic execution. This interweaving of information is not an issue specific to the KeY system, rather it is a peculiarity of the problem and task domain. The amount of information presented is large on the global, as well as on the local level: the proof tree contains a large amount of proof steps and each sequent can contain a large amount of formulas, sometimes filling several screens. Furthermore, the formulas on the sequent can each become large and complex in their structure, e.g., formulas that formalize the heap contents. The information structure therefore requires the user to search through a lot of information to identify the relevant information for the next promising step. These issues may result in situations where users might abandon the proof and restart it with a different proof search strategy. For example, one participant lost the overview over the proof and stated that he would start the proof process over again using the macro `Autopilot` and proceed with manual steps afterwards.

In this thesis we will address the issue of interwoven information of different types in the proof state in Chapter 8 and Chapter 10.

5.7. Results of the User Study – Origin of Formulas

5.7.1. History Mechanism

We conducted this user study using the *thinking-aloud* method in conjunction with a small interview to achieve two goals: to identify where in the proof process users search for which kind of information together with identifying usability issues.

The second goal was more specific and tailored to our assumption that users need the information about the origin of formulas during the proof process to orientate themselves. Therefore, our goal was to find out how our first mocked prototype for showing the syntactical origin of formulas fits the users needs. This included to observe how the participants try to find the origin of a formula without and using the mocked mechanism (presented in Sec. 5.2.2). Furthermore, we wanted to gain first insights into the level of granularity the participants would want to have the mechanism to work on.

Expectations About the Mechanism Prior to Use

As a first step in the user study, we asked the participants what they would imagine what a functionality called *Show History* is. Most of the participants had the right intuition what this functionality may show and answers were, "where a formula originally came from" respectively "how it was derived" or "for a a subformula show, which rules were applied to get that formula".

However, the pre-enquiry revealed that a better name has to be chosen for the mechanism. Some participants already explicitly stated that they would imagine with that name that the functionality shows the origin, when considering the previous tasks in the study. Other participants did not come up with the right idea at all, e.g., one answer was "Shows the development history of the KeY system". Therefore, the name does not fully reflect our intention of the functionality.

After the participants were able to use the new functionality in two tasks, we explored their opinion about the functionality, where it could be used, whether the functionality worked as expected and asked for room of improvement.

Overall Opinion

The majority rated the functionality as being good or helpful. They also gave more details about their opinion or put their opinions into perspective.

However, the expectations of the participants were not fully met for all participants. One frequent criticism was the highlighting was too coarse, for example in the root sequent, where the whole proof obligation was colored.

One participant was not sure how the highlighting worked and was confused, because of the two colors (the red highlighting from the mechanism and the regular green highlighting present in KeY to indicate the position of a rule application) present on the sequent in some cases.

And one participant was unsure about the general helpfulness of the functionality as this participant was indecisive whether he requires this information during proof construction.

We furthermore asked the participants, whether they would use the mechanism and for which purpose. Six participants would use the mechanism. Two participants stated that they would not want to ask such questions for Java problems, respectively their mode of work is more towards gaining a general understanding of the context. One participant would use the mechanism, if he makes the experience that it helps him.

During the questions about the opinion, the expectations and the application areas for the mechanism, the participants expressed their opinions about the mechanism. We grouped the opinions of the participants in three main categories: *supportive*, *unsupportive* and *indecisive*.

The participants considered the mechanism to be supportive or unsupportive for accomplishing specific *intents or tasks* or *application areas or goals*. Some also expressed opinions about specific *features* of the mechanism.

Supportive. Some participants expressed more *generally* a positive opinion about the mechanism. Two participants stated the mechanism covers their manual course of actions.

More specifically *tasks or intents* were mentioned where the mechanism might help. The intents or tasks for which the mechanism was considered supportive ranged from very general or abstract tasks or intents to very specific ones. The more general descriptions included to support the user in understanding the current proof situation or as support to localize points where the user has a relation to.

It was further mentioned that the mechanism supports users in cases they want to find or track the origin of symbols, formulas or expressions in situations where for example the origin is not syntactically visible because of normalization steps. Also, to reconstruct the original formula was mentioned as intent.

Furthermore, it was mentioned that the mechanism helps to find the relation between the formulas in the proof state and the specification, or serves as aid towards finding this relation. The intents to find out why a proof does not close or a formula is not provable (e.g., after the application of the automatic strategies) has been seen as an intent where *Show History* can be supportive.

On the more abstract level the participants also named *application areas* or *specific structures in the proof situation* where the mechanism is of support for the user. For unstructured proofs, e.g., proofs, which did not result from proof problems for Java programs the mechanism seamed helpful. If specific structures, such as large terms resulting from ADT specifications or quantifiers are present on the sequent the mechanism was considered supportive as well. Also, if normalization steps have been performed and temporary symbols have been introduced during the proof process, the mechanism was considered to help to trace back these temporary symbols to symbols resulting from the program and therefore to help recover the relation to the program to prove. One participant mentioned that if the user is deep in the local proof, the mechanism seemed supportive. *Features* that have been considered supportive have been the tool integration, as the preferred way of navigating in KeY is still usable, the filtering of nodes and the possibility to have a visual guidance both, on the sequent and in the proof tree to focus on.

Unsupportive. The participants named *intents* or *tasks* for which the mechanism is not supportive. For gaining an *abstract proof comprehension*, respectively the bigger picture participants deemed the mechanism not supportive. One participant further on mentioned that his course of work is more to understand the global context.

For the intent of finding our which formula is interesting or relevant the mechanism was also considered not to support the user.

For accomplishing the task of finding the origin on the program level, it was mentioned that the mechanism does not help. Furthermore, one participant mentioned that the amount of formulas highlighted in the root sequent for Java programs may not support the user. This leads to the *application areas* where the participant's opinion was that it is not helpful: for proofs resulting from Java programs.

The granularity of the highlighting in the root's sequent was mentioned to be not supportive for finding the origin, as it still requires scrolling in the proof tree.

Two critical opinions have been that the colors distracted the participant and the color scheme needs explanation, as otherwise it is confusing.

Indecisive. Some participants were not able to express a final assessment yet. We have added their opinions to the category *indecisive*.

One participant did not fully understand the highlighting from the two tasks and was confused about the colors visible during the usage of the mechanism. Another participant was interested in the behavior of the mechanism if proofs are large or formulas and terms were not present on the initial sequent. One participant mentioned to be unsure whether he actively asks the question about the origin during the proof process.

Expectations after Usage. After the participants were able to use the mechanism we wanted to know whether the mechanism's behavior matched their expectations. The responses were mixed, some participants stated that it matched their expectations, some were not able to fully judge from the short period of time using the mechanism whether it matched their expectations.

Some participants either expected more, e.g., that the relation to the source code is shown too and that the granularity is more fine-grained, others expected a global history instead of the history for formulas and terms.

Room For Improvement

We explicitly asked for suggestion to improve the functionality, however, also in the course of answering to the other questions about the functionality the participants already expressed room for improvement. We have collected all suggestions and were able to identify that the participants mentioned suggestions for improvement in the areas *presentation/appearance*, *granularity of the mechanism* and *extensions to the mechanism*.

Presentation/Appearance For improving the presentation or the appearance of the functionality, suggestions were mentioned either for the *presentation in the sequent* or *presentation in the proof tree*.

The suggestion mentioned for the *presentation in the sequent* all concerned the color choice or the highlighting. We did not switch off the usual highlighting of the formulas to which a rule has been applied to in KeY. That lead to the effect that in some sequents either a green (from KeY) and a red highlight (from the mechanism) was visible or that the highlighting partly overlapped and therefore resulted in an orange highlighting. This confused some of the participants and they wished for an explanation of the colors. The cases where the green and the red highlighting was visible, have often been due to our choice to include the branching nodes into the view for navigation, although they may have not been subject to change for the mechanism. We chose to use a red highlighting for the mocked prototype, which was disliked by participants.

The participants also mentioned suggestions for improvement for the *presentation in the proof tree*: One was to also show the node directly before a branching node, to reduce the differences between the contents of a node while navigating in the tree. Two suggestions concerned the filtering of the tree: the filtering of the proof tree should first only show nodes that are necessary for comprehension. The solution should provide a possibility to switch the filtering off during usage such that users can inspect the surroundings in more detail. The second suggestion was to only show nodes that contain changes. A further suggestion was made that the selected node during the history view should stay selected when leaving the view. It was also suggested to mark in the proof tree when the selected formula reaches a certain degree of simplicity. One participant recapitulated the opinion from the pre-inquiry and suggested a name change for the mechanism.

Granularity Some participants suggested extending the history mechanism to have a more fine-grained tracking. It was mentioned that the highlighting of the whole sequent in the root is not helpful, and that the subformula that was being tracked should be highlighted. Another suggestion was to track single symbols respectively (sub) terms and not only top-level formulas.

Extensions Also suggestions for extending the mechanism to trace back the origin to the program and specification was mentioned. Another suggestion expressed in the user study was to present a view that shows the original representation of a formula, before the application for normalization rules, for example for quantified formulas.

Discussion

The opinions of the participants about the history mechanism showed that the mechanism is no silver bullet regarding the issues in finding the origin of parts of the proof state. On closer examination the evaluation of the users provides clues for the application areas the mechanism is well suited for: participants found the mechanism to be helpful in proof situations where the origin of a term or formula in a large or "unstructured" proof state was of interest. One specific example that was given concerned proof state parts dealing with ADTs or formulas that where transformed as part of a normalization process (e.g., arithmetic expressions).

Compared to this kind of proof states, participants mentioned that the mechanism is less helpful when it comes to proof state parts that have a more close relation to the proof input artifacts, e.g., for formulas originating directly from annotations.

One conclusion from these observations is that a two-part mechanism would be worthwhile investigating as part of future work: one mechanism to determine the relation of formulas and terms to the annotated program (to also support the user in gaining an overview of the proof situation) and one mechanism for the origin of formulas and terms in unstructured proof states (to support the user when working on a more detailed level).

In addition to these general assessments of the participants, several improvements where mentioned. One concerned the granularity of the highlighting: users wished for that the origin relation was more focused as tracing the origin towards the root of the proof tree results in large parts of the proof state being highlighted with the current mechanism. Regarding the presentation of the mechanism also the coloring of the highlighting could be better integrated into the current highlighting of KeY to provide a clear distinction and better visibility of both types of highlights.

How this new history mechanism is integrated in the existing verification tool affects the usability of the combined system – one positive aspect in this regard that was mentioned by the participants was that the usual way of proof navigation in KeY is still possible such that we believe that we integrated the mechanism into the normal workflow in KeY.

5.7.2. Origin of formulas (Practical Tasks)

We included tasks in the user study where the participants should explain where a specific formula on the sequent of the open goal originates from. First, the participants should solve this task using KeY without any additional support. After introducing that there is new functionality *Show History*, the task was posed again in a new proof situation and the participants were allowed to use the new (mocked) mechanism.[7]

In the following we will summarize our observations. We will focus on presenting the information the participants used to find the origin and the activities they performed. Furthermore, we will discuss peculiarities we observed during the analysis of the sequence models we derived from observing the participants.

In the following we have summarized our observations for the proof states shown to the participants during the user study.

Information accessed during the tasks. For both types of tasks, i.e., with and without using the mechanism, all participants *inspected* all available information in the KeY system on different abstraction level, i.e., the proof tree, including its labels and branching node names, as well as the sequent and its constituents (single formulas, modality and updates). Some participants also used the contents of the text editor. Here, sometimes the intent was expressed that the relation to the program and its annotation is desired.

Activities to find out the origin of a formula. The interactions of the participants can be grouped into different categories. One category is concerned with *navigation* in the proof tree. Here, different sequences were visible. Participants navigated into one direction (either upwards or downwards) through the different nodes. Some sequences contained the navigation through neighboring nodes, others showed jumps between nodes. During the navigation process two variants of inspection actions were observable concerning the sequent: either it was inspected thoroughly or at a quick glance.

Intents that have been expressed during the navigation included that participants retraced a specific formula on the sequent to, for instance, find the (first) position in the proof where it was changed. One particular example of this behavior was that a participant used the mouse pointer as a visual marker of the formula to be tracked when stepping through the different nodes. Another intent of navigating the proof tree and inspecting the sequent was to find out where a particular formula/symbol was *introduced* to the sequent.

It was furthermore visible that some participants used navigation with an *explorative quality*. It seemed that they navigated into one direction until a relevant or interesting information appeared, e.g., a specific formula that is syntactical similar to

[7]An example for a sequence model derived for this task can be found in Section B.3.1 in the Appendix B.

the searched formula drew their attention. It was also observable that participants directly navigated to a particular proof node to select it. Such proof nodes included the root of the proof tree, the open goal and specific branching nodes, such as `Body Preserves Invariant`, or the end of the symbolic execution. We assume the intent that the participants selected a defined entry point for the search, i.e, nodes which they have a clear picture about the proof state (e.g., the root node showing the original proof obligation). Often from this point on the participants inspected the sequent and started navigation activities, as described above.

In case the participant reached a point that is interesting to him, it was observable that the sequent and its neighboring sequents were inspected. On the sequent itself symbols or formulas were searched for (either manually or using the automatic syntactic search provided by KeY). Also the information about the applied taclet was accessed and inspected.

All activities mentioned so far did not change the proof state. Some participants were willing to even perform actions modifying the proof state in order to determine the origin of a formula.

Furthermore, participants said that they either do not want to access the information asked for, or wanted to have a technical support for answering the question. It was mentioned that the proof is too large to manually track the origin. These comments were especially expressed before the possibility to use the mocked mechanism.

Some participants started with the option `Hide Intermediate Proof Steps` turned on from the last task and some used this feature explicitly. We assume that this feature helps to gain an overview over the proof itself. We observed in sporadic cases that the participants retraced the logical argumentation either mentally or with the help of KeY by step-wise navigation of the proof tree.

In the program verification tasks we were able to observe that participants formed the hypothesis about the origin early in the process. Participants moreover tried to relate the observations in the KeY system to the proof input artifacts in the text editor.

Between the activities in the task with and without the mocked mechanism, we were not able to observe significant differences in the activities for the program verification problems. One noticeable difference, however, was that participants did not express their intention to perform state destructive activities in the task where they were allowed to use the mocked mechanism.

For the first-order logic problems we were able to observe differences in the activities. The search for the origin was more targeted and structured, e.g., the repeated alternation between different nodes was in some cases replaced by a more targeted retracing of differences in the proof tree.

Discussion

In the program verification tasks participants used their prior gained knowledge and comprehension about the proof problem to find the origin of the formula. For example, we observed that the participants formed a hypothesis about the origin very early, we assume that the study design and the predecessor task contributed to this behavior. We consider this knowledge to be one reason that we did not observe significant differences in the activities between the program verification tasks with and without the mechanism. In the more abstract first order logic problem the relation to an input problem was not given. It was observable that one participant tried to gain semantic knowledge about the problem by trying to gain type information for the terms for which the origin was asked for. In the more abstract task we were able to see a difference in the activities. The search that seemed unstructured without the mocked mechanism, seemed more structured and targeted when the mocked mechanism was used.

The mechanism only took the syntactic origin into account, without considering the semantic origin, i.e., the contents of the proof input artifacts. For an improvement the semantic origin needs to be considered as well. Participants repeatedly inspected neighboring nodes. We assumed they performed this inspection to see the difference between two nodes and to gain knowledge about the effect of a rule application. The mechanism did not provide a support for this activity however for improvement of the mechanism we devise that this activity needs support as well.

From the observations and the participants responses to the interview questions we conclude that support has to be provided to relate the proof input artifacts to different parts of the proof state. A supporting mechanism such as the one we intended and for which we provided a mock-uped version is a first step towards this direction. However, an alternative to a local improvement such as the history is to consider this relation throughout all activities necessary to prove a program correct. In Chapter 10 we will present an alternative concept which takes the results from this task and the subsequent tasks into account.

5.7.3. Intuition about the Origin of Formulas

As a last practical task in our study we included the task that the participants were shown different formulas and they should point to or state what they consider to be the parents of the formula in focus. Furthermore, they were asked why they consider the chosen formulas as parents. We chose to use formulas which almost all had their origin in the proof input artifacts.

Due to time constraints not all participants where asked to perform this task and not all five subtasks were posed for all participants. Our idea here was to get a first impression whether the participants stop at the root of the proof tree or find the relation in the proof input artifacts. Furthermore, we wanted to see, whether we are able to already get ideas about the granularity for the parent-relation.

In the following we will briefly summarize our observations similar to the preceding task. We were able to observe that participants considered both, the semantic and the syntactic origin. In the first task where the origin was the precondition, participants stopped at the root and considered the root as origin. However, also the precondition was mentioned as origin.

Especially for the origin of the second formula, which originated from the loop invariant annotation the *loop invariant* or the node *body preserves invariant* was mentioned by almost all participants. Some participants also considered specific proof nodes as origin as well.

For the third task participants mentioned intermediate nodes as origin as well as the loop condition. This case was in general more complex, as the formula was moved around the sequent and manually tracking that formula was more difficult. For the fourth task, a few participants mentioned the assignable clause as origin, others mentioned a universal quantified formula or the root as origin.

For some participants we asked how they would like the origin to be presented in the respective situation. One participant mentioned for the origin in the loop invariant and the loop condition that it would be nice to have an indicator, which the user can click on to see the source code location if possible or an indication that the origin is entered manually together with the sequent where the formula was introduced.

5.8. Conclusion and Discussion

In this chapter we have presented the design and conduction of a user study with users of the KeY system. We conducted semi-structured interviews with practical tasks to gain a more detailed insight into the deductive program verification process in KeY. The goal of the user study was to find answers to the questions formulated in Chapter 5.1.1 and to investigate indicators for the gap presented in Chapter 4.1.

We were able to explore important activities in the proof process which include, besides others, the modification of the proof input artifacts, the inspection of the proof state and the proof input artifacts, the selection of proof goals and both the controlled and automatic proof search. During the inspection and modification activities participants switched between a text editor containing the annotated source code and the KeY system, which highlights room for improvement for these activities.

It was also observable that there is not one common proof process for all participants but each participant had an approach that was slightly different from the other observed approaches. The user interface of the KeY system supports this variety in interaction styles by providing users many degrees of freedom to interact with the system in order to find a proof. To determine common interaction patterns and sequences of activities, we created an aggregated sequence model that captures the approaches we have observed during the user study. Moreover, participants may follow a *trial-and-error* approach when trying to construct a proof which was observable in the process. Different proof rules were tried out and the result was observed before reverting the action. Special proof search strategies (or a comparable sequence of manual interactions) were applied to find out which parts of the proof problem may need attention.

For the orientation in the proof process after the application of automatic strategies (the interaction points from our model presented in Chapter 4.1), the participants accessed different parts of the proof state for gaining the orientation (e.g., the proof tree, proof nodes and the sequent). We were not able to identify a common process for orientation from our aggregation of the sequences we observed during the study. However, we were able to obtain first insights into strategies that were performed more often during the study. One finding is that the participants relate the proof state to the proof input artifacts and to achieve this relation switch between accessing parts of the proof state in KeY and the proof input artifacts in the text editor. we also observed that to gain an orientation, participants also searched for specific proof nodes from where they started a more detailed inspection of the proof state. Another important investigation was the strategy of *zooming-in* from an overview over the proof state to single proof nodes and sequents. Moreover, some participants stated the intent of or tried to destruct the proof state and manually perform the proof to gain an orientation in the proof process.

We assume that the switch between the text editor and the KeY system during the inspection activities is an indicator for the gap between the user's model of the proof state and the actual proof state. Another indicator for the gap we believe was the search for specific proof nodes as reference point to start a more detailed inspection. The specific nodes participants searched for were ones where they had an idea about the relation between the proof state and the annotated program.

In the second part of the user study, we have presented a mock-up version for a mechanism that shows the *syntactic origin* of formulas during the proof. We investigated in the user study whether our first idea for a realization of such a mechanism suits the users expectations and needs and whether it helps to bridge the gap between the model of the proof of the user and the actual proof situation. The participants expressed such a mechanism may be well suited for proof situations where the origin of formulas in a large or "unstructured" proof state was of interest. We learned that the mechanism itself can be further improved, to be suitable for proof state parts that have a close relation to the proof input artifacts, by adding the *semantical* origin information, i.e., the relation between terms in the proof state and their corresponding parts in the annotation. Also further improvements were mentioned by the participants, e.g., that the granularity shown by the mechanism is not always sufficient.

The participants also expressed general room for improvement for KeY and for the general user interaction, e.g., concerning the presentation of information in the proof state and proof nodes. Also the granularity of the interaction steps where mentioned: in addition to single rule applications, users would like to be able to use more coarse-grained proof steps similar to proof macros. For some of the improvements it is necessary to investigate the user's needs further before being able to develop an improvement.

Concerning techniques we used to conduct the user study, the *Thinking-Aloud* method we have used to gain insight into the intents of the participants has advantages and disadvantages. An advantage is that the thinking-aloud technique does not need any special setup and it is rather effortless in applying this method. One disadvantage is that users may find themselves in an unnatural situation and try to fulfill expectations. Also, participants do not express their intents for every action they perform and thus intents often have to be deduced after the interview.

The conducted study was a qualitative study in order to explore opinions, issues and evidences for our hypothesis. However, in order to generalize the results a quantitative evaluation has to be carried out with an appropriate number of participants. Our results support that the problems and issues that are addressed in this thesis are ones that users of the KeY system consider needing improvement.

One example for this is that after the study has taken place, a view was added to the KeY system that contains the source code and a highlighting of the symbolically executed lines. Furthermore, the view was being developed to be interactive, such that users can click onto a highlighted line and the proof node that was responsible for the symbolic execution of the selected statement is selected. So the user can use this view to navigate in the proof as well. Furthermore, this view can be shown at all times, such that the support for finding the relation between the source code and the proof state is improved.

From our observations in the user studies, we derived the hypothesis that users of interactive program verification systems need both an overview over the system and the bigger picture of the proof task and a way to focus on specific parts of the proof problem. Furthermore, the switch between the views should be seamless and users should always have the possibility to relate objects in one view to objects in another view, e.g., the annotations in the annotated program to formulas in the proof state. In Chapter 10 we present a concept that takes the aforementioned hypothesis into account.

6. Summary and Conclusion

In the following we will briefly summarize the results of our two user studies presented in this part of the thesis and draw conclusions for the remaining part of this thesis. Related work about the evaluation of the usability of interactive theorem provers will complete this chapter.

6.1. Summary and Conclusion

In this part of the thesis we have presented two qualitative user studies we have performed to investigate the context of use of interactive program verification systems. We have chosen evaluation methods suitable for the relatively small user base of interactive program verification systems.

We have chosen to use two interactive proof systems with significantly different user interaction styles for proof construction as target of evaluation for the first user study using *focus groups*. We performed a focus group discussion for the KeY system, as an example for a verification tool which allows users to inspect all parts of a proof state and construct proofs using direct manipulation interaction. As second target of evaluation for a focus groups discussion we chose to use Isabelle, an interactive theorem prover that allows for text-based interaction using either an unstructured proof language style or a structured proof language.

As a second user study we conducted *semi-structured interviews* with users of the KeY system to investigate specific aspects of the system's usage in more detail. We posed questions about the proof process in KeY and observed the participants while they were using the KeY system to perform parts of the proof process.

The results of the focus group discussions provided evidence for the gap between the user's model of the proof and the current proof state in the systems. Participants mentioned strengths and weaknesses of the evaluated systems related to the gap. For instance, aspects that help to bridge the gap were mentioned: tools that help to understand the proof state, the intuitive structure of Isar proofs or that the tools provide counterexamples. Moreover, aspects that help to keep the gap small, e.g., an intuitive proof presentation, modularization techniques for the proof and proof task, as well as understandable automatic strategies were mentioned. Participants also commented on aspects about the interaction, for example, they wished for interaction on annotation level (in KeY), support for finding the correct tactic (in Isabelle) or that repetitive interactions in similar situations should be avoided.

We explored the feedback the systems give during the proof process, the granularity of proof steps and asked for time-consuming actions in the proof processes of both systems. We learned that the system feedback in both systems is given through open or closed goals and counterexamples. However, users are able to inspect parts of the proof state which is significantly different in both systems. While KeY users can explore the proof tree with all intermediate steps, in addition to the open goals, in Isabelle, only goal states and the structured Isar proof can be viewed. Isabelle, however, provides tools that give additional insight into the proof state, for example the tools try or nitpick.

The proof step granularity was viewed as being too coarse in some situations (in Isabelle) or too fine-grained in some situations in KeY. We conclude from this observation that users may need intermediate steps, but it would be supportive to retrieve them on demand. In Chapter 8 we have incorporated this observation into our interaction concept and provide a way that users can inspect details of the proof on demand, by using stepping functionalities similar to software debugging systems.

As time consuming actions participants drew the attention to, for example, understanding the proof state, as well as performing trivial repetitive interactions and redoing a proofs and to backtrack in the proof. Moreover, in Isabelle cleaning up the proofs for publishing was mentioned to be time consuming.

As conclusion from the focus groups we were able to gain first insights into time consuming actions in the proof process and were able to find arguments that support our hypothesis about the gap presented in Section 4.1. The focus groups also revealed that it may be promising to combine text-based interaction with direct manipulation in order to allow for more effective interactions and more flexibility.

With the insights from the focus groups we conducted a user study with KeY users where we aimed to explore in more detail the proof process by using semi-structured interviews with practical tasks. Also in this user study we were able to find evidence for the gap. Participants navigated and searched in the proof tree to retrace the constructed proof when trying to gain the orientation as well as searched for the relation between the proof state and the proof input artifacts. The user study revealed that providing means to explore the relation between different proof artifacts is essential for a user interface for deductive program verification systems.

With our aggregated proof process as a result from the user study we were able to observe that there is not one single common proof and orientation process but each user has their own way of constructing proofs and trying to gain orientation in the proof. However, for proof construction we were able to identify common activities, such as controlled or automatic proof search, inspection of the proof state or selection of proof goals or branches. We were also able to observe two prominent sequences of activities that especially have to be included into an interaction concept: the sequence where users tried to proof a proof problem using the automatic strategies and the loop where users perform a controlled proof search. Furthermore, a seamless switch between both “interaction loops” has to be supported.

In general, we were able to observe many degrees of freedom for interacting with KeY to gain orientation and to construct proofs. For an effective and efficient interaction concept the right balance between providing many degrees of freedom and providing user guidance has to be found. The users should be guided such that the the comprehension about the proof state is always present and the users do not loose the relation to the overview over the proof, respectively the general proof plan. In Chapter 10, we will present an interaction concept that aims to achieve this goal.

In the next part of this thesis we will present a combination of direct manipulation with script-based interaction for deductive interactive program verification systems and a prototypical realization for the KeY system. We furthermore present a concept for proof exploration that enables users to stay in the context of a proof attempt and explore whether possible changes of the proof input artifacts would lead to a completed proof, instead of directly changing the proof input artifacts and restarting the full proof process. To the end of the next part we will present an interaction concept for seamless deductive program verification that takes the findings of both users studies into account and allows for direct manipulation interaction as well as text-based interaction, in the form of annotation based and script-based user guidance.

6.2. Related Work

The usability of interactive theorem provers has been evaluated using various evaluation methods. Related work is concerned with usability evaluations of interactive theorem provers based on models defined prior to the evaluations. In addition, related work is also concerned with the derivation of models of the interactive proof process from evaluation results.

In previous work [BG12], we have performed a questionnaire-based evaluation of the KeY system based on Green and Petre's Cognitive Dimensions questionnaire [BG07] to get a first impression of the user's perception and to develop first hypotheses about the usability of the KeY system. Beyond that, Kadoda, Stone, and Diaper [KSD96] evaluated proof systems using Green and Petre's Cognitive Dimensions questionnaire to develop a list of desirable features for educational theorem provers.

Aitken et al. [Ait+98; AM00; Ait+95] evaluated the interactive proof systems Isabelle and HOL using recordings of user interactions with the systems in collaboration with HCI experts. During the proof process the users were asked to think aloud and after the recordings the users were interviewed. The goal of this work was to study the activities performed by users of interactive provers during the proof process to obtain an interaction model of the users. Aitken et al. propose to use typical user errors as usability metric and they compared provers w.r.t. these errors. Also, suggestions for improvements of the systems have been proposed by the authors based on the evaluation results, including, besides others, improved search mechanisms and improved access to certain proof-relevant components.

Jackson, Ireland, and Reid used co-operative evaluation methods on the CLAM Proof Planner [JIR99]. Users were asked to perform predefined tasks while using the "think-aloud technique" to comment on what they were doing.

Vujosevic and Eleftherakis used questionnaires and interviews to explore why formal methods tools are not used in industry [VE06]. Their work includes evaluations of usability aspects of several formal methods tools, such as the Alloy Analyzer. For improving the interface of the prover NuPRL, a self-designed questionnaire was used to evaluate the users' perceptions of the interface [Che01].

Similar to our findings, Archer and Heitmeyer [AH97] also realized the gap between the prover's and the user's model of the proof. They have developed the TAME interface on top of the prover PVS to reduce the distance between manual proofs and proofs by automation. TAME is able to prove properties of timed automata using so called *human-style reasoning*. Proof steps in TAME are intended to be close to the large proof steps performed in manual proofs. The authors have developed strategies on top of the PVS strategies that match more closely the steps performed by humans. The goal is to provide evidence and comprehension of proofs for domain but not proof experts.

Lowe et al. describe in their work [Low+96] their approach to building a co-operative theorem prover and describe some undesirable features of ITPs focussing on feedback of the system. They have implemented the BARNACLE interface for the CLAM prover which allows explanations for failing preconditions, which should make proofs more comprehensible for the users.

Ouimet and Lundqvist [OL07] identified different issues, e.g., large proof size and number of proof steps, that have to be addressed in order to have a widespread use of theorem provers in and evaluated the system ESC/Java against these issues. The issues were identified by examining a large case study conducted at Motorola.

Hentschel, Hähnle, and Bubel [HHB16] have empirically evaluated two different user interfaces for the KeY system, the traditional interface and an interface similar to a software debugger with focus on the proof input artifacts. The focus of the user study was on comparing the interfaces regarding their support in understanding the proof state. Their results provided statistical evidence that the debugger-like interface is more effective than the regular interface of KeY. Their result is similar to our observation concerning the gap, where users searched for the relation between the proof state and annotated program.

Merriam and Harrison [MH96] and Merriam and Harrison [MH98] have evaluated interfaces of three theorem provers: CADiZ, IMPS and PVS. In this work they have identified four key activities in the interactive proof process where the user needs support from the proof system: planning, reuse, reflection and articulation. The three theorem provers have been examined with respect to these activities. Based on these results, gaps in user support of the theorem provers have been identified as well as points in the systems' interfaces where the user can make errors that cost him or her a lot of time to recover from. Their activities are similar, but a bit more abstract compared to the activities we were able to identify. Our activities are more fine-grained and helped to capture more details of the proof process. Our activities however, can be categorized into the activities found by Merriam and Harrison.

Merriam [Mer96] developed two approaches for the description of user activities in the proof process. He formalized a generic formal model of the proof using Z as formal language. This model is used to enable to gain insight into which kind of information is necessary for the user to conduct a proof effectively. Merriam assumes in this model that the user forms an opinion during the proof process about the provability of a proof goal using heuristics. He remarks that to model this assumption, a suitable cognitive model of the user is necessary. Interactions the user performs in the system are outside this model and are modelled in a second model of Merriam on the basis of Newman's Action cycle. Both models together were used to evaluate the PVS proof system.

Völker [Völ03] published a discussion paper on requirements and design issues of user interfaces for provers. He presented difficulties in the design of user interfaces of theorem provers developed in academia. In addition, a requirement analysis based on the scenarios using the scenario method has been carried out and resulted in a high-level description of the interaction with the proof system.

The systems Isabelle and HOL have been evaluated by Aitken [Ait96] using records of interactions. A semi-formal interaction model was extracted from the results, by identifying the actions that were performed during proof construction. Of the fifteen actions that have been identified, some relate to mental work of the users and some were direct actions in the system. All actions were modelled as activity diagram and it was distinguished between actions on the logical level and actions on the interaction level. In this work the relation between the problem class, the proof plan and the implementation is depicted.

In the work of Goguen [Gog99] three user roles that can be represented by one single user have been identified: the prover, the reader and the specifier. Each of these roles has different requirements for the interactive proof system and some of the requirements can be conflicting. The authors claim that users of theorem provers need precise feedback on the failure of a proof attempt at the (sub)goal level. Further they argue that an unstructured proof tree is not easy to use as the users need to orient themselves in the proof tree. They present a proof approach where users should form the high-level proof plan and leave the "low-level computations" to the automatic prover. They implement their user interface for the proof assistance tool Kumo.

Part III.

Integration of Direct Manipulation and Script-Based Interaction Styles

Introduction

To support user interaction in interactive program verification we present our approach to enhance a direct manipulation interface with a text-based interaction style. To achieve this combination we introduce a language concept for a flexible and concise proof scripting language tailored to the needs of program verification in Chapter 7. A realization of this concept as the *KeY proof scripting language (*KPS*)* is presented. To combine the two interaction styles – text-based interaction and direct manipulation – we leverage an analogy between software debugging and analyzing failed proof attempts in Chapter 8.1 and additionally adhere to usability principles identified for general interactive theorem provers. The realization of our interaction concept is introduced in Chapter 8, where the *Proof Script Debugger (*PSDBG*)* is introduced as an interface for the KeY system allowing to use both interaction styles in the proof process.

To allow for more purposeful changes of the proof input artifacts, in contrast to a more trial-and-error approach, we will introduce the concept of proof exploration in Chapter 9.

We will conclude this part of the thesis with a seamless interaction concept in Chapter 10 for program verification that that takes the insights from our exploration of the context of use into account and integrates the three interactions styles for proof construction, i.e., direct manipulation, script-based and auto active. Insights from the development of the scripting language and the proof script debugger guided the interaction concept as well.

7. A Domain-Specific Language for Interactive Program Verification

Contents

7.1. Introduction

Program verification proofs have characteristics that are considerably different from proofs of mathematical theorems (e.g., properties of algebraic structures). In particular, they consist of many structurally and/or semantically similar cases which are syntactically large, but usually of less intrinsic complexity. The mechanism for providing user guidance should reflect this peculiarity of proofs in the program verification domain and provide appropriate means for interaction.

In this chapter we present a concept for a proof scripting language tailored to the needs of program verification. We contribute with a concept for a concise and flexible proof scripting language which allows the user to formulate proof statements which are applied to a group of syntactically or semantically similar subproblems. The core of the language concept is a *multi-matching* mechanism to define selection criteria for choosing several goals at a time that can then be treated uniformly.

These selection criteria are resilient to changes in the sense that small changes in the proof require small changes in the corresponding proof script. This reduces the amount of changes necessary between iterations in the incremental proof process.

As introduced in Ch. 3.6, two main interaction styles have emerged in state-of-the-art interactive verification systems: text-based interaction (proof scripts and source code annotations) and direct manipulation interaction (application of calculus rules to terms selected in a graphical user interface as in KeY). Compared to scripting languages where single proof statements apply to only one goal, and to a textual recording of pure direct manipulation interactions, a scripting language with multi-matching allows creating more compact proof scripts.

We showcase our concept, which is particularly well suited for verification systems with explicit proof objects using a sequent calculus, by applying the concept to the interactive program verifier KeY [Ahr+16].

Outline. In the following section, we start by discussing the proof characteristics of interactive program verification. Then, we introduce the concepts for a proof scripting language tailored to the peculiarities of proofs in this domain in Section 7.3 and we present the realization as a proof scripting language for the KeY system. The syntax and semantics of KPS are introduced in Section 7.6.

7.2. Characteristics of Program Verification Proofs

Program verification proofs differ from mechanized proofs of mathematical theorems, particularly in the size and complexity of the occurring formulas and in the number of different cases to investigate. One reason for this is that the proof structure mirrors the program structure and so program verification proofs often have a large number of individual subgoals reflecting the control-flow possibilities in the program.

Each subgoal represents in parts the effect of a possible program execution path, and subgoals for similar paths often have a high degree of similarity since they share common path- and postconditions. Such related subgoals may be treated uniformly, using a common proof strategy. Based on experience with the KeY system, during proof construction, the user switches between focusing on one particular proof goal and looking at a number of proof branches for example to decide which ones are semantically similar.

With increasing complexity of programs and specifications, users normally develop proofs in an iterative and explorative manner, as subtleties of the proofs are often only discovered after an attempt fails. These iterations include modifying the specification or the program, as well as adding information to guide the proof search. Until the verification succeeds, (a) failed attempts have to be inspected in order to understand the cause of failure and (b) the next step in the proof process has to be chosen and applied.

Both (a) and (b) are complex tasks. One reason is the inherent difficulty of understanding a mechanised, formal proof for a non-trivial program property. In addition, proofs generated by verification systems are of fine granularity. This makes it difficult for users to understand the *big picture* of a proof – the abstract argumentation for why the program fulfils its specification. To succeed with subtask (b), performing the next proof step, the user has to understand the nature of why the proof failed:

Is it a mismatch between specification and program or is the guidance for the proof system insufficient? This was also visible in the user study (see Section 5) where the participants compared the proof state of the KeY system with the specified program and tried to find the correspondence between them.

State-of-the-art tools support the user in both tasks by, e.g., providing counterexamples and means to inspect the (incomplete or failed) proof object. However, performing the proof process is still characterized by trial-and-error phases, where users try out different proof search strategies and if they did not lead to a closed proof revoke the application of the respective strategies.

We claim that support for *debugging* large proofs is needed, providing means for explicating the correspondence between parts of the proof and parts of the program and its specification, for automating repetitive tasks and applying them to a number of uniform proof goals, and for analysing failed proof attempts. The interaction has to use a suitable level of granularity. Most existing verification tools with explicit proof object – i.e., a concrete proof object consisting of atomic rule applications, – only support the most detailed granularity, whereas systems using proof scripts – i.e., the proof object is implicitly known to exist but not actually constructed, – support interaction on a more abstract level and also allow repetition of proof steps (but mostly, repetition can only be applied to single or to all proof goals, but not to matching subsets).

7.3. Concept for a Proof Scripting Language

To improve on state-of-the-art in user interaction support for interactive program verification systems, we present a concept for a proof scripting language that takes the peculiarities of program verification proofs into account. We consider a rule-based program verification system operating on program logics. Additionally, we consider the verification system to present an explicit proof object to the user and natively target direct manipulation as primary interaction style for proof construction. For this verification system we aim to add text-based interaction, such that it is possible for the user to construct proofs using both interaction paradigms interchangeably.

The basic principles of the language are introduced in the following.

Requirements for the Language. The characteristics of proofs for program verification as described in the previous section lead to the following important elements of our concept for a proof scripting language:

1. integration of domain specific entities like *goal, formula, term* and *rule* as first-class citizens into the language;
2. an expressive proof goal selection mechanism that allows the user to
   - identify and select *individual* proof branches,
   - easily switch between proof branches,
   - select *multiple* branches for uniform treatment (*multi-matching*);

   and that is resilient to small changes in the proof;

3. a repetition construct which allows repeated application of proof commands (i.e., calculus rules or proof strategies);

4. support for proof exploration within the language.

The objects manipulated during proof construction are called *proof goals*. We assume that each (open) proof goal is unique and identifiable by its contents (e.g., its sequent together with the position in the proof tree, when using a sequent calculus).

Applying calculus rules or proof strategies to a proof goal can have two results: either the goal is replaced by newly created goals or the goal is removed from the proof in case the strategy closed the proof goal.

Performing proof construction is characterized by explorative phases in which the user tries to determine the best way to approach the remaining proof tasks. One example for this is when the user suspects that a fact is derivable but is not certain. In the user study, we could observe the cases where participants performed mental deduction steps while analyzing the sequent of an open goal. We could also observe that participants selected formulas in the open goal to see which rules are applicable in the respective proof situation. In such cases, the user may try different proof strategies or different lightweight techniques (such as bounded model-checkers to find counterexamples). In our user study, we were able to observe that a participant wanted to generate a counter example and in case this activity would not be successful use an SMT solver or generate a test case to analyze how a specific case is handled in the Java program. These exploration activities have to be considered for the design of a proof scripting language – for example by supporting (hypothetical) queries to the underlying proof system or other reasoning systems without disturbing the current proof state.

In the following, preliminaries are introduced, followed by a description of our concept for a proof scripting language taking the aforementioned requirements as well as insights from the user studies into account. The language concept is introduced using an abstract syntax and an informal description of the language semantics. The concrete semantics are then presented when introducing our realization of the language concept for the KeY system.

Many definitions introduced in this chapter will be instantiated for the KeY system but can be easily generalized to other program verification tools, if not indicated otherwise.

7.4. Prerequisites For the Proof Scripting Language

The script language supports local, goal-bound variables of types boolean, integer, and of domain-specific types such as goal, formula and term. Variables and constants are the most basic expressions of the language which can be combined to compound expressions using arithmetic operators, boolean connectives, subterm selectors, and substitution expressions for concrete and schematic terms and formulas. Evaluations of expressions and assignments to variables are defined as usual for simple expressions[1].

We distinguish between two kinds of states for the evaluation of a proof script: (a) proof states of the verification system characterised by the set of open proof goals and (b) script proof states, which in addition to a proof state contain the value of state variables that are local for each open proof goal.

There are three cases in which the evaluation of a script terminates: (1) there are no further statements to execute (the end of the script is reached), (2) an error state is reached, or (3) the set of remaining open proof goals is empty.

In a *proof* a transition from proof state to another is performed by applying *mutators*, which can be for example calculus rule applications or calls to decision procedures.

Proof State. In the following, we will consider a proof state to be a proof state of a sequent calculus where the state is a (partial) *proof tree* as defined in Def. 3.4.1 which is based on [Ahr+16]. However, the concept for the proof scripting language is also applicable to other rule-based calculi.

For the self-containment of this chapter, we will recall the definition of a proof tree from Sec. 3.4 and the corresponding terminology:

A proof tree is a tree such that

1. each node is labeled with a sequent
2. leaves are either labeled with a sequent or the symbol $*$, marking a closed leaf.
3. if an inner node n is annotated with $\Gamma \Longrightarrow \Delta$ then there is an instance of a rule whose conclusion is $\Gamma \Longrightarrow \Delta$ and the child nodes of n are labeled with the premise or premisses of the rule instance.

A branch in the proof tree is called closed if its leaf is labeled with $*$. A proof tree is called closed if all its branches are closed, or equivalently if all its leaves are labeled with $*$. All other leaves are called *open.* A proof tree is partial, if not all leaves are closed. We will call the open leaves of the proof tree *proof goals.*

The root's sequent is called the *(original) proof obligation.*

Script Proof State. On top of the proof states as given by the verification system, we introduce a *script proof state.* Each script proof state contains exactly one proof state, represented by the set of open proof goals of the proof state (as depicted in Fig. 7.1).

The object that is being manipulated during proof construction is a single *open proof goal.* In a script proof state at most one proof goal of the set of open proof goals is selected. The proof goals are manipulated independent from each other. The essential part of a script proof goal is the open verification condition of the contained proof goal. In case of a proof in the sequent calculus an open verification condition is the sequent of an open goal. In addition, each proof goal can contain variables. These variables are *goal-local*, i.e., changing the value of a variable has only a local effect. All variable assignments of siblings of this goal in the state remain unchanged. When

[1]The language also contains special *match expressions*, which are evaluated with a side effect that results in binding matched terms to variables (see Ch. 7.7.1)

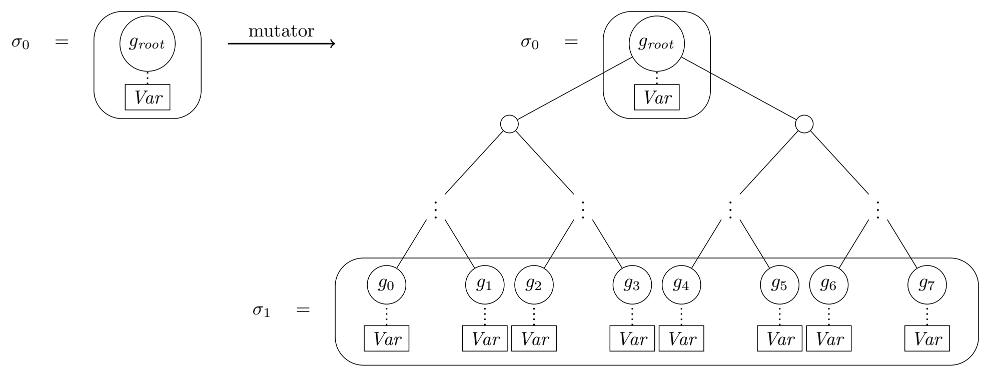


Figure 7.1.: Relationship between script proof state and proof state. The script proof states are depicted with a rectangle around them (σ_0, σ_1). The proof state is the partial proof tree. The transition from σ_0 to σ_1 is invoked by a *mutator*.

a new goal is created, it inherits its parent goal's assignments. To model the goal-local variable assignments in a script proof state we use a function that returns the variable-assignment function for a particular given proof goal. The returned function is defined over the set of typed variables *Var* with a typing function δ, assigning each variable $v \in \mathit{Var}$ its type $t \in T$ in goal $g \in \mathcal{G}$, and over the typed set of variable values with typing function $\delta_{\mathit{Val}} : \mathcal{D} \mapsto \tau$, mapping the domain values to its types.

Definition 7.4.1 (Script Proof State)**.** *Given a partial proof $\mathcal{P}$ and a proof script π, let $\mathcal{G}$ be the set of open proof goals of $\mathcal{P}$. Furthermore, let Var be a set of variables typed according to a given typing function δ and Val be a set of variable values. A script proof state $\sigma = (g, \mathcal{G}, \mathcal{V})$ is a triple with*

- *$g \in \mathcal{G} \cup \{\bot\}$ the selected goal, where $\bot$ denotes that no goal is selected, and*
- *$\mathcal{V}\colon \mathcal{G} \to \mathit{Var} \to \mathit{Val}$, the well-typed (w.r.t. δ) variable assignments.*

Whether the variables of a goal $\mathcal{V}(g_i)$ in a script proof state are well-typed is dependent on the proof goal g_i. For example, a variable `x` may have different types depending on in which proof goal the variable is evaluated.

7.5. Script Language Constructs

The three main building blocks of the scripting language are *mutators*, *control-flow* structures, and *selectors* for proof goals. We describe the general concepts in the following. The syntax can be found in Figure 7.2.

Mutators. Mutators (M in Fig. 7.2) are the most basic building blocks of the proof script. When executed a mutator may change the proof script state by changing variables and the underlying proof state by adding nodes to the underlying proof tree.

$$
\begin{aligned}
M &::= (\mathit{script\_name} \mid \mathit{native\_command})\ \mathit{args} \\
C &::= C_1; C_2 \mid \mathit{var}\ :=\ \mathit{expression} \mid \texttt{repeat}\ \{C\} \mid \texttt{foreach}\ \{C\} \mid \texttt{theonly}\ \{C\} \\
&\quad \mid \texttt{cases}\ \{\texttt{case}\ S_1 : \{C_1\}\ \ldots\ \texttt{case}\ S_n : \{C_n\}\ \} \mid S \\
S &::= \mathit{expression} \mid \texttt{matchSeq}\ \mathit{schemaSeq} \mid \texttt{closes}\ \{C\} \mid \\
&\quad \mid \texttt{matchLabel}\ \mathit{regexp} \mid \texttt{matchRule}\ \mathit{rulename}
\end{aligned}
$$

Figure 7.2.: Abstract syntax of the proof scripting language.

Proof commands that correspond to calculus rule applications or strategy applications are called *native*, as their implementation is not written in the proof scripting language. Besides variable assignments and (native) proof commands, a third type of mutator is the call of other scripts.

Example 7.1. Consider the following example of the calculus rule *applyEq* of the KeY system, which is modeled as a mutator in the proof script. The calculus rule *applyEq* applies an equality contained in the antecedent of a sequent to a formula in the succedent containing the left-hand side of the equality.

$$
\mathit{applyEq}\ \frac{\Gamma, x = t \Rightarrow f(t), \Delta}{\Gamma, x = t \Rightarrow f(x), \Delta}
$$

The mutator

$$
\texttt{applyEq on=}\ \overbrace{\texttt{'==> x=y'}}^{\mathit{mutation\ target}}\ \texttt{with=}\ \overbrace{\texttt{'y=1 ==>'}}^{\mathit{side\ condition}}
$$

in KeY has the semantics that an equality `y=1` occurring in the antecedent (the part to the left of `==>` in the goal) is to be applied to the formula `x=y` in the succedent (the part right of `==>`), replacing `x=y` with the formula `x=1`. In the following sample sequent this state transition is depicted:

$$
\overbrace{\texttt{x=1, y=1 ==> x=y}}^{\text{goal before }\texttt{applyEq}} \quad \leadsto \quad \overbrace{\texttt{x=1, y=1 ==> x=1}}^{\text{goal after }\texttt{applyEq}}
$$

If either of the formulas `y=1` and `x=y` is not present in the goal, this mutator is not applicable. In this case the script's execution results in an error state.

The semantics for both mutator types is similar: they change the set of open proof goals of the proof state. However, in our concept, native proof commands are only applicable to a single goal. If the goal set of a proof state consists of more than one goal, we define the execution of a command to result in an error state to avoid the ambiguity to which of the goals the command should be applied to.

Some native proof commands may run indefinitely (e.g., strategies for the automatic proof search). For these commands, proof systems often allow the user to limit the execution time, e.g., by specifying the maximal number of proof steps or a timeout. For these native proof commands that can be parameterized as just described, our script language allows the user to provide these parameters as arguments for the invocation of native proof commands.

Control Flow. Besides sequential composition and variable assignment, the language supports control structures (C in Fig. 7.2) targeting command application to one or more proof goals. To be able to apply proof commands to a single proof goal repeatedly, we include a `repeat` statement. The semantics of the statement is that the command following `repeat` is applied until it does not modify the proof state anymore.

Example 7.2. Consider the following example script for KeY containing a repeat command:

```
repeat { andLeft; }
```

The non-splitting calculus rule `andLeft` deconstructs a binary conjunction in the antecedent of a sequent into two conjuncts.

$$\mathit{andLeft}\ \frac{\Gamma, A, B \Rightarrow \Delta}{\Gamma, A \wedge B \Rightarrow \Delta}$$

As long as the non-splitting rule `andLeft` is applicable in a sequent, it is applied. This is a typical situation for the verification tasks in the KeY system where the original proof obligation contains a conjunction of formulas resulting from the method's preconditions.
After applying this script to the sequent `A & (B & C) ==> D & E`, we get the new sequent `A, B, C ==> D & E`. The rule `andLeft` does not have arguments, therefore the underlying verification system needs to find the right formula to apply the rule to. In case there is more than one formula that the rule can be applied to, an argument indicating the right formula is needed. Note that, by its definition in KeY, the rule `andLeft` is only applied to the conjunctions in the antecedent.

Selectors. During proof construction, the application of calculus rules can cause a proof goal to split into different cases. In our concept, we have chosen that proof commands only operate on single goals to avoid user confusion. Therefore, our concept for the proof scripting language provides the possibility to specify which goal to operate on using a mechanism called *selectors* which allow to specify the proof goal based on information of the proof state (e.g., the syntactical structure of the proof goal).

Generally, when there is more than one open goal in a proof, there are different choices on how to apply the next script statement:

(a) apply the next script statement to a single proof goal in a state that is determined by a chosen strategy (e.g., the first goal or the last goal),
(b) apply the next statement to multiple/all proof goals in a state, or
(c) stop the application and end in an error state.

For our concept we have chosen option (c), to stop the application, if a proof command is invoked in a state with more than one open goal. We chose this option to avoid confusion by the user concerning the selected goal, if the user has a wrong idea about the order of goals (for option (a)) and to avoid that the user has to determine to which goal the command was applied, because it was applicable (option (b)).

For option (c) we then need a possibility for the user to indicate to which proof goals a proof command is to be applied. Selectors (S in Fig. 7.2), a flexible mechanism to select one or more proof goals, can be used for this purpose. We included the *cases*-command into our concept as selector. It is tailored to the needs of proving in the domain of program verification, allowing the formulation of proof goal sets using *matching conditions* (an example for a matching condition can be found in Example 7.3, where `match '==> ?A -> ?B'` is a matching expression). Matching conditions are expressions evaluated for each proof goal; all goals which satisfy a matching condition S_i are subject to the corresponding proof command C_i. Thus uniform treatment for several goals can be realized. If a proof goal satisfies more than one matching condition, the first match is chosen. The application of a `cases` command results in a script state consisting of the union of all open goals of each case, after the corresponding commands are executed.

In our language concept, we support three fundamentally different types of matching conditions: *State conditions* consist of an expression over the script variables (for example `?X>0`, to express that the value of the script variable `?X` has to be greater than 0). Script evaluation selects those proof goals in which the specified expression evaluates to true. *Syntactical conditions* (keyword `matchSeq`) allow the specification of a logical sequent with schematic placeholders. The condition satisfies those proof goals for which the schematic sequent can be unified with the proof goal's sequent. *Semantic conditions* (written as `closes {C}`) involve the deductive capacities of the verification system to decide the selection of proof goals. A proof goal is selected if and only if the application of the proof command C would close this goal.

Syntactic matching is not limited to the goal's sequent (using`matchSeq`) but can also be applied to rule names (using `matchRule`) and to labels put on the branches of a rule application (using `matchLabel`).

In addition to the `cases` command, `foreach {C}` and `theonly {C}` are included for convenience purposes. Both apply command `C` to each goal in the state and are semantically equivalent to `cases { case true: {C} }`. Command `theonly`, in comparison also passes a warning to the user, if there is more than one goal when the command is evaluated. The idea behind this command is that the user can use this command in situations where the user expects that there is exactly one goal in the proof state, and be notified if this expectation is not met.

Schematic placeholders used for syntactic goal matching have names that start with '`?`'. When they are instantiated while matching against the sequent of a proof goal, these instantiations can be accessed also in the embedded proof command (e.g., as argument for a calculus rule) to direct the proof using information present on the sequent. If there is more than one possibility for instantiating the schema variables during constraint solving, the first match is used.

Example 7.3. Consider the following simple example for the use of a matching condition within a `cases` selector, where the schematic sequent matches sequents containing a top-level implication in the succedent:

$$\texttt{case match '==> ?A -> ?B' : }\overbrace{\texttt{\{ impRight; andLeft on='?A';\}}}^{C}$$

In case of a match, the left side of the implication is assigned to the variable `?A` and the right side is assigned to `?B`. Then, the proof command C is executed. After applying the rule `impRight`, the rule `andLeft` is applied to the formula bound to `?A`. This example reveals a requirement for the underlying verification system: it needs to check whether the formula bound to `?A` is still on the sequent when applying the rule `andLeft`. If there is more than one occurrence on the sequent (e.g., if the term or formula bound by `?A` would occur within different formulas), one of them is chosen for rule application. If the formula is not present anymore (because other rules have been applied before) the rule is not applicable, which results in an error state.

Proof Exploration. To support proof exploration in the scripting language, we include the expression "`closes` { C }" in our concept. It examines whether applying the proof command C would close the current goal (without actually affecting the current state). We have implemented the `closes` expression by performing the actual proof and discarding it afterwards. A `closes` expression can only be used in the cases statement as matching condition.

Example 7.4. Assume that a proof command is (only) to be applied to those goals, which can be closed once some formula F is added to the succedent of the goal's sequent (i.e., the formula F is derivable from the sequent).
This may for example be expressed by an explorative expression such as using `closes` in a `cases` statement as follows: `closes {assume '==> F'; auto}: { C; }`, where `assume '==> F'` is a proof command adding `F` to the succedent. However, such a proof command has to be offered by the underlying verification system and adding arbitrary formulas to the proof obligation during proof construction is unsound in general. Thus, the `assume` command should only allowed in a `closes` expression. The proof command `auto` is then used to try to prove the newly created proof obligation. If the newly created proof obligation closes, the proof commands that follow the `closes` case, are applied to the proof goal before the exploration step.

Exploration expressions that check whether a certain formula is derivable (as shown in the example above), come in handy when we want to match a formula, such as `x > 0`, but on the sequent a stronger formula, such as `x > 1`, is present. While `case match 'x > 0'` would miss the goal node, an expression checking for derivability of `x > 0` would match the sequent.

Another possibility for such a proof exploration expression is to include a special match expression `case derivable 'F':` that has a similar effect. The command opens a new side proof of the proof goal to prove whether the formula `F` can be derived using the prover's capabilities and if it is the case the proof goal is selected for further use adding the derived formula to the proof obligation. In the following sections we will present an instantiation of the language concept for a concrete proof scripting language where we have chosen to include such a proof exploration expression.

7.6. An Instantiation of the Language Concept for a Proof Scripting Language for KeY

In the following section we will introduce an instantiation of our language concept for the KeY system leading to the scripting language KPS. As a running example for a script in KPS, we use a script constructing a proof for the correctness of the pivotal split in a Quicksort implementation (see Fig. 7.5).[2]

Example 7.5. This proof script can be used for proving the correctness of the split method of Quicksort implementation that is shipped as a standard example with the KeY system.

```
1  script quicksort_split() {
2    autopilot_prep;       //perform symbolic execution and simplify
3    foreach { tryclose; } //try to close all trivial cases
4    foreach { simp_upd;   seqPermFromSwap;  andRight; }
5    cases {
6      case match `==> seqDef(?,?,?) = seqDef(?,?,?)`: auto;
7      case match `==> (\exists ?X (\exists ?Y ?))` :
8          instantiate  var=X with=`i_0`;
9          instantiate  var=Y with=`j_0`;
10         auto;
11   } }
```

In the first lines (2 – 3) a pre-processing of the proof state is performed. After application of simplification steps and the application of a rule specific for the data type sequence (`seqPermFromSwap`), user guidance in the form of quantifier instantiations is required (lines 8–9). The match expression in line 7 matches sequents that contain a formula which consists of at least two nested existential quantifiers and binds the concrete terms of the quantified variables

[2]The full Java source code and its specification is shipped as one of the examples with the KeY system.

to the schema variables `?X` resp. `?Y` to be used in lines 8 and 9 where they are parameters for the proof command `instantiate`.

In this section we describe the language KPS which we have developed based on our concept presented in Sect. 7 to develop proofs in a text-based way with the KeY system. As proof construction is characterized by selecting and manipulating proof goals, the concept behind KPS considers *goal selectors* and *mutators* to be the basic building blocks for a proof scripting language, as introduced in the concept description.

Types in KPS We chose to design KPS as a statically typed language to catch certain scripting errors in advance. This should enable the users to retrieve type-error messages before the interpretation of proof scripts in order to not disturb the proof process during interpretation. However, limitations exist for expressions where the evaluation is script state dependent, e.g., the matching expressions. The types that we support in KPS are booleans (`bool`), unbounded integers (`int`), strings (`string`) and terms (`TERM`). Terms can have a sort S (`TERM<S>`) which depends on the sorts for terms supported by the underlying theorem prover KeY. Variables containing a sequent or a formula have the type `TERM<bool>`.

7.6.1. Syntax of KPS

The syntax of KPS is presented in Figure 7.3. A KPS script starts with the keyword `script` followed by an identifier denoting the name of the script *id* and possibly named arguments enclosed in parentheses (*args*). Arguments are always given in the form *id*:=**value**. Such a script is referred to as a *proof script* or simply *script* in the following.

The script's body is a sequence of KPS statements. Statements in KPS are either:

- assignments of values to variables,
- control-flow statements (element (*ctrlflowkeyword*){*statements**} in the grammar in Fig. 7.3),
- error handling statements (element (*errkeyword*){*statements**} in the grammar in Fig. 7.3), and
- the selector statement *cases* which contains matching expressions.

Expressions in KPS are expressions over the domain elements, i.e. matching expressions over terms or binary expressions over integers. We denote binary operators over the domain objects with **bop**, unary operators with **uop** and we denote concrete domain elements with **lit** including the elements **stringLit**, which denotes strings, and **termLit**, which denotes terms and formulas following the syntax for terms and formulas in the KeY system enhanced with special symbols and operators for matching expressions. Special in KPS are the expressions over proof goals and terms. In KPS the substitution expressions for terms (see 7.7 in the grammar in Fig. 7.3) allow to substitute schema variables appearing in a term by values from the current goal. Furthermore, in KPS, special expressions over proof goals and terms which can only be used in a `case` selector respectively in a matching expression exist: the proof exploration expressions `closes` and `derivable` (see 7.9 and 7.10 in Fig. 7.3).

Additionally, KPS contains matching expressions (see 7.10 in the grammar in Fig. 7.3) over the domain elements sequent, term, goal and rules, that are boolean expressions which can change the script state. Matching expressions and their evaluation are described in detail in Sect. 7.7.1. Furthermore, the abstract syntax in Fig. 7.3 does not contain a further decomposition of the non-terminals *id* and *type*. The non-terminal *id* denotes names for variables or proof commands and with *type* we denote the possible types in KPS.

As already briefly mentioned, during script execution errors may occur. The intention of the keywords `strict` and `relax` in front of a block of statements to control how errors that may occur during the interpretation of a block are handled. Errors in a block may occur for example if a proof command is not applicable because of the ambiguity of formulas where a command may be applied to or because the formula to which a command may be applied to is not present on the sequent anymore. The keyword `strict` reports errors directly back to the user and the interpretation of the script is terminated. If `relax` is used, statements that result in an error when applying them in a script state are skipped and the next statement following the failing statement in the block is evaluated in the state instead.

Mutators Mutators are commands which modify a single proof goal by manipulating the verification condition or changing the variable assignment. If a proof state contains more than one goal, before applying mutators a single goal has to be selected.

Definition 7.6.1 (Mutator)**.** *Let $\mathcal{S}$ be the set of script proof states. A mutator m is a function $m : S \mapsto S$ which is undefined for $g \in \mathcal{G}$ with $g = \bot$, denoting that no proof goal is selected.*

Examples for mutators are rule applications and proof strategies, which consist of several rule applications, typically provided by the underlying verification system, calls to sub-scripts and script variable assignments. In the language KPS mutators are realized by providing the command's name and positional arguments specific to the command.

In the example in Fig. 7.5, one of the mutators is `autopilot_prep` (line 2), an internal prover strategy of KeY that performs symbolic execution of the program to be verified with intermediate simplification steps. Another mutator in the example is `instantiate` (lines 8 and 9), which is a rule with the named parameters `var` and `with`. This rule instantiates the quantified variable bound by the value of `var` with a term provided as argument for the variable `with`.

Goal selectors Proof goals in a script proof state can be selected by using the selection statements as introduced in the description of the concept. The three different types of selectors are realized in KPS using the keywords `foreach`, `theonly`, and `cases` each followed by a block of statements. These selectors allow to select a subset of proof goals from a script proof state.

$$
\begin{align}
\mathit{script} &::= \texttt{script}\ \mathit{id}\ (\mathit{args})\ \{\mathit{statements}^*\} \tag{7.1}\\
\mathit{statements} &::= (\mathit{assignment} \mid \mathit{invokeMutator}) \mid \mathit{cases} \mid \tag{7.2}\\
&\quad (\mathit{ctrlflowkeyword} \mid \mathit{errkeyword})^+\{\mathit{statements}^*\} \tag{7.3}\\
\mathit{assignment} &::= \mathit{id}\ (:\ \mathit{type})^?\ := \mathit{expression}; \tag{7.4}\\
\mathit{invokeMutator} &::= \mathit{id}\ (\mathit{id}{=}\mathit{expression})^*\ \mathit{expression}^*\ ; \tag{7.5}\\
\mathit{expression} &::= \mathit{expression}\ \mathbf{bop}\ \mathit{expression} \mid \mathbf{uop}\ \mathit{expression} \mid \tag{7.6}\\
&\quad \mathit{expression}[\ (\mathit{id}/\mathit{expression})^*\ ] \mid \mathbf{lit} \tag{7.7}\\
\mathit{cases} &::= \texttt{cases}\ \{\ (\mathit{case})^* \tag{7.8}\\
&\qquad (\texttt{default}\ :\ \mathit{statements}^*)^?\} \notag\\
\mathit{case} &::= (\texttt{case}\ \mathit{matchExpr} \mid \texttt{closes}\ \{\mathit{statements}^*\})\texttt{:}\ \{\mathit{statements}^*\} \tag{7.9}\\
\mathit{matchExpr} &::= \texttt{match}\ (\mathbf{stringLit} \mid \mathbf{termLit} \mid \texttt{derivable}\ \mathbf{termLit}) \mid \tag{7.10}\\
&\quad \mathit{expression} \mid \mathit{matchExpr}\ (\ \texttt{\&} \mid \texttt{|}\ )\ \mathit{matchExpr} \notag\\
\mathit{ctrlflowkeyword} &::= \texttt{foreach} \mid \texttt{theonly} \mid \texttt{repeat} \tag{7.11}\\
\mathit{errkeyword} &::= \texttt{strict} \mid \texttt{relax} \tag{7.12}\\
& \tag{7.13}
\end{align}
$$

Figure 7.3.: Syntax of KPS. The individual components are described in Section 7.6.1. The symbols **bop**, **uop** and **lit** are abbreviations for the standard binary operators, unary operators and literals (see Section 7.6.1 for details). With **stringLit** and **termLit** we denote string and term respectively formula literals, which both are a subset of **lit**. The operators `&` and `|` are binary operators and are part of **bop**.

With `foreach`, a mutator is applied to all proof goals (lines 3 and 4 in Ex. 7.5 on page 173). This selector allows to uniformly mutate all proof goals in a state. The behaviour of the selector `theonly` is similar to the behaviour of `foreach` with the difference that the selector has an additional effect in the case that it is applied in a state which contains more than one proof goal. The effect is a warning to the user that the mutators enclosed in the `theonly` block is applied to more than one proof goal. The selector `theonly` can be used if a user expects a state to only consist of one goal, but does not want to disturb the proof script interpretation if this assumption is not valid. The `cases` selector is used to make case distinctions over proof goals based on matching expressions (in lines 5 to 10 in Fig. 7.5 on page 173 there are two cases). This selector allows for a more fine grained selection to mutate a subset of proof goals uniformly. Selection criteria for proof goals is defined using case expressions containing matching expressions (7.9 in Fig. 7.3 on page 173). Matching expressions can be of three types: expressions over the local goal variables, syntactical as well as semantical conditions over the constituents of a proof goal (e.g., formulas of the open verification condition).

Besides explicit variable assignments, also the evaluation of matching expressions manipulate the proof script variables: After the evaluation of a matching expression, the state is updated by variable bindings resulting from the evaluation. If matching expressions are combined in a case expression, such as for example in the expression `case match '?X & p(a)==> '& match '==> ?X'` the variable `?X` has to be bound to the syntactically same term with the same type in both expressions, such that the combined expression evaluates to true. In this case the expression matches to sequents that contain a conjunction in the antecedent, where the second conjunct is the predicate `p(a)` and where the first conjunct is also present in the succedent. For example the match expression matches to the sequent `p(b) & p(a) ==> p(b)`. The semantics of matching expressions over the proof obligation of proof goals will be presented in Section 7.7.1.

7.6.2. Configuration and Variables

Before introducing the formalized semantics of KPS, we have to introduce the notion of configuration or script state.

A configuration σ_{script} in KPS consists of a script proof state (as defined in Def. 7.4.1 on page 168) together with a proof script π to be executed. As the configuration contains a proof script state, it therefore also contains the open goals of the proof state of which at most one is selected. The connection between the proof script state and the configuration is depicted in the following equation:

$$\sigma_{script} = (\overbrace{(g, \underbrace{\mathcal{G}}_{\text{proof state}}, \mathcal{V})}^{\text{script proof state}}, \pi)$$

Definition 7.6.2 (Configuration)**.** *Let π be a proof script in* KPS *and $\sigma = (g, \mathcal{G}, \mathcal{V})$ a script proof state. We define the* configuration*for π and σ to be $\sigma_{script} = (\sigma, \pi)$ or also written as $\langle \pi, \sigma \rangle$.*

In the following we will also refer to configurations using the term *script state*. Besides the variables occurring in the proof state σ (i.e., logical variables and program variables in the sequents in σ), in KPS, *script variables* can be used. These variables can be of one of the types mentioned before in 7.6.

Script variables are declared and bound to values within proof scripts. The values can be modified by *assignments*, mutators and by implicit bindings within a match expression (as introduced in Sec. 7.7.1). For convenience, we also expose the configuration of the underlying proof system as special variables. These variables start with an underscore "_". This design allows changing prover settings during proof construction. For example, the `_KEY_MAX_AUTO` variable is used for setting the maximal number of steps that are automatically applied by the prover's strategies when invoked. A list of currently available configuration variables can be found on the webpage[3]. Regular script variables and configuration variables can be used in expressions.

All variables are *goal-local*, meaning that their value is determined by the current selected goal and inherited by states resulting from the application of mutators. An assignment to a variable in a script state (σ, π) respec. $((g, \mathcal{G}, \mathcal{V}), \pi)$ updates the function $\mathcal{V}$.

In our implementation of an interpreter and debugger for KPS (with KeY as underlying verification system) the following information about an open proof goal is available:

- a set of goal-local script-variables and their values (in $\mathcal{V}$)
- a sequent (which is part of g)
- different types of labels (which are part of g and provided by the underlying proof system):
  - *branching labels*, which name the different proof branches
  - *rule labels*, which contain the name of the applied rule
  - *program lines labels*, which indicate which lines of the program to prove are already symbolically executed
  - *program statements labels*, which indicate which statements of the program to prove are already symbolically executed.

7.7. Formalized Semantics of KPS

We will define the semantics of KPS using operational semantics. For the definition of the operational semantics of our language we need to define the state content selector function and introduce the notation for the execution of a script:

Definition 7.7.1 (State Content Selector)**.** *Let $\mathbb{G}$ be the power set of $\mathcal{G}$. We define the state content selector as the function* $\mathrm{goalsOf}\colon \mathbb{G} \mapsto \mathcal{S}$ *with*
$\mathrm{goalsOf}((g, \mathcal{G}, \mathcal{V}), \pi) = \mathcal{G}$.

For the execution of scripts we will use the notation $\langle \pi, \sigma \rangle \rightsquigarrow \langle \pi', \sigma' \rangle$ to denote that the program π is executed from state σ to σ' with π' being the rest of the script σ that is

[3]https://formal.iti.kit.edu/psdbg

not executed yet. If $\pi' = \epsilon$, we denote that a script is fully executed until no statement is left and the empty script ϵ is reached. If there exists a σ' such that $\langle \pi, \sigma \rangle \rightsquigarrow \langle \epsilon, \sigma' \rangle$ then the execution of the script has terminated and σ' is the final state. In KPS, there is only one script statement that can prevent termination, which is the `repeat` statement. The termination of all other statements, especially of mutators rely heavily on the termination of the proof search strategies of the underlying verification system. For this reason mutators may have a timeout variables as an argument to be able to control the termination of the proof search. One example for such a mutator would be a mutator that calls an external solver. If an error occurs during the execution of a script, no final state σ' exists, and the script execution is stuck. We provide two keywords to control the handling of a stuck execution: `strict` and `relax`. In case the error control keyword `strict` is used, the execution stays stuck. In case the keyword `relax` is used, the erroneous statement is skipped and the next statement is executed instead, i.e., execution proceeds.

When the execution of the proof script in configuration $\langle \pi, \sigma \rangle$ results in $\langle \epsilon, \sigma' \rangle$ i.e., $\langle \pi, \sigma \rangle \rightsquigarrow \langle \epsilon, \sigma' \rangle$, we write $\langle \pi, \sigma \rangle \rightsquigarrow \sigma'$ to denote the state[4] σ' that results from the execution.

To evaluate expressions we furthermore need an evaluation function eval_g which evaluates an expression in a goal g to a value. For simple Boolean expressions the semantics are defined as usual, arithmetic expressions are evaluated over the unbounded integers $\mathbb{Z}$. Variables are evaluated in the currently selected goal, i.e., if the variables are not declared for the selected goal (i.e., for a variable x the function $(\mathcal{V}(g))(x)$ is undefined), the evaluation of the variable results in an error, i.e., the script execution is stuck. The same holds for violation of the type conformance during the assignment values to variables.

Matching expressions M in a goal g is done using the function $\text{matches}(M, g)$ which performs a single-sided matching of the goal g and the matching expression M, which will be defined in Section 7.7.1. Here we will only briefly introduce the idea of this function applied to matching expressions. As notation we will use θ for a substitution, containing *replacements* in the form of $(?X \mapsto t) \in \theta$ for a schema variable $?X$ and a term or formula t. With Θ we denote the set of all substitutions. The function $\text{matches}(M, g)$ can return three results:

- $\theta_\perp$, indicating that the matching expression M did not match the goal g.
- $\mathcal{V}_m \subseteq \Theta$, as a result of a successful matching, where each element of $\mathcal{V}_m$ is a substitution and $\mathcal{V}_m \neq \theta_{id}$.
- $\mathcal{V}_m = \{\theta_{id}\}$, as a result of a successful matching, where θ_{id} denotes the identity function i.e., the identity matching that does not need a substitution.

To evaluate a matching expressions M in a goal g, the function $\text{eval}_g(M)$ translates the results of $\text{matches}(M, g)$. The function $\text{matches}(M, g)$ evaluates the matching expression M in a goal g and returns whether the goal could be matched against the matching expression. This is independent of whether the matching expression is an expression over the script variables, a syntactic or a semantic matching expression. When applied

[4]The proof scripting language is deterministic and we assume the underlying proof system to construct proofs deterministically.

to a syntactic matching expression, the function eval translates the result $\theta_\perp$ to false and the result $\mathcal{V}_m$ with $\theta_\perp \notin \mathcal{V}_m$ to the value true. If the matching was successful with $\mathcal{V}_m \neq \{\theta_{id}\}$, then eval transforms one element of the set from $\mathcal{V}_m$ to variable assignments. Furthermore, in this function has as side effect that the goal's variable assignments are updated with the matching results. Evaluating matching expressions therefore has two results: the indication whether the expression matched the goal and the variable binding from the substitution in case of a successful match. Matching expressions can be combined using the two Boolean operators `&` and `|`, where both operands may be matching expressions containing schema variables. Matching expressions are always evaluated independently. The evaluation of a matching expression is performed before substitutions to matched variables occur. Therefore, if a matching expression contains a variable substitution, the value of the substituted variables is taken from the pre-state of the evaluations of the matching expressions. Furthermore, if the name of variables of two combined matching expressions is equal, the binding after the evaluation of the matching expression has to be equal regarding the type and the value.

Mutators. In the following the semantics of mutators is defined. A mutator is an *assignment* or a *call* statement and abstractly defined as the function from script proof state to script proof state ($m : S \mapsto S$ in Sect. 7.6.1). In the following we will define the semantics of the interpretation of mutators for configurations.

We have to distinguish three types of mutators: (a) assignments, (b) calculus rules or strategies, or (c) calls to sub-scripts.

Executing assignment statements in a goal g in KPS results in evaluating the expression on the right-hand side of the assignment statement and updating the variable function for the goal g. The types of the right and the left-hand side have to conform. For simple types, such as Boolean or integers the types have to be equal, for parameterized types, in KPS this is the type `TERM<SORT>`, two terms are conform, if the sorts are compatible (which is dependent on the type system of the underlying proof system). Therefore, executing an assignment statement is only successful if the variable and the value of the expression are well-typed. Otherwise, a typing error occurs and the program execution would get stuck.

Definition 7.7.2 (Assignment Statement). *Let $(g, \mathcal{G}, \mathcal{V})$ denote a script proof state. Furthermore, let P_1 denote a sequence of proof script statements and let x be a variable name. We define the evaluation of an assignment statement $x := expr;$ in a proof script $x := expr; P_1$ as:*

$$\textit{assignment}\ \frac{\mathcal{V}' := (\mathcal{V}(g))(x) \mapsto eval_g(expr)}{\langle x := expr; P_1, (g, \mathcal{G}, \mathcal{V})\rangle \rightsquigarrow \langle P_1, (g, \mathcal{G}, \mathcal{V}')\rangle} \tag{7.14}$$

If the mutator is a calculus rule or strategy the evaluation of this mutator in the script state requires a call to the underlying proof system. This call returns a set of possibly empty or newly created proof goals. To capture this behavior we will assume a given function $\mathcal{PROVER}$ which takes a proof goal, the variables of the proof goal and a proof command as arguments. This function has three different return values.

If the function returns an empty set of proof goals, all proof branches were closed by the underlying proof system using the given calculus rule or strategy. If the function returns a set of newly created proof goals, the prover created new siblings, and if the proof goal that was passed as parameter is returned, the calculus rule was not applicable or the rule does not exist in the set of rules of the underlying prover. In the following, we will use the term *proof command* for both, calls to calculus rules and calls to the underlying prover strategies.

Invocations of a proof command with named parameters result in evaluating these parameters. The evaluated parameters are added to the variable assignments that are passed on, together with the selected goal and the proof command, to the prover. The call to the prover has three results:

1. The selected goal remains unchanged (see commandAppSingle in Definition 7.7.3), with $g' = g$). In this case the state remains unchanged.
2. The prover returns a single new goal which is different from the passed goal (see commandAppSingle in Definition 7.7.3). In this case the state is updated. The goal passed to the prover is removed from the set of open goals and the newly created goal is added to that set replacing the removed goal. Additionally, this newly added goal is set as selected goal.
3. The prover returns a non-singleton set of new open proof goals (see commandAppMulti in Definition 7.7.3). In this case the passed goal is removed from the set of open proof goals, the newly created set is added to the set of open proof goals and the selected goal is set to $\bot$, indicating that no goal is selected in the state.

Before calling the underlying proof system, the parameters passed as arguments need to be evaluated in the script state and a variable assignment function $\mathfrak{W}$ is created. The underlying proof system is then called with the assignments, the proof command and the proof goal. The script state is then updated with the result of the proof system. If the evaluation of the parameters results in an error there are two possibilities: when the error keyword `relax` is given the offending mutator statement is skipped, otherwise the script execution will get stuck.

Definition 7.7.3 (Invocations of Prover Commands). *Let $\mathbb{M}$ be the set of all available proof commands of the underlying proof system, let $\mathbb{G}$ be a set of proof goals and $\mathbb{V}$ be a set of variable assignment functions. Given a prover evaluation function $\mathcal{PROVER} : \mathbb{M} \times \mathbb{G} \times \mathbb{V} \to \mathcal{P}(\mathbb{G})$. Let `command` $\in \mathbb{M}$ be a name of a calculus rule or prover strategy. Let $(g, \mathcal{G}, \mathcal{V})$ be the script proof state before executing the proof command with parameters `command` $id_1 = expr_1, \ldots, id_n = expr_n$ with id_i being an identifier and $expr_i$ expressions:*

$$\frac{\begin{array}{c}\mathfrak{W} := \{(id_1, eval_g(expr_i1)), \ldots, (id_n, eval_g(expr_n))\} \\ \mathcal{PROVER}(\texttt{command}, g, \mathfrak{W}) = \{g'\} \\ \mathcal{G}' := (\mathcal{G} \setminus \{g\}) \cup \{g'\}\end{array}}{\langle \texttt{command}\ id_1 = expr_1, \ldots, id_n = expr_n; P_1, (g, \mathcal{G}, \mathcal{V})\rangle \rightsquigarrow \langle P_1, (g', \mathcal{G}', \mathcal{V})\rangle}$$

(commandAppSingle)

Note that it is possible that $g' = g$, in which case also $\mathcal{G}' = \mathcal{G}$. In this case there is no change in the state. This is the case when a proof command is not applicable or could not be found.

$$\frac{\begin{array}{c}\mathfrak{W} := \{(id_1, eval_g(expr_i 1)), \ldots, (id_n, eval_g(expr_n))\} \\ \mathcal{PROVER}(\texttt{command}, g, \mathfrak{W}) = \{g_1, \ldots, g_n\} \\ \mathcal{G}' := (\mathcal{G} \setminus \{g\}) \cup \{g_1, \ldots, g_n\} \\ n > 1\end{array}}{\langle \texttt{command}\ id_1 = expr_1, \ldots, id_m = expr_m; P_1, (g, \mathcal{G}, \mathcal{V})\rangle \rightsquigarrow \langle P_1, (\bot, \mathcal{G}', \mathcal{V})\rangle}$$
(commandAppMulti)

KPS includes the selector `cases` that allows grouping and similar treatment of proof goals. For the definition of `cases` we will introduce a filter function $\lhd$. The filter function $\lhd$ filters a set of goals according to the evaluation of a given matching expression M. For a proof goal $g \in \mathcal{G}$ in a configuration, if and only if $\text{eval}_g(M)$ returns true then g is in the return set of $\mathcal{G} \lhd M$. The function $\text{eval}_g(M)$ only returns true if the function $\text{matches}(M, g)$ does not return $\{\theta_\bot\}$.

Definition 7.7.4 (Filter Function $\lhd$)**.** *Let M be a matching expression, $\mathcal{G}$ a set of proof goals and $g \in \mathcal{G}$ a proof goal. We define the filter function $\lhd$ such that $\mathcal{G} \lhd M := \{g \in \mathcal{G} |\, \text{eval}_g(M) = true\}$.*

Let $\mathcal{G}_i$ denote a set of proof goals. Let B denote a Boolean expression and M denote a matching expression, which is a special Boolean expression that can manipulate the state by assigning values to variables. For simplification, we omit the variable assignments in the states in the following. The variable assignments are updated with every statement if the statement's effect contains variable assignments. The semantics of the `cases` statement is a usual fall-through semantics of guarded commands.

The effect of a `cases` statement is that the goals in the script state are grouped according to the match conditions M of the respective `case`.

For each goal that was matched by a match condition the body of the respective `case` is executed. All goals not matched by a match condition are passed to the next case. The last `case` can be a `default` case, which is the same as `case true`, which matches all goals. At the end of a `cases` statement the resulting goals are collected in the goal set of the post state. In the formalized semantics of the `cases` selector, let $\mathcal{G}_i$ in a script proof state $(g, \mathcal{G}_i)$ denote a set of proof goals. Let M denote a matching expression and "_" denote a *don't care symbol* in a script proof state $(\_, \mathcal{G})$.

$$\textit{casesNonEmpty} \quad \frac{\begin{array}{c}\langle \texttt{foreach}\{P_0\}, (\bot, \mathcal{G} \lhd M)\rangle \rightsquigarrow \sigma_1 \\ \mathcal{G}_1 := \text{goalsOf}(\sigma_1) \\ \langle \texttt{cases}\{P_1\}, (\bot, \mathcal{G} \lhd \neg M)\rangle \rightsquigarrow \sigma_2 \\ \mathcal{G}_2 := \text{goalsOf}(\sigma_2)\end{array}}{\langle \texttt{cases}\{\texttt{case}\ M\colon P_0; P_1\}\ ;\ P_2, (\_, \mathcal{G})\rangle \rightsquigarrow \langle P_2, (\bot, \mathcal{G}_1 \cup \mathcal{G}_2)\rangle} \tag{7.15}$$

$$\textit{casesDefault} \quad \frac{}{\begin{array}{c}\langle \texttt{cases}\{\texttt{default}\colon P_0;\ P_1\}\ ;\ P_2, (\_, \mathcal{G})\rangle \rightsquigarrow \\ \langle \texttt{cases}\{\texttt{case}\ true\colon P_0; P_1\}\ ;\ P_2, (\_, \mathcal{G})\rangle\end{array}} \tag{7.16}$$

$$\textit{casesEmpty}\ \frac{}{\langle \texttt{cases}\{\}, (g,\mathcal{G})\rangle \rightsquigarrow \langle \epsilon, (\bot,\mathcal{G})\rangle} \tag{7.17}$$

To be able to apply the same sequence of statements to more than one goal, KPS contains the `foreach` statement. Here, each goal of the script state is selected and the body of the `foreach` statement is executed with this selected goal. All resulting open goals of the body's executions are collected as set of open goals at the end of the `foreach` statement for the post state. In the definitions we have added a semicolon as separator between the `repeat` and the `foreach` statement and their successor statements. These semicolons serve only as indicators for the sequential composition of statements and are not needed in the actual script language.

$$\textit{foreach}\ \frac{\begin{array}{c}\text{for all } g \text{ with } g \in \text{goalsOf}((g,\mathcal{G})).\langle P_0, (g,\{g\})\rangle \rightsquigarrow \sigma_g\\ \mathcal{G}' := \bigcup_{g\in\text{goalsOf}((g,\mathcal{G}))}(\text{goalsOf}(\sigma_g))\end{array}}{\langle \texttt{foreach}\{P_0\}\ ;\ P_1, (g,\mathcal{G})\rangle \rightsquigarrow \langle P_1, (\bot,\mathcal{G}')\rangle} \tag{7.18}$$

The execution of the `theonly` in a script state is similar to the execution of a `foreach` statement. The difference is the assumption by the user that the set of open goals only $\mathcal{G}$ in the pre-state of the `theonly` statement contains only one element. If this set contains more than one element $|\mathcal{G}| \geq 1$, it has to be reported to the user upon execution of the `theonly` statement.

To be able to iteratively apply proof commands, KPS contains the `repeat` statement. The `repeat` loop is iterated as long as the loop body changes the script proof state, i.e., as long as the set $\mathcal{G}$ changes.

$$\textit{repeatLoop}\ \frac{\langle P_0, (g,\mathcal{G})\rangle \rightsquigarrow \langle \epsilon, (g',\mathcal{G}')\rangle \qquad \mathcal{G} \neq \mathcal{G}}{\langle \texttt{repeat}\{P_0\}\ ;\ P_1, (g,\mathcal{G})\rangle \rightsquigarrow \langle \texttt{repeat}\{P_0\}\ ;\ P_1, (g',\mathcal{G}')\rangle} \tag{7.19}$$

$$\textit{repeatEnd}\ \frac{\langle P_0, (g,\mathcal{G})\rangle \rightsquigarrow \langle \epsilon, (g',\mathcal{G}')\rangle \qquad \mathcal{G} = \mathcal{G}'}{\langle \texttt{repeat}\{P_0\}\ ;\ P_1, (g,\mathcal{G})\rangle \rightsquigarrow \langle P_1, (g',\mathcal{G}')\rangle} \tag{7.20}$$

The execution of the `repeat` statement is not guaranteed to terminate. The termination relies on the termination of the statements within the body of the `repeat` statement. The statements, besides the `repeat` statement, that may not terminate are native mutators. The termination of native mutators relies on the termination in the underlying verification system, which usually contains possibilities to set a timeout for the proof search. For the termination of the `repeat` statement itself the interpreter for the language has to contain means to abort execution. One possibility is to maintain a counter and abort execution when the counter reaches a pre-defined maximal number of executions.

We also add the constructs if and while, which are typical in programming languages. These constructs do not add new functionality to the scripting language and can be defined in terms of the aforementioned building blocks as follows:

$$if\ \frac{}{\langle \texttt{if}\ B\ \{P_0\}\ ;\ P_1, \sigma\rangle \rightsquigarrow \langle \texttt{cases}\{\texttt{case}\ B\colon P_0\}; P_1, \sigma\rangle} \tag{7.21}$$

$$whileLoop\,\frac{\langle P_0, (g, \mathcal{G})\rangle \rightsquigarrow \langle \epsilon, (g', \mathcal{G}')\rangle \qquad \mathrm{eval}_g(B)}{\langle \texttt{while}\ B\ \{P_0\}\ ;\ P_1, (g, \mathcal{G})\rangle \rightsquigarrow \langle \texttt{while}\ B\ \{P_0\}\ ;\ P_1, (g', \mathcal{G}')\rangle} \tag{7.22}$$

$$whileEnd\,\frac{\langle P_0, (g, \mathcal{G})\rangle \rightsquigarrow \langle \epsilon, (g', \mathcal{G}')\rangle \qquad \neg\,\mathrm{eval}_g(B)}{\langle \texttt{while}\ B\ \{P_0\}\ ;\ P_1, (g, \mathcal{G})\rangle \rightsquigarrow \langle P_1, (g', \mathcal{G}')\rangle} \tag{7.23}$$

7.7.1. Evaluation of Matching Expressions

In the semantics definition for KPS we have excluded the definition of a matching expression in a `cases` statement. We define matching expressions over the logical structures of a proof script goal, i.e., the proof obligation. As we have instantiated our language for the KeY system, the logical structure of a proof goal is a sequent in JavaDL. In the following, to simplify the presentation, all presented expressions are implicitly parenthesized, similar to the presentation of terms and formulas in KeY.

Example 7.6. Let's consider a simple example: a user provides the matching expression `p(?X) && p(y)`, where `?X` stands for a placeholder for a concrete formula/term. This expression matches against the concrete formula `p(x) && p(y)`, if the placeholder `?X` is replaced by the concrete term `x`. However, against the formula `p(x) && p(y) && p(z)` it does not match, as no substitution for `?X` can be found, such that both formulas are syntactically equal. To match any sub term with the pattern `?X && p(y)`, one may use the pattern `...?X && p(y)...` containing the *ellipsis operator* `...<PATTERN>...` which contains a match pattern `<PATTERN>`.

For the definitions of matching schematic formulas and sequents, we will extend the (usual) definition of the set of all first-order logic formulas $\mathcal{F}$ over a signature Σ by allowing the use of schema variables from the set Var_{Sch} (disjoint from the symbols in the signature of set $\mathcal{F}$, i.e., disjoint from the sets of predicates and function symbols and *Vars* of $\mathcal{F}$) in place of terms or formulas and as quantified variables. To denote schema variables in a schematic formula, we use the syntax $?X$ in the following. For the following definitions, we consider formulas to be terms of type Boolean and may use the notion *term* and *formula* interchangeably. The logical connectors are therefore considered to be special predicate symbols, i.e., a formula $A \wedge B$ is denoted by using the prefix notation $\wedge(A, B)$ for convenience reasons.

Besides schema variables, schematic formulas may additionally be constructed using the following three elements in place for terms and formulas:

1. the *don't care* symbol ? that matches any term or formula,
2. the matchbinder `<TERM_PATTERN>:<SCHEMA_VARIABLE>` to bind a term or formula that has been successfully matched against the `<TERM_PATTERN>` to the variable given by `<SCHEMA_VARIABLE>`, and

3. the ellipsis ...`<TERM_PATTERN>`..., to denote that the term or formula appears as subterm[5] or sub-formula within another term or formula.

The set of all schema formulas is denoted with $\mathcal{F}_{Sch}$ and by definition, every concrete formula is a schema formula $\mathcal{F} \subset \mathcal{F}_{Sch}$.

Furthermore, let $\mathcal{S}$ be the set of all first-order logic sequents consisting of first-order logic formulas, and $\mathcal{S}_{Sch}$ be the set all schema sequents, consisting of schema formulas, with $\mathcal{S} \subset \mathcal{S}_{Sch}$.

In the following, the semantics of sequent matching expressions is defined recursively. We start by defining the semantics of match_t, which matches a schema term against a concrete term. Using this definition as basis we will build the definition of the semantics of match_{seq} which matches a schema sequent against a concrete sequent.

Let $t \in \mathcal{F}$ be a term and let $\text{subterms}(t)$ be the function that returns the set of all subterms of t. For a schema variable $?X$ and a value val, we call the mapping $?X \mapsto val$ within a substitution a *replacement* (we also use the notation $(?X, val)$ in the following to denote a replacement). The value val is in this case a concrete term. In general, we include the convention, if a schema variable is not explicitly included in a substitution, it is mapped to itself, i.e., $\theta(?X) = ?X$. Applying a mapping $v \in \theta$ with $v = (?X \mapsto val)$ to a schema term or formula f results in a schema formula with all occurrences of $?X$ in f replaced by val. We may use the set notation for a substitution containing only those replacements that are not mapped to itself in the form of pairs, i.e., a substitution θ with $dom(\theta) = \{?X, ?Y, ?Z\}$ with $?X \mapsto ?X$, $?Y \mapsto a$ and $?Z \mapsto a$ we will write $\{(?Y, a), (?Z, a)\}$. The notation for the substitution θ_{id} will therefore the empty set $\emptyset$.

Definition 7.7.5 (Substitution)**.** *A substitution $\theta : Var_{Sch} \mapsto Var_{Sch} \cup \mathcal{F}$ is total function that maps schema variables to formulas and terms over Σ or to schema variables. If for all $?X \in Var_{Sch}$. $\theta(?X) = ?X$ we call this substitution θ_{id}.*

A substitution will be denoted by θ and the set of all substitutions by Θ. We will include a special substitution, denoted by $\theta_\perp$ that represents an *unsuccessful* match. The special substitution is not in Θ as it is not a total function and not a valid substitution. One distinctive substitution is the identity function denoted by θ_{id} with $\theta_{id} \in \Theta$.

Matching Terms

In the definition of the matching functions, sub-results of matchings need to be compared and unified where possible. For this comparison we have to define in which case substitutions are *compatible* beforehand.

Definition 7.7.6 (Compatible Substitutions)**.** *Let $?X$ be a schema variable. We call two substitutions $\theta_1, \theta_2 \in \Theta$* compatible *if and only if*

$$\forall\ ?X.\ \theta_1(?X) \neq ?X \wedge \theta_2(?X) \neq ?X \rightarrow \theta_1(?X) = \theta_2(?X) \tag{7.24}$$

holds. For any substitutions θ_i holds that θ_i and $\theta_\perp$ are not compatible. Furthermore, $\theta_\perp$ is compatible with $\theta_\perp$.

[5]Note that for our Chapter we define that the term t is part of the set of subterms of t.

We call two substitutions *compatible*, iff all schema variables occurring in both substitutions either have a different name, or if the names are equal the types are equal and they are assigned the same value. We assume that variables are well-typed. In case the types of variables with the same name but different types exists in two substitutions, the two substitutions are not compatible.

We define the function *clash-free composition* that returns $\theta_\bot$ if two substitutions are not *compatible* and a composition of the substitutions otherwise.

Definition 7.7.7 (Clash-free Composition). *We define the function* $\oplus : \Theta \cup \{\theta_\bot\} \times \Theta \cup \{\theta_\bot\} \to \Theta \cup \{\theta_\bot\}$ *such that:*

$$(\theta_1 \oplus \theta_2)(?X) = \begin{cases} \theta_\bot & \textit{if } \theta_1 \textit{ and } \theta_2 \textit{ are not compatible} \\ \theta_1(?X) & \textit{if } \theta_1(?X) \neq ?X \textit{ and } \theta_1 \textit{ and } \theta_2 \textit{ are compatible} \\ \theta_2(?X) & \textit{if } \theta_1(?X) = ?X \textit{ and } \theta_1 \textit{ and } \theta_2 \textit{ are compatible} \end{cases}$$

We have to overload the function *clash-free composition* from the pairwise composition to the composition of an arbitrary number of substitutions to be able to use it in the following.

Definition 7.7.8 (Extension of Clash-free Composition). *We define the application of the function* $\oplus$ *to an n-tuple of substitutions as:*

$$\oplus((\theta_1, \theta_2, \ldots, \theta_n)) = ((((\theta_1 \oplus \theta_2) \oplus \ldots) \oplus \theta_{n-1}) \oplus \theta_n)$$

When unifying sets of substitutions, it is possible that the special symbol $\theta_\bot$ is part of the substitution set. This can happen when using the ellipsis in a matching expression, as the ellipsis recursively matches all subterms. While evaluating matching expressions, we are only interested in successful matching results, if they exist. To exclude unsuccessful matching results, we define the function *filter bottom* that removes the substitution $\theta_\bot$ from a set of substitutions if the unsuccessful match is part of the set of substitutions.

Definition 7.7.9 (Filter Bottom). *Let* $\mathbb{V}_1 \subseteq \mathcal{P}(\Theta \cup \{\theta_\bot\})$ *be a set of substitutions. We define the function* $\Downarrow : \mathcal{P}(\Theta \cup \{\theta_\bot\}) \to \mathcal{P}(\Theta)$ *as follows:*

$$\Downarrow(\mathbb{V}_1) = \begin{cases} \mathbb{V}_1 \setminus \{\theta_\bot\} & \textit{if } \exists \theta_i \in \mathbb{V}_1 \colon \theta_i \neq \theta_\bot \\ \{\theta_\bot\} & \textit{if } \mathbb{V}_1 = \{\{\theta_\bot\}\} \end{cases}$$

Example 7.7. Lets consider matching the matching expression `...p(?X, ?Y)...` against the conrete formula `p(a,b) && p(b,a) && p(c,c)`. This matching expression matches the first subformula with the substitution `{(?X, a), (?Y, b)}`, the second subformula with `{(?X, b), (?Y, a)}` and the third subformula with `{(?X, c), (?Y, c)}`.
The result of the matching is a set with the three substitutions, i.e.,
`{{(?X, a), (?Y, b)}, {(?X, b), (?Y, a)}, {(?X, c), (?Y, c)}}`.

In this Example 7.7, matching a pattern returns more than one possibility to substitute the schema variables. As we have to obtain a non-ambiguous matching result in the interpretation of KPS scripts, we have decided to return the first set of substitutions in the result set to the user. Furthermore, it would have been possible to show all results to the user and let the user decide. This however would require that the interpretation of scripts has to be interrupted and the user has to be prompted.

The evaluation of matching terms is defined using the function $match_t$, which is presented in the following.

Definition 7.7.10 (Match Term)**.** *In the following definition for the function matching terms, we denote predicate and function symbols of* Σ *with* α, β.

We define the function $\mathrm{match}_t : \mathcal{F}_{Sch} \times \mathcal{F} \to \mathcal{P}(\Theta \cup \{\theta_\perp\})$ *for term patterns* $p \in \mathcal{F}_{Sch}$ *and* concrete *terms* $t_e \in \mathcal{F}$ *as follows.*

$$\mathrm{match}_t(?, t_e) = \{\theta_{id}\} \quad \textit{for all } t_e \in \mathcal{F} \tag{7.25}$$

$$\mathrm{match}_t(t', t_e) = \begin{cases} \{\theta_{id}\} & \textit{if } t' \doteq t_e \\ \{\theta_\perp\} & \textit{otherwise} \end{cases} \tag{7.26}$$

$$\mathrm{match}_t(?X, t_e) = \{\{(?X, t_e)\}\} \tag{7.27}$$

$$\mathrm{match}_t(\ldots p' \ldots, t_e) = \Downarrow (\bigcup_{\forall t' \in \mathrm{subterms}(t_e)} \mathrm{match}_t(p', t')) \tag{7.28}$$

$$\mathrm{match}_t(p' : ?X, t_e) = \Downarrow (\{\{(?X, t_e)\} \oplus \theta_i \mid \theta_i \in \mathrm{match}_t(p', t_e)\}) \tag{7.29}$$

$$\mathrm{match}_t(\alpha(p'_1, \ldots, p'_n), \beta(t'_1, \ldots, t'_m)) = \begin{cases} \{\oplus(m) \mid m \in \Pi_{i=1}^{n} \mathrm{match}_t(p'_i, t'_i)\} \\ \qquad\qquad \textit{if } \alpha \doteq \beta \wedge n = m \\ \{\theta_\perp\} \qquad \textit{otherwise} \end{cases} \tag{7.30}$$

$$\begin{aligned} &\mathrm{match}_t(\forall\{p_1, \ldots, p_n\}(p'), \\ &\qquad \forall\{y_1, \ldots, y_m\}(t')) = \begin{cases} \{\theta_i \oplus \theta_Q \mid \theta_i \in \mathrm{match}_t(p', t') & \textit{if } m = n \\ \{\theta_\perp\} & \textit{otherwise} \end{cases} \\ &\textit{with } \theta_Q = \oplus_{0 \le i \le m}(\mathrm{match}_t(p_i, y_i)) \end{aligned} \tag{7.31}$$

The Equation 7.31 also holds for the existential quantifier $\exists$ *instead of* $\forall$.

The result of a successful match is either $\{\theta_{id}\}$ if no substitution is necessary or a set of substitutions. Matching the *don't care* symbol "?" with any term always results in a successful match without a substitution (in the equation 7.25). This is in contrast to matching a concrete term against a schema variable (in 7.27), here the result is a a set containing variable substitution of the schema variable.

Matching a concrete ground term t' against another concrete ground term t (in equation 7.26) results in a successful match, iff both terms are syntactically equal and in an unsuccessful match otherwise.

We included the possibility to match on any subterm of a term t using the *ellipsis* around a match pattern p (in 7.28) written as "$\ldots p \ldots$". All subterms are matched and the results are unified, resulting in a set of substitutions. The unsuccessful match, i.e., the substitution $\theta_\perp$, is removed from the result if successful matching results exist in the set.

Example 7.8. Lets consider matching the match pattern `... p(?X) ...` against the concrete formula `p(x) && p(f(y))`.
First the match pattern `p(?X)` is matched against all subterms of the concrete formula, so

- $\text{match}_t(\texttt{p(?X)}, \texttt{p(x) \&\& p(f(y))}) = \{\theta_\perp\}$,
- $\text{match}_t(\texttt{p(?X)}, \texttt{p(x)}) = \{\{(\texttt{?X}, \texttt{x})\}\}$,
- $\text{match}_t(\texttt{p(?X)}, \texttt{x}) = \{\theta_\perp\}$,
- $\text{match}_t(\texttt{p(?X)}, \texttt{p(f(y))}) = \{\{(\texttt{?X}, \texttt{f(y)})\}\}$,
- $\text{match}_t(\texttt{p(?X)}, \texttt{f(y)}) = \{\theta_\perp\}$,
- $\text{match}_t(\texttt{p(?X)}, \texttt{y}) = \{\theta_\perp\}$.

The union of all results is built and results in the set of substitutions $\{\theta_\perp, \{(\texttt{?X}, \texttt{x})\}, \{(\texttt{?X}, \texttt{f(y)})\}\}$. The unsuccessful match $\theta_\perp$ is removed before the result is returned from matching because successful matches are part of the resulting set. So the result of our example is a set containing the two substitutions $\{\{(\texttt{?X}, \texttt{x})\}, \{(\texttt{?X}, \texttt{f(y)})\}\}$.

To match functions or predicates the symbols and the arity have to be equal (equation 7.30 of Def. 7.7.10), for example matching `p(?X,y)` against a concrete term results in $\{\theta_\perp\}$ if the function or predicate symbols or the number of subterms of the concrete term don't match. Otherwise the subterms are matched recursively and the results of the recursive matching are consolidated using the *clash-free composition*.

Example 7.9. As an example for the matching described by equation 7.30 of Def. 7.7.10, consider matching the match pattern `p(?X,y)` against the concrete formula `p(x,y)`. As the predicate symbols p are identical and the arities of both predicates ($m = 2$ and $n = 2$) are equal the subterms have to be matched, with the following results:

- $\text{match}_t(\texttt{?X}, \texttt{x}) = \{\{(\texttt{?X}, \texttt{x})\}\}$ and
- $\text{match}_t(\texttt{y}, \texttt{y}) = \{\theta_{id}\}$.

Building the finite Cartesian product of both results results in a set containing two elements:

$$\{(\{(\texttt{?X}, \texttt{x})\},\ \theta_{id}),\quad (\theta_{id}, \{(\texttt{?X}, \texttt{x})\})\}\ .$$

Applying the clash-free union to each of the elements of the resulting set, i.e., $\{\oplus(\{(\texttt{?X}, \texttt{x})\},\ \theta_{id}),\quad \oplus(\theta_{id}, \{(\texttt{?X}, \texttt{x})\})\}$ results in the set $\{\{(\texttt{?X}, \texttt{x}), \theta_{id}\}\}$. As we can leave out empty identity substitutions, we obtain the set $\{\{(\texttt{?X}, \texttt{x})\}\}$ as final result from our matching.

Using the match expressions it is also possible to bind a term that is matched by a match pattern to a variable by using the match binder operator "`:`" together with a schema variable (in 7.29). For example, the pattern `(...p(x)...):?X` matches against all terms that contain `p(x)` as subterm and binds the matched top-level term to the variable `?X`.

Matching formulas containing quantifiers needs special matching because of the variables bound by the quantifier. In the definition we have defined the semantics in 7.31 only for the universal quantifier, however, the same definition holds for the existential quantifier as well.

A quantified formula consists of three parts: the quantifier, the set of variables bound in the scope of the quantifier y_i and the quantified formula t'. A match pattern is built out of the same components, however, instead of the concrete bound variables, match patterns p_i may be used, and instead of the bound formula a match pattern p' may be used. Matching a match pattern containing a quantifier against a concrete quantified formula is successful, if the quantifier symbols match, the patterns of the bound variables match and the quantified formula and the respective match pattern match by considering the substitutions from the matching of the bound variables.

Example 7.10. Lets consider the concrete formula `\forall y:int; p(y)` and the match pattern `\forall ?X; p(?X)`. The two match with the substitution $\{(\mathtt{?X}, \mathtt{y})\}$, however matching the pattern `\forall ?X; p(j)` against the same concrete formula results in an unsuccessful match. In this example we have added a type to the quantified variable `y` in the concrete quantified formula. In the KeY system, the quantified variables are always typed. If during a matching evaluation the typing of a schema variable and the bound term do not match, the matching is unsuccessful.

Matching Sequents

We extend the definition of $match_t$ for sequents. For this definition we will use the definition for term matching to match semi-sequents. Furthermore, we will use the definition for semi-sequents to define the matching of sequents. As patterns to match sequents in a matching expression, we use schema sequents. Schema sequents have the form $\mathrm{antec}_{Sch} \Rightarrow \mathrm{succ}_{Sch}$, where antc_{Sch} and succ_{Sch} are sets of schema formulas, also called *semi-sequents*. The sequent arrow in the schema sequent is used to indicate on which side of the sequent schema formulas should be matched.

An example for a schema sequent is `p(?X) ==> q(?X)` which matches any concrete sequent that has a unary predicate `p` in the antecedent and a unary predicate `q` in the succedent. Furthermore, the sub-terms of both need to refer to the same term. In this simple example, the schema sequent only contains one formula in the antecedent and one in the succedent. In general, the patterns for sequents may contain multiple match patterns in the antecedent and succedent as shown in the running example in Ex. 7.12.

Example 7.11. As a running example, lets consider the schema sequent `p(?X), ?X=?Y ==> p(?Y)` and the concrete sequent `p(a), p(b), a=b, a=c ==> p(c), p(d)`. Matching the schema sequent against the concrete sequent results in the set of substitutions $\{\{(?\mathtt{X},\mathtt{a}),(?\mathtt{Y},\mathtt{c})\}\}$

The function match_t will be overloaded in the following, such that we can use it for the matching of sequents. We will indicate this by the subscript *Seq* in the function's name match_{Seq}.

Definition 7.7.11 (Match Sequent). *Let* $(\mathrm{antec}_{Sch} \Rightarrow \mathrm{succ}_{Sch}) \in \mathcal{S}_{Sch}$*, with* $\mathrm{antec}_{Sch}, \mathrm{succ}_{Sch}$ *being two possible empty set of formulas of* $\mathcal{F}_{Sch}$ *and* $\mathrm{antec}_{Con} \Rightarrow \mathrm{succ}_{Con} \in \mathcal{S}_{Con}$*, with* $\mathrm{antec}_{Con}, \mathrm{succ}_{Con}$ *being two possible empty set of formulas of* $\mathcal{F}$*.*

Let $\mathrm{match}_{Seq}\colon \mathcal{S}_{Sch} \times \mathcal{S} \to \mathcal{P}(\Theta \cup \{\theta_\perp\})$ *be the function matching a schema sequent against a concrete sequent such that:*

$$\begin{aligned}&\mathrm{match}_{Seq}(\mathrm{antec}_{Sch} \Rightarrow \mathrm{succ}_{Sch}, \mathrm{antec}_{Con} \Rightarrow \mathrm{succ}_{Con}) =\\ \Downarrow (\{&m_{antec} \oplus m_{succ} \mid m_{antec} \in \mathrm{match}_{SemiSeq}(\mathrm{antec}_{Sch}, \mathrm{antec}_{Con}) \wedge\\ &\qquad m_{succ} \in \mathrm{match}_{SemiSeq}(\mathrm{succ}_{Sch}, \mathrm{succ}_{Con}))\})\end{aligned}$$

The problem of matching a schema sequent against a concrete sequent is divided into sub-problems. The antecedent of the schema sequent is matched against the antecedent of the concrete sequent. The same applies for the succedent of both sequents. We obtain matching results for both semi-sequents which may contain clashing substitutions if combined to the final result. The resulting set of substitutions is combined by building the cross product over the results of both sides and discarding all sets containing clashing substitutions using the overloaded function *clash-free composition.*

Example 7.12. Lets reconsider the last example: matching `p(?X), ?X=?Y ==> p(?Y)` against `p(a), p(b), a=b, a=c ==> p(c), p(d)`. We first match `p(?X), ?X=?Y` against `p(a), p(b), a=b, a=c` resulting in the following sets of substitutions: $\{\{(?\mathtt{X},\mathtt{a}),(?\mathtt{Y},\mathtt{b})\},\ \{(?\mathtt{X},\mathtt{a}),(?\mathtt{Y},\mathtt{c})\}\}$
We also match the succedents `p(?Y)` against `p(c), p(d)` resulting in the following set of substitution sets: $\{\{(?\mathtt{Y},\mathtt{c})\},\ \{(?\mathtt{Y},\mathtt{d})\}\}$
The results of both sides now have to be combined by building the cross product:

$$\begin{aligned}\{&(\{(?\mathtt{X},\mathtt{a}),(?\mathtt{Y},\mathtt{b})\},\{(?\mathtt{Y},\mathtt{c})\}),\\ &(\{(?\mathtt{X},\mathtt{a}),(?\mathtt{Y},\mathtt{c})\},\{(?\mathtt{Y},\mathtt{c})\}),\\ &(\{(?\mathtt{X},\mathtt{a}),(?\mathtt{Y},\mathtt{b})\},\{(?\mathtt{Y},\mathtt{d})\}),\\ &(\{(?\mathtt{X},\mathtt{a}),(?\mathtt{Y},\mathtt{c})\},\{(?\mathtt{Y},\mathtt{d})\})\}\end{aligned}$$

This cross product contains clashing substitution sets, e.g., $\{(\{?\mathtt{X},\mathtt{a}),(?\mathtt{Y},\mathtt{b})\},\{(?\mathtt{Y},\mathtt{c})\}\}$, where `?Y` is bound to `b` and `c`. Applying the clash-free composition to each pairs in the cross-product results in discarding exactly those sets and resulting in the following set: $\{\{(?\mathtt{X}, \mathtt{a}), (?\mathtt{Y}, \mathtt{c})\},\ \{\theta_\perp\}\}$. The result of the clash-free union can also contain unsuccesful matches, in this case the set has to be filtered using the *filter bottom* function.

Our matching language also allows to use a schematic sequent expression without the sequent arrow, e.g., `p(?X), p(?Y)`. In this case, we define the evaluation of the function match_{Seq} such that we match twice, once for the succedent and once for the antecedent.

Definition 7.7.12 (Match Sequent without $\Rightarrow$)**.** *Let $\{f_1, \ldots, f_n\}$ be a set of formulas of $\mathcal{F}_{Sch}$ and $s \in calS$ a concrete sequent. We overload the function* match_{Seq} *as follows:*

$$\begin{aligned}\mathrm{match}_{Seq}(\{f_1, \ldots, f_n\}, s_c) = &\\ &\mathrm{match}_{Seq}(\Rightarrow \{f_1, \ldots, f_n\}, s_c)\\ &\cup \mathrm{match}_{Seq}(\{f_1, \ldots, f_n\} \Rightarrow, s_c))\end{aligned}$$

We have defined the matching of sequent using a definition of matching semi-sequents. All elements of the schematic semi-sequent has to be matched against all elements of the concrete semi-sequent, clashing substitutions and substitutiosn containing $\theta_\perp$ have to be removed.

Example 7.13. Lets reconsider the running example (Ex. 7.12): matching `p(?X), ?X=?Y ==> p(?Y)` against `p(a), p(b), a=b, a=c ==> p(c), p(d)`. The focus of this example is on the intermediate steps of matching the semi-sequents. First, we match each of `p(?X), ?X=?Y` against each of the formulas in the antecedent, i.e., `p(a), p(b), a=b, a=c`. From these matches, we build the cross product, resulting in the following set of pairs:

$$\begin{aligned}&\{(\{(?\mathtt{X}, \mathtt{a})\}, \{(?\mathtt{X}, \mathtt{a}), (?\mathtt{Y}, \mathtt{b})\}), \quad (\{(?\mathtt{X}, \mathtt{a})\}, \{(?\mathtt{X}, \mathtt{a}), (?\mathtt{Y}, \mathtt{c})\}),\\ &\ (\{(?\mathtt{X}, \mathtt{b})\}, \{(?\mathtt{X}, \mathtt{a}), (?\mathtt{Y}, \mathtt{b})\}), \quad (\{(?\mathtt{X}, \mathtt{b})\}, \{(?\mathtt{X}, \mathtt{a}), (?\mathtt{Y}, \mathtt{c})\}),\\ &\ (\{\theta_\perp\}, \{(?\mathtt{X}, \mathtt{a}), (?\mathtt{Y}, \mathtt{b})\}), \quad (\{\theta_\perp\}, \{(?\mathtt{X}, \mathtt{a}), (?\mathtt{Y}, \mathtt{c})\}),\\ &\ (\{(?\mathtt{X}, \mathtt{a})\}, \{\theta_\perp\}), \quad (\{(?\mathtt{X}, \mathtt{b})\}, \{\theta_\perp\}), \quad (\{\theta_\perp\}, \{\theta_\perp\})\}\end{aligned}$$

The resulting intermediate set contains clashing substitutions. Applying the clash-free composition to each element of the cross product results in the set:

$$\{\{(?\mathtt{X}, \mathtt{a}), (?\mathtt{Y}, \mathtt{b})\}, \{(?\mathtt{X}, \mathtt{a}), (?\mathtt{Y}, \mathtt{c})\}, \{\theta_\perp\}\}$$

The clash-free set contains successful and unsuccessful matching results. The unsuccessful results are removed using the *filter* function $\Downarrow$. The resulting substitutions are: $\{\{(?\mathtt{X}, \mathtt{a}), (?\mathtt{Y}, \mathtt{b})\}, \{(?\mathtt{X}, \mathtt{a}), (?\mathtt{Y}, \mathtt{c})\}\}$.

Definition 7.7.13 (Match Semisequent)**.** *We define the function* $\mathrm{match}_{SemiSeq}$ *for a set of schematic formulas* $P = \{p_1, \ldots, p_n\}$ *and a set of concrete formulas* $G = \{g_1, \ldots, g_m\}$ *as:*

$$\mathrm{match}_{SemiSeq}(P, G) = \Downarrow (\{\oplus\ m \mid m \in (\mathrm{match}_t(p_1, G) \times \ldots \times \mathrm{match}_t(p_n, G))\})$$

with an extension of the definition of match_t *to match a pattern* p *against a set* $G = \{g_1, \ldots, g_m\}$ *of concrete formulas:*

$$\mathrm{match}_t(p, G) = \mathrm{match}_t(p, g_1) \cup \ldots \cup \mathrm{match}_t(p, g_m).$$

According to Def. 7.7.13, a schematic semi-sequent matches a formula set G iff every pattern p_i of the schematic semi-sequent matches a formula in G. Whereby, different patterns can match the same formula. For example, for a pattern `p(?X), p(?Y)` and a concrete formula `p(a)` we obtain the resulting substitution `{(?X,a), (?Y,a)}`. For the realization in KeY we have restricted matching the semi-sequent to allow for more meaningful match patterns. If a pattern p_i matches a formula g_j we disallow that any other pattern p_k with $k \neq i$ can be used to match the formula g_j. By disallowing such a *multi-matching*, we enable to write matching expressions that are more fine-grained, as we are now able to use the number of occurring patterns to express how many (different) formulas at least have to be present on the semi-sequent.

7.7.2. The keywords `closes` and `derivable`

We have introduced the semantics for the common constructs of our language and left out the semantics for the special `case closes` $\{$*statements*$\}$. Here, instead of matching a goal, the goal is selected, if the $\{$*statements*$\}$ interpreted on that goal, results in a closed goal, e.g., by performing a side-proof with the proof obligation from the goal that should be matched. In any case, the further interpretation of the script after the interpretation of the `closes` statement if performed using the goal from the pre-state of the `closes` statement. However, if the statements of the `closes` statement would close the goal, the body of the respective case is evaluated. As syntactic sugar respectively ease of use we have added a special case: `try` $\{$*statements*$\}$ in our realization for KPS. The `try` case is equivalent to a `closes` case where the statements after the keyword `closes` and the body of the `closes` case are identical. Therefore, the `try` case closes a goal if the body of the case closes the goal, otherwise the goal is left untouched.

The keyword `derivable` followed by a concrete term can be used instead of a match pattern. In this case the prover is called in the background with a side proof and it is evaluated in the current goal, whether the concrete term is derivable. If this is the case the newly created goal from the side proof is used for the rest of the proof and the `case`'s body is executed on the new goal, otherwise the side proof is discarded and the goal is not matched by the case containing the `derivable` expression.

7.8. Conclusion and Future Work

In this chapter we have presented a general concept for a proof script language that allows users to construct and explore program verification proofs textually. The design of the language is inspired by our observations of the user interactions in the user study presented in Chapter 5. In addition to this general concept we have presented KPS as an instantiation for a program verification system using a sequent calculus, including the formalized semantics of KPS. The general language concept, however, is not restricted to one particular setup but can easily be instantiated for other verification systems with rule-based calculi. The language and the grammar is intentionally flexible, as depending on the verification system for which the language concept is instantiated it is not known a priori which actions users may want to perform. Rather the verification system defines the actions as mutators for the language. For example in KPS, we have two different mutators defined by KeY: individual calculus rules and macros (i.e,, proof search strategies).

One feature of the proof scripting language is that it allows to flexibly match proof goals for proof construction by using special selector statements. To enable this flexibility the selector statements are built from matching expressions over the constituents of a proof goal (e.g., the proof verification condition). The language of the matching expressions depends on the structure of the proof verification conditions of a proof goal. We have introduced the matching of sequents using schematic sequents, which are syntactically similar to the schematic expressions used in KeY's taclet language [RU16]. The matching expressions in the selectors can lead to more robust scripts that are more resilient to changes in the proof input artifacts in the iterative proof process.

By design our language concept and the instantiation for KeY were developed with the purpose of constructing individual proofs, in contrast to other scripting languages which target the definition of general proof strategies. Although the language contains some constructs to allow for more general proof scripts, compared to the possibilities in general programming languages it is not as expressive. For example, in the current version of KPS we have not included goals as explicit type for variables. Rather, we consider goals as implicit data structures of the state. To be able to use a scripting language as language for general purpose proof search strategies, data structures should not be restricted and it has to be possible to exchange and save information about the whole proof state and not only about single goals. Furthermore, iterations, as well as other accessor functions over all data structures should be possible.

The matching language introduced in this chapter is dependent on the structure of the proof obligation and in the case of KPS based on the structure of sequents in KeY. For future work it remains to extend the matching language and the evaluation of the matching expressions to all structures that can occur on a sequent in KeY.

Two aspects not considered in this chapter are the typing of schema variables and matching of program structures in the sequent. Concerning the first aspect, even though typing of schema variables and term patterns is not explicitly discussed, our definition of the evaluation of match expression is also applicable for typed first-order logic. In that case the schema variables are also typed, according to the type of the matched term respectively formula. In the typed setting it is also possible to specify type-constraints for the schema variables, for example `p(...?X:int...)` which matches all predicates `p` containing at least one subterm with type `int`.

Regarding the second aspect, we defined the evaluation of matching expressions only for first-order logic constructs so far. As we want to use it in the context of program verification using the KeY system, we also have to consider program modalities and updates in Java DL formulas. In the current version these elements are opaque, i.e., it is not possible to match against these constructs in the current version. To be able to match updates and modalities the language of the match patterns has to be extended for the substructures present in modalities (e.g., Java programs) and updates (e.g., the assignments in the updates and the structure of sequential and parallel updates). For the matching of Java statements one solution would be to allow regular expressions over strings to match programs in modalities. For matching updates the evaluation of terms has to be extended by defining how updates are matched.

In a case study (described in Ch. 8) a first evaluation of KPS has been performed which has shown that the primitives of the language are in general sufficient to construct proofs similar to proof construction using direct manipulation in KeY. However, to obtain a more conclusive result a larger case study has to be performed that takes the proof search itself into account. Furthermore, the language contains primitives for proof exploration. The usage and the usefulness of these primitives have to be evaluated in a user study.

Proof construction in deductive program verification is also characterized by inspection of the proof states if a proof attempt is not successful. To allow for support in developing proof scripts and inspecting proof attempts a tool is necessary. In Chapter 8 we will introduce one possibility for a tool support.

7.9. Related Work

Many general purpose proof assistants using higher-order logic feature text-based interaction (e.g., Isabelle/HOL [NPW02] and Coq [BC04]). They mostly use an implicit proof object, where the user can only inspect the goal states but not the intermediate atomic proof states. Proofs are performed either using the system's programming language or by using a language that directly communicates with the system's kernel and builds an abstraction layer on top of the kernel. All such languages have in common that they serve as the only interaction method. Therefore, care has been taken to design proof languages that are both a human-readable input method for proofs and a proof guidance language with which it is possible to control the prover's strategies (also called tactics). Isar [Wen99] is the most prominent state-of-the-art language that serves these purposes. Proof exploration can be done by providing proof commands or by postponing proof tasks using a special keyword.

On top of the proof language the aforementioned systems offer languages that allow to write strategies (e.g., Eisbach [MMW16] for Isabelle or MTac [Zil+13] for Coq) to enable users to program their own tactics tailored to the proof problem. *Proof-Script* [OSF16] is a proof language inspired by the programming language B-17 and the proof language Isar. It is intended for the use in collaborative proving in *Proof-*

Peer and is designed to overcome the language stack present in the aforementioned systems, providing one language that fits all purposes. All these languages contain mechanisms for matching terms and formulas to select proof goals for rule application. We refer to [MMW16] for an overview of proof languages. Another example for a tactic language is Bellerophon [Ful+17], which is a tactic language for hybrid systems verification. It is a programming language which can be used to automate individual proofs as well as proof search procedures in KeYmaeraX.

There also exist approaches to debugging proof tactics and gain more insight. For example, Tinker2 [LLG16] is a graphical tool for inspecting the flow of goals in proof tactics. And Hentschel [Hen16] applies debugging concepts to the verification domain in his symbolic execution debugger built into KeY. This debugger supports the user in case the cause of a failed proof attempt is a mismatch between the program and its specification. However, it does not give significant insights if the proof fails because of insufficient user guidance.

8. Integrating Direct Manipulation with Script-Based Interaction for Program Verification

Contents

In this chapter we will demonstrate how to integrate direct manipulation and script-based interaction to leverage the advantages of both interaction styles for interactive program verification. In particular, we will present a realization of this combination by integrating text-based interaction into the KeY system using KPS as a proof script language.

The combination results in the ability to construct proofs using different interaction styles, which follows the usability principle of *flexibility*. This principle was also identified by Easthaughffe [Eas98] to be an important feature for theorem provers [Eas98]. More precisely, the integration of interaction styles adheres to the principle of *substituitivity*, as there are two exchangeable ways in expressing the user interaction in program verification. Our goal is to support users in: (1) finding and repeatedly applying suitable proof commands, (2) switching between different contexts, like modification of the underlying problem and proof construction, and (3) finding the right steps to successfully continue in the proof process.

Additionally, by introducing this text-based interaction style, proof construction becomes similar to regular software debugging, as we can give an analogy between finding bugs in programs and finding the cause for failed proof attempts. This analogy allows us to adapt well-known concepts from software debugging for the user interface and the proof process.

We will first describe a concept for analyzing failed proof attempts that is based on the analogy between finding the cause for unfinished proofs and software-debugging. We will then present our integration of a text-based with a direct manipulation interaction style for interactive program verification which takes the debugging metaphor into account – first as a concept, followed by a concrete realization for the KeY system.

8.1. Debugging Proof Attempts

The main goal of program debugging systems is to support the software engineer in finding defects in a program. In addition to defect localization, it is also possible to use debugging systems to understand the functionality of a program. Analogously, the main goal of a program verification system is to support the user in completing a verification attempt by either allowing the user to identify where the program does not meet its specification or, alternatively, where further user guidance is needed for the automation to be able to complete the proof. This support should enable the user in transforming a proof sketch for the correctness of a program to concrete steps to advance the proof, as well as to find mistakes in the formalization and to check the proof.

A verification attempt is not successful, for example, if the user receives the impression that automatic proof search does not advance the proof or a resulting proof state shows that continuing the proof will not lead to a closed proof. In such a situation the user needs to determine the cause for the failed attempt. We distinguish three kinds of activities the user may perform in this case: (a) the user proceeds in the proof process by changing the proof state target-oriented, (b) the user tries to comprehend the proof state without changing the proof state and (c) the user explores the proof state, i.e., the proof state is changed by the user with the goal to comprehend the proof state and may be changed back to the state before the change later in the proof process.

These activities are similar to finding and correcting defects in a program: (a) the software engineer changes the program state by correcting a defect in the source code, (b) the engineer reads the source code to comprehend it and (c) the engineer changes the source code, e.g., during the debugging phase, and executes the program again.

There is no clear a priori distinction which actions can be categorized as exploration (c) and which actions belong to the activities comprehension (b) and continuation (a) – rather, the context in which these actions are used and the intent of the user play a large role in determining the category of an action. One example for evidence that suggests an action belonging to proof exploration is that the user performs activities that change the proof state and which the user reverts afterwards.

In the following section, we will use the analogy between program verification and software debugging to support the user of a program verification system in making

Table 8.1.: Analogies between program debugging and debugging failed proof attempts.

proof debugging	↔	program debugging
proof script	↔	program source code
script state (incl. proof state)	↔	program state
sources and open proof goal(s)	↔	program input
proof tree	↔	traces of all threads
proof branch	↔	trace of an individual thread
partial proof	↔	traces of an incomplete program run
completed proof	↔	traces of a successfully terminating program run

the decision about the next step in the proof process by adapting functionalities from software debugging to program verification.

Within the program verification process, there are two main perspectives a user may take: either the perspective of a software engineer, i.e., by thinking in terms of the input artifacts (program together with its annotations) and the program execution, or the user takes a perspective where the logical argument on the proof obligation level is in focus, i.e., formulas and possible deductions are relevant. We believe a user may have both perspectives and in the following we concentrate on the latter perspective, complementing the work done by, e.g., [Hen16], where support for the first perspective is presented. As a consequence, the user needs the possibility to identify the reason for an unfinished verification attempt in terms of the logical proof.

To be able to adapt concepts from software debugging to program verification we will introduce the concepts of debugging software systems and present for each concept the similarities or differences to program verification.

8.1.1. Analogy between Programs and Proof Scripts

Scripts, formulated in a scripting language like the one presented in Chapter 7 can be considered to be programs that construct (partial) proofs for a given proof obligation: They take the initial proof goal as input and derive a set of new goals. The input goal is successfully proved if the derived goal set is empty. The similarity between proof scripts and imperative programs allows us to draw an analogy between implementing and debugging programs on the one hand and coming up with proof scripts and analyzing failed proof attempts on the other. The main analogies between the two processes are summarized in Table 8.1.

Note that evaluating a proof script corresponds to executing a *multi-threaded* program because of the proof-forking nature of some proof commands (which implement case distinctions). Proof commands on different open goals can be handled independently and in parallel. In that sense, executing a `cases` command (see Ch. 7.3) corresponds to forking threads, which are joined again when the `cases` command terminates. The proof tree that is built when executing a script corresponds to the set of traces of all threads when executing a program.

However, there is also an important difference between proof scripts and general programs: The result of a successful proof script evaluation, i.e., a closed proof, is known a priori (the empty set of goals). Since no output object needs to be constructed, in many cases predefined operations lead to success. This is the reason why users often at first follow a try-and-error approach: Just using the `auto` command for automatic proof search works for many simple proof goals – which is not possible for arbitrary simple computation tasks as these differ in their expected outputs.

8.1.2. Analogy between Debugging and Failed Proof Analysis

Software debugging is the analysis process of understanding unexpected program behavior, localizing the responsible piece of code, and mending it. Typically, a concrete run of the program exposing the defect is analyzed using specialized software (a debugger) which supports the user in the process by various means of visualization and abstraction. The features help the user comprehend and explore both individual program states at various points of the execution and paths through the program taken by the execution.

Powerful modern debugging tools also allow the engineer to modify an intermediate system state (e.g., by changing the values of variables) to conduct what-if-analyses which help them understand and explore the system.

When mechanizing a formal proof, the user often has the main arguments of an abstract proof plan in mind which (supposedly) lead to a closed proof. However, this plan is often at a high abstraction level such that it cannot be transformed directly and easily into proof script commands; the user has to refine the proof plan first to be able to formulate it as a proof script. Especially in early stages of a proof process, the evaluation of a proof script is likely to fail. The typical reasons for a failed proof attempt include that auxiliary annotations (such as loop invariants) may be insufficient, that there may be defects in the source code or the specification, or that the proof script itself may be misleading or not detailed enough. Eliminating all such deficiencies is an iterative process, which may also affect other proofs of the same overall verification task (since there are interfaces and interdependencies between system components even if they are verified separately).

When the evaluation of a proof script does not lead to a closed proof, the user needs to be able inspect the intermediate and final proof states in order to *understand* the undesired behavior. This process involves to *localize* the responsible part of the proof and to identify the type of failure: Does the verification system require more or better guidance? Is there a defect in the program, the specification, or the proof script?

The same kind of questions arise in conventional program debugging (Are the data as expected at this point? Is the next statement in the program the correct one? Are all parameters to a routine call correct?). Hence, the user needs tool support to decide these questions also for debugging proof scripts. Similar inspection possibilities are required to come up with actions in the proof process. It must be, in particular, possible to link proof states to commands in the proof script and to the user's mental proof plan. To find a suitable course of action, the user needs to have means to *explore* the proof state and to test hypotheses about the cause of failure and about effects of next steps to the proof.

8.1.3. Adoption of Program Debugging Methods for Proof Debugging

The analogy between proof scripts and programs, and the similarities between the software debugging process and the process for the analysis of failed proof attempts allow us to adopt well-known techniques from software debugging to the debugging of (failed) proof attempts. In this Section, we focus on user support for the activities of localization, comprehension, and exploration. Additionally, we adapt the presentation of program states for script states, allowing a detailed inspection. A description of the realization of the concept and a screenshot of our prototype[1] (based on the KeY system) is presented in Sec. 8.2.

State Presentation. Program states in software debugging may be very complex, e.g., a program state may contain many instances of different data structures and different variable scopes – at the same time, often only a small portion of the state is actually relevant for the debugging task. To support the user in inspecting and understanding a state, debugging systems present the state's information in a structured manner, e.g., a hierarchical representation of data structures and their contents.

Our concept for proof states includes a structured presentation and functionalities for inspecting the state similar to program debugging systems. As a requirement for our concept, we have identified the following parts of a state that should be visualized in isolation: (a) the proof tree with a visual highlight of the current proof node (i.e., the node containing the open goal to which the currently active proof command is being applied), (b) sequent of the current proof node (i.e., the current open goal), (c) the currently active proof command in the script together with its position in the script, (d) the path in the program that corresponds to the currently selected proof branch, and (e) the values of all local variables in the script state.

Localization. To support the user in localizing the cause of a defective behavior, debugging systems provide *breakpoints*. These allow the user to pause execution and inspect the program execution in detail when a program location is reached.

In the setting of program verification, defective behavior corresponds to a proof with open goals, and the user is mostly interested in understanding the open goals. In our concept, using direct manipulation interaction on the explicit proof object, users have the flexibility to navigate in the proof tree in both directions: from the root to the open goals (leaves) and backwards from the leaves to the root. The user can follow two possible strategies: (a) Inspecting an open goal that contains unexpected formulas or terms and performing a backwards search to localize where this information was introduced into the proof. (b) Starting from a familiar and expected state and tracing the proof in a forward fashion.

In order to support these strategies, we adopt the idea of breakpoints in two ways: *regular breakpoints* and *(reverse) conditional breakpoints*.

[1] `http://formal.iti.kit.edu/psdbg`

A *regular breakpoint* is a syntactical marker that represents a location in the proof script. If the execution of the proof script is started with enabled breakpoints and reaches the breakpoint, execution is stopped and the current proof state is presented to the user. Similar to program debugging, breakpoints may be conditional. Such *conditional breakpoints* include Boolean expressions indicating that execution shall only stop if conditions on the state evaluate to true when the breakpoint is reached.

For backwards search, we provide *reverse conditional searchpoints*, which consist of a Boolean condition and a goal node. While breakpoints are the endpoint of a search in the script's execution, searchpoints are the starting point for a search on the (partial) proof after the script's execution is paused or terminated. The backwards search in the (partial) proof – from the searchpoint towards the root node – stops at the first intermediate proof node for which the condition is evaluated to true and the corresponding proof state is presented to the user.

Conditions in breakpoints and searchpoints can be Boolean expressions from the script language, in particular all matching conditions can be used here. This design allows the user to find states where certain formulas are introduced into the sequent or nodes in the proof tree where certain rules are applied. Breakpoints can also be used to select states where the complexity or number of formulas in the sequent reaches a certain threshold.

Stepping, Tracing, and Comprehension. Once the user has located an entry point from where to perform a more detailed inspection, the next activity is to step-wise retrace what state changes are made by the proof script. In some program verification systems, such as the KeY system, retracing of state changes is already possible on the visual representation of the proof object. The granularity of the proof steps however, is rather fine-grained and the process may become tedious due to the large number of proof steps the user has to navigate through. To simplify this process, the proof script debugger allows the user to limit the inspection to interesting parts of the script (step-into) and to omit the details of subscripts that are deemed irrelevant (step-over). Stepping in our concept can be performed in a forward fashion, similar to regular debugging systems and in a backwards fashion on the performed proof, similar to offline debugging systems. The step-wise retracing allows the user to comprehend the effects of proof commands and subscripts and the creation of proof goals.

Expression Evaluation. Software debugging systems support the user task of forming hypotheses about the cause of a defect in a program by allowing the evaluation of user-provided expressions in the current state. A functionality for proof debugging corresponding to expression evaluation is to allow the user to provide a set of formulas, which may or may not be a subset of formulas present in the proof state, and to evaluate whether these formulas are derivable in the context of a node in the proof tree.

One may use external solvers or verification systems to determine whether the set of formulas is satisfiable or not and to get a model in the first case. This is particularly helpful in cases where the size of the sequent prevents the underlying proof system from finding a counterexample.

Changing the State: "What-if"? We adopt the idea of allowing the user to explore the behavior of the proof script by actively changing the proof state in debug mode. Similar to the exploration of program states, the changes performed during such an exploration should be performed in an own environment (e.g., on a copy of the proof object or proof state in an own window) and easily reversible automatically, if the user leaves this environment.

Thus, the user may gain information about which changes are necessary to advance the proof search. In a second step, this knowledge may then be used to, e.g., analyze whether the origin of the part of state that was changed (e.g., the precondition of the program) has to be adapted. In Chapter 9, we will present such a *proof exploration* functionality for the KeY system in detail.

Hot-Swapping. A further element of the proof debugging concept is to allow *hot swapping*, i.e., the user can change parts of the proof script while the script is executed in debug mode, in order to explore hypotheses about how the proof construction can proceed successfully. Hot-swapping in the proof script is the counterpart to a *what-if* exploration, where the proof state may be changed.

8.2. Integrating Direct-Manipulation and Script-Based Interaction

We developed a concept, together with a prototypical realization, called *Proof Script Debugger* (PSDBG), for a new user interface for the KeY system that integrates both script-based and direct manipulation interaction styles. This interaction concept is based on the proof debugging metaphor described in Section 8.1.2.

In the following we will present the details of the ideas and principles used for the user interface and we will give an example how interaction using PSDBG may be performed to construct proofs.

Using a preexisting verification tool as an integration platform always imposes additional constraints on the realization of the user interface, as well as restrictions on the type and amount of information that is accessible through the tool's API. One restriction is for example that the integration of new views into a preexisting user interface that is inflexible and contains already many views may result in a cluttered user interface. As an alternative, we will present a concept for seamless interactive verification that is only loosely tied to existing user interfaces of the underlying verification system and its implementation.

Integrating a text-based proof construction interface to KeY involves adding a new view containing the proof script, alongside the already existing views that show the proof state in KeY (i.e., the proof tree and sequent view). One challenge in displaying these different views to the user is that each view has to have a clear purpose that is easily recognizable by the user, both conceptually and visually.

The contents of the view have to match the user's expectations and mental model, as otherwise, more views may be a source for confusion. Having many views with different interactions demands the views to be consistent in order to follow the usability principle of *consistency* and therefore lead to fewer surprises for users when switching views.

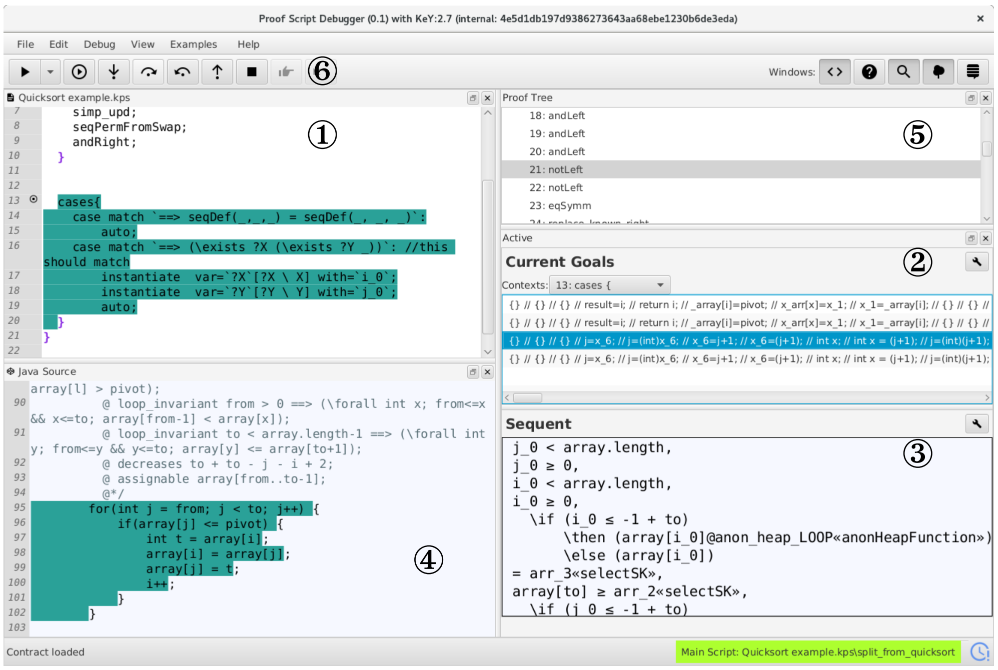
We claim that this view consistency is a central usability aspect to achieve, resulting in the following invariant during proof construction: the proof script, as the textual representation of the proof, and the view onto the underlying proof state have to always show the same state.

Furthermore, to be able to build up a mental model or to examine whether the own mental model is consistent with the proof shown in the system, the user should be able to retrace what has happened in the proof process. For this we included the possibility to step through the proof script's execution in a forward and a backward fashion from the concept described in Sect. 8.1.3. The consistency invariant should also lead to a consistent view for the user while stepping through the proof script. Violations of the invariant should be limited to user-induced editing of the script text. In the following, we present our approach of combining KeY with a text-based interface in more detail.

Different Projections of the Proof State. One guideline for theorem prover interfaces is that multiple views support the user in the complex task of theorem proving [Eas98]. Following this guideline, our user interface includes different projections of the proof state, similar to software debugging systems, where different views are used to present a potentially complex and large program state to the user. With the help of these views users can focus on their preferred representation of information in the current proof situation. To enable the user to flexibly switch contexts as necessary and to adhere to the usability principle of *customizability*, users are able to request these views (or remove them from the screen) at any time to choose the amount and type of information suitable for the current task of the verification process.

Program verification proofs contain information about the program and the mathematical logical proof interwoven. In large proofs it is challenging for the user to differentiate between both kinds of information. In our user study (see Chapter 5) for example, we observed that the participants searched for the relation between the proof state in KeY and the annotated program in a text editor for the proof construction task as well as for the task where the proof state had to be explained to the moderator. To handle this challenge, KeY implements proof search strategies that focus either on performing logical steps or on the steps that perform symbolic execution. However, we additionally support the separation by using separate views. Structuring the proof state into different views allows the user to focus on specific parts of the proof problem – either on the general structure of the proof in relation to the program or on single proof goals. The different proposed views we consider for program verification maintain the relation between the program and the proof obligation. This contributes to the principle of displaying only relevant information to the user [Eas98] – by separating the interwoven information the user can now focus on the relevant information by choosing the preferred view.

The standard view on the proof state in our concept (see Fig. 8.1) contains the proof script with a visual highlight of the next statement to be executed (①) and a list of all open proof goals (②) with a representation of the proof obligation of the


Proof Script Debugger (0.1) with KeY:2.7 (internal: 4e5d1db197d9386273643aa68ebe1230b6de3eda)
File
Edit
Debug
View
Examples
Help
⑥
Windows:
Quicksort example.kps
①
Proof Tree
⑤
Active
Current Goals
②
Contexts: 13: cases {
Java Source
④
Sequent
③
Contract loaded
Main Script: Quicksort example.kps\split_from_quicksort

```
    simp_upd;
    seqPermFromSwap;
    andRight;
  }


  cases{
    case match `==> seqDef(_,_,_) = seqDef(_, _, _)`:
         auto;
    case match `==> (\exists ?X (\exists ?Y _))`: //this
should match
         instantiate  var=`?X`[?X \ X] with=`i_0`;
         instantiate  var=`?Y`[?Y \ Y] with=`j_0`;
         auto;
  }
}
```

```
18: andLeft
19: andLeft
20: andLeft
21: notLeft
22: notLeft
23: eqSymm
```

```
{} // {} // {} // result=i; // return i; // _array[i]=pivot; // x_arr[x]=x_1; // x_1=_array[i]; // {} // {} //
{} // {} // {} // result=i; // return i; // _array[i]=pivot; // x_arr[x]=x_1; // x_1=_array[i]; // {} // {} //
{} // {} // {} // j=x_6; // j=(int)x_6; // x_6=j+1; // x_6=(j+1); // int x; // int x = (j+1); // j=(int)(j+1);
{} // {} // {} // j=x_6; // j=(int)x_6; // x_6=j+1; // x_6=(j+1); // int x; // int x = (j+1); // j=(int)(j+1);
```

```
array[l] > pivot);
          @ loop_invariant from > 0 ==> (\forall int x; from<=x
&& x<=to; array[from-1] < array[x]);
          @ loop_invariant to < array.length-1 ==> (\forall int
y; from<=y && y<=to; array[y] <= array[to+1]);
          @ decreases to + to - j - i + 2;
          @ assignable array[from..to-1];
          @*/
        for(int j = from; j < to; j++) {
            if(array[j] <= pivot) {
                int t = array[i];
                array[i] = array[j];
                array[j] = t;
                i++;
            }
        }
```

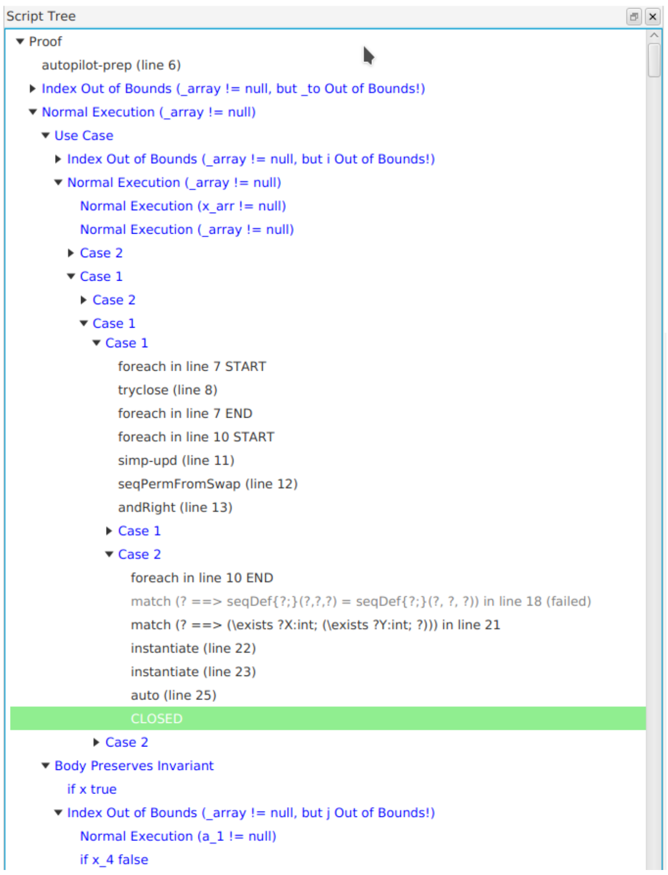
```
j_0 < array.length,
j_0 ≥ 0,
i_0 < array.length,
i_0 ≥ 0,
  \if (i_0 ≤ -1 + to)
      \then (array[i_0]@anon_heap_LOOP«anonHeapFunction»)
      \else (array[i_0])
= arr_3«selectSK»,
array[to] ≥ arr_2«selectSK»,
  \if (j_0 ≤ -1 + to)
```

Figure 8.1.: Screenshot of our proof debugger prototype based on the KeY system. On the left ① is the proof script editor (in this case containing the script for the verification of an InsertionSort implementation); the currently active proof command is highlighted. In ②, the open goals of the current proof state are listed; in this screenshot, the third goal is selected. Below, the sequent of the selected goal is shown ③. The source code panel ④ shows the Java program and highlights the symbolic execution path traversed for the selected sequent. The proof tree for the open proof is shown in ⑤. On the left-hand side of the toolbar ⑥ UI elements for invoking the different possibilities to execute the script and for stepping through the proof script are shown. The buttons on the right-hand side of the toolbar enable to hide or show the different views.

currently selected proof goal (③). A view on the program code with a visual highlight of the program statements that are covered by the proof obligation (④) is available. In addition to these views, the users are able to view the full proof tree as in KeY (⑤). Manipulating this view, e.g., by hiding and expanding of proof nodes and sub proof trees, is necessary for handling the sizes of proof trees occurring in program verification tasks.

After a first evaluation (see Sect. 8.3), we additionally devise a view showing a tree representation of the script combined with the proof tree to allow a visual combination of the underlying proof structure and the script execution. Nodes in this tree are either statements from the script, with a reference to their location in the script (line number) or branching nodes from the underlying proof tree. Furthermore, proof commands that were not applicable are marked. This view is depicted in Figure 8.2.

Changes need to be propagated to all views, to maintain the principle of *consistency*, when displaying different views. The proof tree as well as the goal list have to be advanced when the script is executed. In addition, when using direct manipulation for proof construction the actions performed there need to be reflected in the script text.



```
Script Tree
▼ Proof
    autopilot-prep (line 6)
  ▶ Index Out of Bounds (_array != null, but _to Out of Bounds!)
  ▼ Normal Execution (_array != null)
    ▼ Use Case
      ▶ Index Out of Bounds (_array != null, but i Out of Bounds!)
      ▼ Normal Execution (_array != null)
          Normal Execution (x_arr != null)
          Normal Execution (_array != null)
        ▶ Case 2
        ▼ Case 1
          ▶ Case 2
          ▼ Case 1
            ▼ Case 1
                foreach in line 7 START
                tryclose (line 8)
                foreach in line 7 END
                foreach in line 10 START
                simp-upd (line 11)
                seqPermFromSwap (line 12)
                andRight (line 13)
              ▶ Case 1
              ▼ Case 2
                  foreach in line 10 END
                  match (? ==> seqDef{?;}(?,?,?) = seqDef{?;}(?, ?, ?)) in line 18 (failed)
                  match (? ==> (\exists ?X:int; (\exists ?Y:int; ?))) in line 21
                  instantiate (line 22)
                  instantiate (line 23)
                  auto (line 25)
                  CLOSED
            ▶ Case 2
    ▼ Body Preserves Invariant
        if x true
      ▼ Index Out of Bounds (_array != null, but j Out of Bounds!)
          Normal Execution (a_1 != null)
          if x_4 false
```

Figure 8.2.: Visualization of the script as tree. The figure depicts a part of the script tree of the script that is used to prove the `split()` method of the Quicksort implementation. Branching labels of KeY are colored blue, and matching expressions that did not match are colored gray with an indication that the expression failed. Each node containing a script statement also has a line number next to it, indicating the statement that was executed.

Focusing on Details of the Proof State. Besides being able to switch representations of the proof state to the most suitable for the problem at hand, our approach also allows for a drill-down focus (as we were able to observe such a focusing during our user study). We consider the proof script to be the central proof state representation which serves as a more abstract alternative to the proof tree. Details, such as applications of single calculus rules that were performed by the prover's strategies, are hidden under proof commands with the name of the invoked strategies.

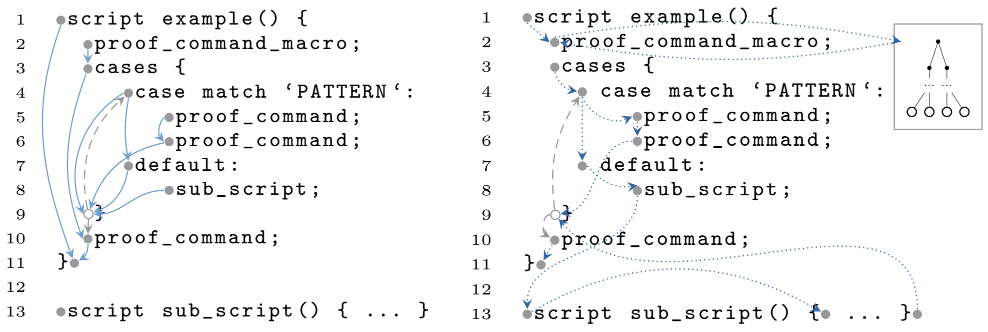
When proof attempts fail, users need to find the cause for the failure. To achieve focusing on details, we adapt stepping functionalities from software debugging, where users can step into methods of a program's execution in case detailed insight is needed. Similar to software debugging, we implemented the following ways of retracing the proof construction: step-over, step-over-reverse, step-into and step-into-reverse. We will explain the behavior of the different stepping functionalities briefly with the help of a simple abstract proof script, which is shown in Fig. 8.3 on page 208. In this figure, the step-wise execution of the script is illustrated by introducing so-called script execution pointers, indicated by a small gray circle throughout the script. In step-wise debugging, starting from a script execution pointer, statements are executed until the next pointer is reached where the script execution is to be paused according to the stepping strategy that was invoked. In Figure 8.3 (a) we have depicted the step-over function and in Figure 8.3 (b) the step-into function.

Intuitively, the step-over function keeps the script execution pointer at the same syntactic nesting depth level in the step-wise execution of the script with one exception: if the function is invoked in front of the last statement on one level, the execution pointer is set in front on the statement of the next higher nesting level (if such a level exists).

For example, when stepping-over the script header in line 1 of Fig. 8.3 (a), the execution pointer is put to the end of the script body in line 11 and thus the script is paused right at the end of the script. Stepping over proof commands or calls to subscripts sets the execution pointer in front of the next statement on the same nesting level (for example the step-over from line 5 to line 6) and the proof command or subscript is executed. Similarly to stepping over the script header, when invoking the step-over function in front of the *last* proof command in line 10, as there is no statement on the same nesting-depth level, the pointer is set to the end of the script after the proof command was executed.

Intuitively, when the statements `foreach` and `cases` are executed, their effect is an iteration over all goals of the state just before the `foreach` or `cases` statement. For the statement `repeat`, states are iterated over until no state change results after executing the body of the `repeat` statement. Stepping over these three statements is similar to stepping over proof commands, e.g., stepping over a `cases` block in line 3 places the execution pointer after the `cases` block in line 9.

For the bodies of the script statements `foreach`, `cases` and `repeat` the behavior of the step-over function depends on the script state. In particular, for the body of the `cases` and `foreach` statements it depends on whether there are still goals left to iterate over. This decision point is depicted in Fig. 8.3 (a) with a gray, non-filled circle in line 9, where the two gray arrows indicate the two possible outcomes and the



```
1  script example() {
2    proof_command_macro;
3    cases {
4       case match `PATTERN`:
5          proof_command;
6          proof_command;
7       default:
8          sub_script;
9    }
10   proof_command;
11 }
12
13 script sub_script() { ... }
```

(a)

```
1  script example() {
2    proof_command_macro;
3    cases {
4       case match `PATTERN`:
5          proof_command;
6          proof_command;
7       default:
8          sub_script;
9    }
10   proof_command;
11 }
12
13 script sub_script() { ... }
```

(b)

Figure 8.3.: Figure (a) depicts the step-over functionality and in Figure (b) the step-into functionality is shown. A native proof command is denoted with `proof_command_macro`, proof commands representing calculus rules are denoted with `proof_command` and calling subscripts is denoted with `sub_script`. The gray filled circles denote the positions where the execution pointer may be. The blue arrows indicate the transition of the pointer depending: blue dotted arrows indicate the transitions invoked by the step-into function, blue bold arrows indicate the step-over transitions. The non-filled circles indicate positions where the execution pointer is never actually set two, but from this position on the pointer is directly set to the end points of the gray arrows, depending on the current state.

corresponding placement of the execution pointer. From this decision point the next position for the execution pointer can either be in front of the first statement of the body of the `cases` or `foreach` statements to allow processing further goals, or, if there are no more goals to handle, the execution pointer is placed one nesting level above the `cases` or `foreach` statements.

For example, stepping over the `case` block in line 4 has two possible results: If the current goal is matched by the matching expression, the execution pointer is set to the end of the `cases` in line 9 and the body of the respective `case` is executed. Otherwise, the execution pointer is placed in front of the next `case` block in line 7 and the body of the `case` is not executed. When execution reaches the end of the `cases` block in line 9 a decision is made: if still goals have to be handled by the `cases` the execution pointer is directly placed in front of the first `case` of the `cases` block. Otherwise the pointer is set to the statement after the `cases` in line 10.

Also if the execution pointer is in front of the last statement in a `case` block (e.g., in line 6 and line 8 of Fig. 8.3 (a)) and the step-over function is invoked, the execution pointer is set to the decision point at the end of the `cases` in line 9.

Compared to step-over the step-wise execution via step-into is far more fine-grained. Intuitively, the step-into function moves the script execution pointer down one nesting level, e.g., when starting the step-into function at a script header the execution pointer is placed in front of the first statement of the script's body and execution is paused at this position.

Due to this behavior one major difference between step-over and step-into is the effect of the step-wise execution of proof commands that correspond to proof search strategies (such as a macro step in KeY as shown in line 2 of Fig. 8.3 (b)). Stepping into these proof commands results in a view that displays a proof subtree, containing all calculus rules applications that were performed by invoking the macro step (depicted by the proof tree pictogram on the right in Fig. 8.3(b)). In comparison, stepping into proof commands that correspond to *single* calculus rules has the same effect of a step-over, i.e., the execution pointer is set in front of the next statement on the same nesting level, e.g., in lines 5 and 6.

For the script statements that iterate over goals or states (`repeat` and `foreach`), respectively select goals (`cases`), the behavior of the step-into function is state dependent, and in some cases identical to the step-over function. Invoking the step-into function in front of a `cases` (line 3) places the execution pointer in front of the first `case` (line 4). This behavior is similar for stepping into `repeat` or `foreach`.

Stepping into a `case` is also state dependent and can have two results: if the match expression of the `case` is evaluated to true in the current goal (e.g., in line 4 in the example of Fig. 8.3 (b)), or it is a `default` case then the execution pointer is set to the first statement in the `case` (line 5 in our example). The other possibility is that the matching expression is evaluated to false and the execution pointer is set to the next case, resulting in the same behavior as stepping *over* the `case`.

If the step-into function is invoked in front of the last statement of a nesting level, the execution pointer is set to the next statement one nesting level above (if such a statement exists). For instance, when invoking step-into in front of the last proof command in the script in line 10, the execution pointer is set to the end of the script in line 11.

Stepping into a subscript (e.g., in line 8 of the example) results in placing the execution pointer in front of the subscript header (line 13). If the execution pointer is at the end of the subscript and the function step-into is called, the execution pointer is placed after the original call statement responsible for the subscript invocation. The exact position of the execution pointer that results after subscript execution is identical to the position that would result after stepping-into a single proof command.

Similar to the stepping over behavior, the stepping into behavior of a `case` has the same decision point in line 9. If the execution pointer reaches the decision point, the pointer is set to the first `case` if still goals are left to iterate over, otherwise the pointer is set to the first statement after the `cases` block.

The *step-into reverse* and *step-over reverse* functions are the inverse version of the step-into respectively step-over function. In Fig. 8.3 the reverse functions correspond to inverting the direction of the arrows. In our approach users can step (reverse) into proof commands to retrieve more details about the performed proof steps. Additionally, users can inspect the sequent of each proof node by selecting an executed script statement and requesting the functionality for further inspection while using the stepping functions and while executing the proof script. If the user chooses to further inspect states, a new tab is added in the goal and sequent view that contains the state after executing the selected statement with all goals present in that state.

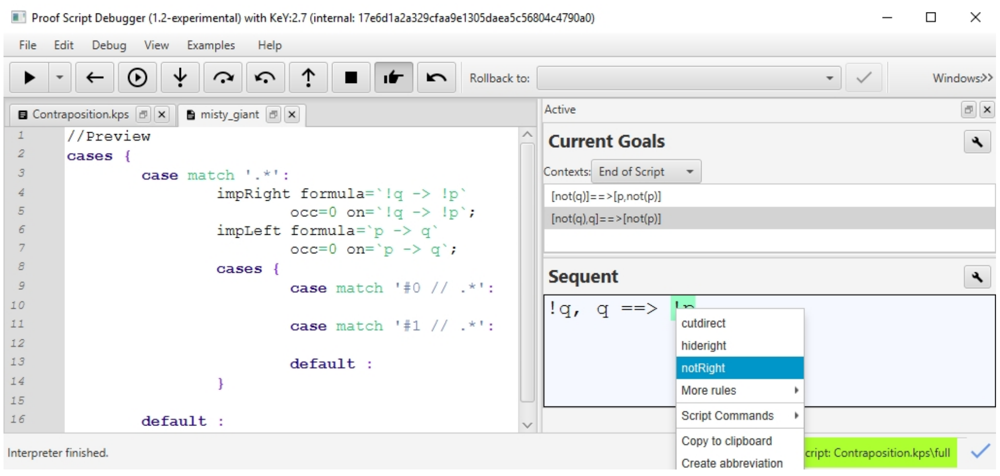
Switching Interactions. We combine two different interaction styles which can only be successful if the transition between them requires only little effort for the user. One disadvantage of direct manipulation is that persistence of interactions is not directly supported. We counteract this disadvantage with the script serving as a log of the user's interactions and adhering to the invariant for a consistent state. Our approach has two *modes* which may be selected by the user: the *script mode* and the *point-and-click mode*. Switching between writing the script and interacting on the open goals works as follows: In the point-and-click mode a new temporary script (see Fig. 8.4) is created which contains already pre-computed selectors for all open goals (noted down as `cases` statements) in the script. Users can now interact on the sequent like in KeY using point and click interaction as well as retrieving applicable rules as suggestions in a context menu. Each rule application is added to the corresponding `case` in the script text. When leaving the interactive mode the user can choose to either discard the script, e.g., in case users tried to explore the proof state and this proof exploration was not successful, or to add the generated script to the main script before the interaction step. This especially allows easily reversible actions and proof exploration.

We do not allow adding these scripts generated by point-and-click action inside (partially) executed loop constructs such as the `repeat` statement to avoid user confusion: to add an interactive proof script inside a loop construct, the original loop would have to be replaced by three parts: the loop iterations executed so far, the newly generated interactive script and a copy of the original loop.

In the script-mode the proof script can be written and executed step-wise. To determine which proof commands are possible, the user may click onto formulas which results in suggestions for rule application that need to be manually added to the script. Our feature of generating proof scripts from direct manipulation user interaction is comparable to the tactic extraction found in KeYmaeraX [MP16].

Further Support for Proof Construction Adhering to the principle of *least effort*, which was instantiated for theorem prover interfaces by Völker [Völ03], text-editing support, such as it is common in IDEs and note-taking programs, needs to be offered to the user. In the case of scripting program verification proofs, syntax highlighting as well as auto-completion for the variety of proof commands needs to be available to adhere to the principle of *recognition rather than recall*.

As further support, especially for the constructs in KPS, our approach contains aids for finding matching expressions for the different proof cases. This support was identified and realized in the course of an evaluation of KPS and PSDBG [Luo18] (see Sect. 8.3 for details): the sequent matcher (see Fig. 8.5(a)). In this window the user can enter a match expression and retrieve information which goals are matched by the entered matching expression. Furthermore, similar to the state inspection in debugging systems, we also provide a view onto the current variable values of a selected goal (see Fig. 8.5(b)). Both kinds of information are relevant for the user when coming



```
//Preview
cases {
        case match '.*':
                impRight formula=`!q -> !p`
                        occ=0 on=`!q -> !p`;
                impLeft formula=`p -> q`
                        occ=0 on=`p -> q`;
                cases {
                        case match '#0 // .*':

                        case match '#1 // .*':

                        default :
                }

        default :
```

```
[not(q)]==>[p,not(p)]
[not(q),q]==>[not(p)]
```

```
!q, q ==> !p
```

```
Interpreter finished.
```

```
cript: Contraposition.kps\full
```

Figure 8.4.: Interactively advancing the proof script.

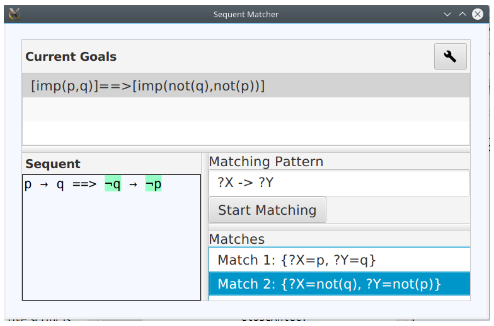
up with possible matching expressions, either over the script variables, or the sequent.

Support For Proof Exploration. Based on a first experiment and on the analogy to software debugging, we also devise means for exploring proof states. Besides the proof exploration facilities of KPS and the facility to use point-and-click in the interactive mode and to dispose the temporarily created script if not needed, our concept foresees further user support: the user should be able to continue a proof using text-based interaction to explore how it develops and being able to easily discard the written script if the result seems not promising.

In the course of the evaluation *safepoints* have been introduced as exploration feature. *Safepoints* are proof script commands which store the entire current proof state. This stored proof state can be re-loaded later, e.g., to dispose explorative steps performed by the user. We consider safepoints to be used to mark the start of a proof exploration phase using the script and for mass-reverting of proof commands, following the safepoint command.

Further support for proof exploration already available in KeY are special macro steps, e.g., to close a proof branch or revert all steps back to the proof state where it was applied. Also in the scripting language different language constructs to perform proof exploration exist, e.g., the keyword `try` in a `cases`-statement to apply proof commands and if not successful to roll back the proof (see [BGU17] or Chapter 7 for more details). One additional feature that was identified during the evaluation to support users in proof exploration is a further proof script command: the `rewrite` command. This command allows to replace a user-provided term by another term.[2] In our concept, the execution of the `rewrite`-command tries to find a calculus rule that rewrites the user-provided term by the other term and if not possible result in

[2]The syntax of the rewrite command is: `rewrite find='f(x)' replace='g(x)';`.



Current Goals

[imp(p,q)]==>[imp(not(q),not(p))]

Sequent

p → q ==> ¬q → ¬p

Matching Pattern

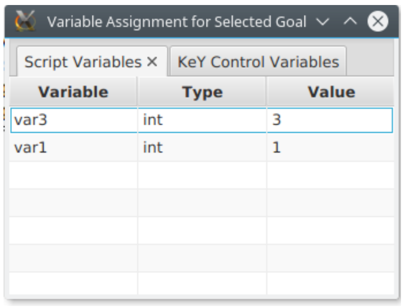
?X -> ?Y

Matches

Match 1: {?X=p, ?Y=q}

Match 2: {?X=not(q), ?Y=not(p)}

(a)



Variable	Type	Value
var3	int	3
var1	int	1

(b)

Figure 8.5.: Support for proof construction. The screenshot in Fig. (a) shows the sequent matcher. Users can enter a matching expression, select a proof goal and evaluate the matching expression for the selected proof goal. The result is a highlighting of the terms and formulas that are assigned to the schema variables during matching. If more than one possible matching result is present, users can inspect all results with a corresponding highlighting in the sequent view. In Fig. (b) the variables and their values for a selected proof goal are shown to the user.

the application of a `cut` command. The proof obligation to show that these terms are indeed equivalent and thus can be replaced by each other, can then be handled by either lightweight tools, such as SMT solvers, or the prover's strategies. This allows for a sound replacement of terms. A similar concept can be found in KeYmaeraX, by using the edit-button for formulas on the sequent.

8.3. First Experiments Using the Proof Script Debugger and KPS

The proof script language KPS, by using a first version of the PSDBG, was evaluated in first experiments in [Luo18; GLW18]. In the following we will briefly summarize the experiments, the results and the room for improvement identified in the experiments based on [Luo18; GLW18].

For the experiments, existing proofs that have been performed using the direct manipulation interface of the KeY system were transformed into proofs performed using KPS. To check whether the script lead to the same proofs, PSDBG was used to interpret the scripts and visualize the proof state.

8.3.1. Objectives of the Experiments

The objective of the experiments was to determine the effectiveness of KPS by assessing three aspects of KPS, referred to in the following as (a) (relative) feasibility of deductive proofs, (b) stability of the script against changes in the program or specification, and (c) the conciseness.

The aspect *(relative) feasibility* refers to the possibility to formulate all proofs originally performed in KeY by using the proof commands offered by KPS. The aspect "*stability* indicates the degree of effects caused by minor changes to the program or its specification" and the aspect *conciseness* refers to the possibility to shorten KPS scripts in contrast to performed user interactions in KeY.

The three aspects together are indicators for the effectiveness of KPS. The *relative feasibility* indicates whether the expressiveness of the textual interaction and the direct manipulation interaction for proof construction is the same, i.e., that KPS covers all possibilities to perform proofs relative to the direct manipulation interaction in KeY.

The *stability* measures, whether KPS scripts are resilient to changes in the proof input artifacts. The changes considered in the experiments were changes that are assumed to be performed during the development of the proof input artifacts. Reloading of proofs after a change in the proof input artifacts is assumed to be an issue for the user. Users often have to perform all interactions again to reach a similar proof state as before the changes.

With the aspect *conciseness*, it should be measured whether using KPS is suitable to reduce the amount of interactions, in the form of proof commands, by using the different structures of KPS. This aspect is measured by comparing scripts that directly encode the user interactions performed in the KeY system using direct manipulation for proof construction with the compressed scripts using the multi-matching feature of the scripting language.

8.3.2. Performing the Experiments

To evaluate the *(relative) feasibility* of deductive proofs in the first experiments, existing proofs of the Java standard libraries have been transformed into proofs using KPS. The specific proofs have been chosen by their complexity and amount of user interaction during proof construction, to have a chance of using the different structures of KPS. The underlying assumption is that trivial proofs would often only contain the proof commands `symbex`, followed by a `foreach` statement containing `auto`, or just the proof command `auto`. For the feasibility the following proofs have been replayed:

- the `compareMagnitude()` method of `BigInteger` class [Pfe17]
- the `split()` method of the `DualPivotQuicksort` implementation [Bec+17]

The aspect *stability* was determined for the following changes in the proof input artifacts:

- renaming variables
- simple, semantic-preserving rephrasings of loop conditions
- repositioning of commutative terms.

All changes have been performed for the existing proof of KeY's standard example `SumAndMax.java` and the renaming of variables has been also evaluated for the proof of the `split()` method of the `DualPivotQuicksort` implementation [Bec+17]. During the experiment, first the existing proof was transformed into a proof using KPS. As a second step each of the changes was performed and after each change the script was reloaded, as well as the existing proof in the KeY system. It was analyzed, which changes lead to an incomplete loading of the proof in KeY respectively an incomplete loading of a KPS proof.

For the aspect *conciseness*, the transformed scripts of the *feasibility* experiment were analyzed in detail, and if possible proof commands were summarized using KPS structures. The results were compared. The underlying assumption is that each proof command in the proof scripts corresponds to an interaction in the KeY system.

8.3.3. Analysis of the Results and Room for Improvement

The experiments for the aspect *feasibility* of KPS revealed that not all proof commands as defined by the KeY system have been implemented into the prototype yet. Furthermore, due to the named parameters and the aim for unambiguous proof commands the script commands in KPS were lengthy.

In detail, it was possible to fully transform the proof of the `split()` method of the `DualPivotQuicksort` implementation. The incomplete realization of KeY's proof commands was then revealed in the the proof of the `compareMagnitude()` method of the `BigInteger` class. Some of the *built-in* rules used in the original proof have not yet been realized in the early prototype for the user study.

For the *conciseness* it was possible to compress those scripts that contained repetitive interactions using the multi-matching facility of the `cases` statement (Table 8.2 shows the detailed results).

Name of the proof	Number of Interactions/Proof commands	
	KeY	KPS
Quicksort (sort)	6	6
Quicksort (split)	8	4
BigInteger (compareMagnitude)	587	503
DualPivotQuicksort (split)	127	127

Table 8.2.: Comparison of interactions respectively proof commands for the conciseness experiment [Luo18]

In these experiments, the size of the terms in the scripts that are passed as parameters to the proof commands were lengthy. Not only the term itself but also its context had to be passed to the commands, i.e., the formula on the sequent it appears in, needed to be given, to avoid ambiguous term references. Due to these lengthy expressions, the proof scripts were hardly readable.

The experiments that should measure the *stability* revealed that the scripts were stable, i.e., fully reloading the scripts was possible after the changes, in all tested cases, except the renaming of variables. Here, the script was not reloadable without issues, until a simple find-and-replace feature of a text editor was used to rename the variables appearing in the script. This find-and-replace, however, did not work for the reloading of proofs in the KeY system, because the names of the program variables directly influence the names of logical variables which may be stored in the proof file.

8.3.4. Experiences in Using PSDBG for the Experiments and Improvements

A report about the experiences in using PSDBG for the experiments and suggestions for improving the user interaction and the user interface is also included in [Luo18; GLW18].

In the evaluation, PSDBG was not only used for evaluating aspects of KPS but also to perform a proof not yet performed in the KeY system: a proof for the sortedness of a variation of the BubbleSort algorithm was started in the evaluation [Luo18; GLW18]. The focus was on the usage of PSDBG to construct proofs. It was reported by the experimenter that for the inspection of the proof state, when a proof attempt was unfinished, breakpoints, the tree visualization and stepping functions have been used and deemed themselves supportive.

The report concludes that the scripting language concept is sufficient to reconstruct typical program verification proofs using KeY. However, in the prototypical implementation technical support was missing for “efficient and sufficient interactions” and proof exploration in the evaluated version was challenging, e.g., as reloading a proof script was time-consuming.

The evaluation also resulted in lessons learned either for the concept or for the technical realization. Based on [GLW18], we will describe the lessons derived from the evaluation results in the following. Some of the ideas for improving the proof script debugger have been included into the concept description.

8.3.5. Lessons Learned from the Evaluation

The evaluation of KPS, while using PSDBG, resulted in lessons we have learned for the concept and the prototype. Some of the lessons drawn from the evaluation have already been included in the concept presented in Sect. 8.2. In the following, we will present the lessons in more detail.

Breakpoints as Support for Orientation As already considered in the interaction concept, the script representation together with the facility to add breakpoints to script statements deemed themselves helpful for finding the orientation in the proof after changes to the program and its specification. This was not possible in KeY without the script component.

Proof Script as Central Proof Representation Our idea was that the script together with the list of open goals and the currently selected open goal contains all necessary information for proof construction. Our assumption was that the stepping functionalities would be sufficient if more details are needed. The experiment showed that the proof tree of the KeY system is still vital for proof orientation and construction and should be available if the user needs the tree representation. KeY's proof tree allows distinguishing different proof branches when locally focussing on one branch. The branching labels in the proof tree contain information about the program state at that point or case distinctions in the proof. One downside of KeY's proof tree is that temporal sequences of proof commands are difficult to inspect, due to its size alone. A more condensed view of the proof tree with a clear correspondence to the script is necessary to gain a better overview. One challenge for such a view is a clutter-free presentation of proof branches. Based on this result, we have included a script tree view into our concept and prototype (see Fig. 8.2).

Representation of Terms and Formulas Terms and formulas in program verification proofs can be large and the user needs to reference them when using proof commands. Therefore, a suitable representation has to be found that is comprehensible for the user, requires little effort to reference and is unambiguous, such that the proof system is able to determine which exact formula or term the user referenced. Our solution uses schematic terms with place holders and schema variables as abstraction from the large concrete terms similar to the idea presented in [Völ03]. In the evaluated version, we included this representation only in matching expressions and for assigning matched terms to script variables. For proof commands, users had to reference the terms by their syntactical representation and their surrounding top-level formula. We already expected that this representation may lead to ambiguous matches, e.g., when a formula contained a subterm more than once. However, the experiment showed that this ambiguity appears disruptively often in practice. Using schematic expressions as parameters for proof commands, as considered in our concept, may attenuate this issue and a preprocessing step is needed before handing the expressions to the underlying proof system.

The challenge to handle references of large terms or formulas concisely in the proof script remains for future work. One option would be to hide the representation in the script using code-folding, reducing the size of the displayed script considerably. The solution would be as unambiguous as when referencing the whole term in a formula. Another possibility is to use term indices for representing terms. However, such references are unstable when reloading the script after a change, due to changes in positions of terms in a sequent. Also during the exeuction of a `foreach` statement their positions may differ. Another disadvantage would be that the user needs to inspect the sequent to know to which formula or term a command will be applied to.

A third option is to abbreviate terms using variables in the script and to reuse this abbreviation in the sequent view. This option has a drawback, as the stored term, when reused at a later point in the proof script may not be present on the sequent anymore. Either the abbreviated term must evolve with evolving proof, which may cause confusion for the user, or such a variable may only be used once in the script.

In addition to supporting users in viewing respectively displaying terms in the script, also support for inspecting terms or adding them to the proof script is needed. One option is that on term selection using direct manipulation (in the sequent as well as in the script) a context-menu with different possibilities could be offered to the user. Functionalities may include creating a matching expression for the selected term or decomposition of the term into its subterms together with a suggestion for a matching expression for subterm selection.

Selection using Match Expressions To select proof goals we allowed using match expressions over sequents and over the branching labels of the proof. Our hypothesis was that distinguishing between goals is mainly done via information found in the sequent. Experience showed however that matching of labels was used more often than matching terms, although this representation may be less resilient to changes (e.g., as Skolem variables are also part of branching labels, which may be subject of renaming when reapplying a proof script). One explanation is the effort for users to find the matching expression that only matches the intended goal together with the size of the terms. This is a consequence of the principle of *least effort.* Branching labels are easier to use, so the user gains *immediate benefit* when matching against labels, while the costs of finding more resilient matching expressions are *deferred* until the script has to be reloaded after performing changes in the proof input artifacts. Our idea for future work is to include a suggestion mechanism in the sequent matcher window computing a general matching expression for a selected term and also support for adding terms to the script by point-and-click on the sequent. This suggestion mechanism would also be beneficial for the latter issue of representing terms in scripts, where the suggested term may then be used as the term reference in the script.

Script-Construction Support In the KeY system users do not have to remember the exact names of all proof commands, as they are suggested by the system upon user request. This suggestion mechanism has to be provided in the script (see also [Völ03; Eas98]), e.g., by implementing it as context-sensitive auto-completion. Otherwise, the user has to remember all proof commands, of which more than 1500 are available in KeY. Auto-completion support is part of future work as the evaluated version of PSDBG did not contain this feature.

This support may come in two different variations. In the first version, the user types parts of a proof command and the system suggests applicable proof commands syntactically matching the parts the user has entered already. In case the user chose an applicable proof command from the suggestions of the system, the user gets visual highlights of terms where the command is applicable to and is now able to select a suitable term by point-and-click interaction, or by using the keyboard for selection. In the second version the user selects a term using direct manipulation to retrieve a list of applicable commands to choose from, similar to KeY's current support.

In the case that there is more than one proof goal present in the proof state, there also has to be a support to construct the appropriate textual selector for the chosen goal to which a proof command should be applied to. Additionally, to get context-sensitive support, the system needs to know the proof state in which the rules should be applied. For this, the proof would have to be constructed in the background up to the state that corresponds to the proof script location the user is currently working on. This is a huge difference to the pure direct manipulation interaction in KeY where the user always operates on the current proof state and applying rules always result in a new state.

Support for Proof Exploration Proof construction always contains proof exploration actions. Our experiment showed that different scripts or proof commands may be tried out to determine the next successful proof step in a proof state. For proof exploration the evaluated implementation contained the exploration facilities of KPS, as well as the possibility to use the direct manipulation interaction style to compose a proof that could be disposed if not needed. However, the version only allowed undoing single interactively applied proof commands, and disposing whole interactive scripts was added in the course of the evaluation to the prototype.

Efficient Script Reloading In the first implementation, script reloading was implemented such that the script was always fully executed after reloading it. This resulted in large loading times, which made proof construction more time consuming. It remains for future work to implement an incremental proof reloading functionality that makes use of differences between the proof objects and the proof scripts and only reloads those parts of the proof that were affected by a script change. The introduced *safepoints* may be used for incremental script reloading.

Additionally, we supported the switch of interaction in the first implementation only after executing a proof script. The seamless switch of interactions, from script-mode to interactive mode and vice versa, is left for future work. This functionality will benefit from using differences in the proof objects for incremental reloading of scripts.

8.4. Conclusion

In this chapter, we reported on the combination of the direct manipulation interaction style of the program verification system KeY with a text-based interaction style to construct program verification proofs. Our approach is implemented in the research prototype PSDBG, using the proof script language KPS. We also used a subset of KPS in the prototype of our seamless interaction concept presented in Chapter 10.

First experiments to assess the language KPS and the interaction concept realized in PSDBG have been carried out by transforming non-trivial proofs saved in KeY's proof format to KPS proof scripts to evaluate our proof scripting language. Furthermore, PSDBG was used to construct a proof to test the practical usage of our approach. We have reached the stage where it is possible to construct proofs using both interaction paradigms, however, the experiments demonstrated that the proof process needs further support.

Ongoing and future work includes a better support for the incremental execution of proof scripts, the handling of term references in the script and the support of all KeY proof commands. Functionalities adapted from software debugging such as using breakpoints and step-wise retracing proof construction seem helpful for keeping the orientation in the proof. Ongoing work is also to extend the proof exploration support, both on the language level and using direct manipulation. One possibility it to integrate the concept presented in Chapter 9 into the proof script debugger.

The combined interaction concept offers the user a larger range of interaction styles, and thus allows using different styles in different proof situations: While direct manipulation interactions are time efficient, making them more suitable for applying single rules, the text-based interaction style enables the user to use script commands and control flow constructs to handle repetitive user interactions.

It remains for future work to perform more elaborate case studies using PSDBG as well as to design and conduct a user study that explores whether adjustments have to be made to allow for an efficient and effective user interaction in PSDBG.

8.5. Related Work

The need to analyze failed proof attempts in interactive theorem proving has lead to different mechanisms for gaining insight into proof construction. The interactive theorem provers Isabelle [NPW02] and Coq [BC04], both, provide text-based interaction, and the way in which proofs are constructed allows to step over tactics, to revert a tactic application, and to add tactic invocations iteratively. The user can inspect the proof states between tactic applications. To get a deeper insight into tactics, both tools allow for the use of debuggers for the language in which the tactics and the tools are implemented (Standard ML respectively OCaml). While tactics implement generic proof strategies independent from the concrete proof problem, proof scripts are usually tailored to the current verification task and to the program to be verified. This difference manifests itself when debugging proof scripts in contrast to debugging tactics: when debugging a proof script, the user only has to consider the current proof task at hand; when debugging a tactic one has to keep in mind the different possible applications of this tactic in other proof situations.

Additionally, Hupel proposes an interactive tracing of Isabelle's simplification tactic [Hup14]. LEAN's metaframework [Ebn+17] – an API to the theorem prover LEAN – provides support for classical program debuggers to step through the execution of the declarative language of Lean.

Similar to the approach of designing KeYmaeraX [MP16], our approach of combining text-based with direct manipulation interaction for deductive program verification is based on existing work on general usability guidelines, HCI principles and requirements for user interfaces of theorem provers. These findings and principles are extended or specialized for the combination of direct manipulation with text-based user interaction in the domain of program verification.

Usability guidelines for tactic-based interactive theorem provers include providing different complementary views of the proof construction with the possibility to choose among them, offering meaningful operations on each view, the flexibility of proof commands and the focus on relevant information for the user [Eas98]. We have instantiated the multiple views for our use case by adapting the view concept from software-debugging to the domain of program verification. The principle of focusing on relevant information is realized in our case by providing different projections of the proof state together with a drill-down focus.

Additional requirements for usable theorem prover user interfaces contributing to the principle of *least effort* have already been identified [Völ03]. Examples include auto-completion for proof commands or filtering information to ease the identification of relevant information for making decisions in the proof process.

The stability issues for subterm selection and a suggestion that a pattern language for subterm selection is needed for more robust proofs was identified as requirement for theorem prover user interfaces. Our approach considers to use matching expressions, similar to the pattern idea of [Völ03]. We added different ideas and a discussion for handling the reference to large formulas in program verification proofs.

The authors of the guidelines [Völ03; Eas98] contrarily discussed the benefit of having a visual proof tree. While it can serve as an aid for orientation and abstraction [Eas98], also evidence for being not helpful was assembled in [Völ03]. Furthermore, it was suggested to offer text-based as well as graphical interaction in cases there is no clear advantage for either of both interaction styles. Already our first experiment showed that for large program verification proofs this visual aid can be helpful.

There are three categories of existing attempts to combine both interaction styles for theorem provers: The first includes systems that natively offer text-based user interaction for proof construction and are extended by elements that allow direct manipulation. These type of systems usually have their strengths in supporting the well-established script-based interaction style by offering helpful features also present in general purpose IDEs (e.g., auto-completion, syntax highlighting or indication of syntax errors). Here the scripts serve as focus for proof construction and feedback about the state is given by presenting goal states, icons in the editor's gutters or color coding. Proof navigation can be performed by clicking on the proof script statement. Two examples for such systems are the general-purpose interactive theorem provers Coq [BC04] and Isabelle/HOL [NPW02]. The approach of Coq additionally allows the user to apply tactics using point-and-click similar to KeY. However, visual aids such as the proof tree are not present, or are being subsequently implemented.[3] The debugging metaphor which enables us to use breakpoints for a better orientation in the iterative proof process and the stepping-into possibility down to the level of single calculus rules is not part of theses approaches.

The second combination are systems that originally offered direct manipulation interaction and were either redesigned to combine different proof styles for proof construction (such as KeYmaeraX [Pla18; MP17] as the successor of KeYmaera [Pla10], which only offered direct manipulation) or systems that originally only offered direct manipulation and added text-based support afterwards, such as our contribution. In both cases the designers benefit from the experiences with a system offering direct ma-

[3]The addition for Coq is available at `https://askra.de/software/prooftree/`.

nipulation. The approach of KeYmaeraX solved the issue of referencing subterms by using indices that are added to each formula and visualized for the user in the respective proof state. The proof script serves as a textual representation of the proof tree sacrificing a visual representation of the whole proof tree. The proof tree is rendered as collapsed deduction paths from the root to the open goal. The deduction paths are represented as tabs in a tabbed view, and users can unfold the top-most element, which is a sequent, in a tab to navigate further to the root node. This unfolding allows users to navigate the proof and to step-wise add more information about the deduction path to retrace the proof performed so far. Switching goals can be performed by switching tabs. The proof script in KeYmaeraX can be extended by the direct manipulation interactions performed on the open goals and by textually adding proof commands. To ease interaction, control-flow structures for exhaustively applying rules are present in the script language. For each newly created goal a placeholder is added in the proof script. Proof goal selection is then performed either by direct manipulation interaction in the goals tab or by adding proof commands at the respective position in the proof script. The user interaction and user interface of KeYmaeraX follows the principles *familiarity, traceability, tutoring, flexibility* and *experimentation* [MP17].

The third approach for a combination of both styles is combining two existing interfaces of a general purpose theorem prover, such as the approach of PGWin for the theorem prover Isabelle [AL04]. Here the direct manipulation interface IsaWin that follows a notepad metaphor and iconifies domain elements was combined with the proof management interface ProofGeneral. PGWin as the combination contains both representations – the proof text and the iconified representation. The user can choose which representation to use to construct proofs. Here the direct manipulation aspects of PGWin differ already in the representation of domain objects from our approach.

9. Proof Exploration in Interactive Program Verification Systems

Contents

9.1. Introduction

The process of proving programs correct is characterized by iterations of unfinished proof attempts. When encountering an unfinished proof attempt, users of interactive program verification systems need to find the cause for the open proof. This can be a mismatch between the program and its specification or missing proof guidance in the proof process. The user now needs to determine which of the cases applies in the situation. If the cause is found, the user needs to interact with the system to proceed in the proof process. This can be a change of the specification or of the program, or an interactive rule or proof search strategy application.

Handling unfinished proof attempts can, for example, also be identified in the process of performing mathematical proofs with pen and paper. One strategy for mathematicians when handling open proofs is to use *proof exploration actions* to determine promising proof steps. Such actions can be side-calculations with concrete values or adding assumptions and proceeding with the proof process in order to determine whether these changes would result in a complete proof. Evidence for this *trial-and-error* approach can also be found in excerpts of an interview with Gert Faltings presented in [Mel94], where it was stated that in some cases proofs also contain trial-and-error phases, if the mathematician does not know directly how to proceed.

Exploration actions are often performed in a situation where the user has already engaged in the proof situation – these actions are performed in the context of the current partial proof (e.g., trying out different alternatives or instantiating concrete values for variables).

We want to adapt this idea to the situation of program verification to support the user in understanding the cause for a failed proof attempt without leaving the current proof situation. Leaving a current proof situation to change the proof input artifacts, i.e., the program and its annotations, according to a hypothesis a user has formed and to test this hypothesis leads to another iteration in the proof process. Starting a new iteration has the negative consequence that the user needs to gain orientation in the proof again and has to find back to the same or a similar situation he has seen before the change.

With our concept presented in the following we want to minimize the number of situations where users need to start a new iteration in the proof process. We want to achieve this goal by allowing the user to be able to test a formed hypothesis in the current situation and only adjust the proof input artifacts if the hypothesis seems promising.

9.2. Our Concept for Proof Exploration

In the following for our concept, the starting point in a proof process is an unfinished proof attempt in a program verification system with a sequent calculus. For the demonstration of our concept, we will again use the KeY system, because proof construction in KeY is performed using a sequent calculus. Although we will rely on peculiarities of the KeY system to some extent, the concept is applicable to other interactive proof systems implementing a sequent calculus for proof construction.

As initial situation we consider a proof formed by automatic strategies, possibly interwoven with interactive proof steps, that contains several open goals. Regarding the intent of the user, we assume for this situation that he starts to inspect the sequents of the open goals for details in order to comprehend and assess the current proof state. During this inspection, the user may discover formulas on a sequent that may not coincide with his mental model about their origins or truth values in the current proof state. To be able to test whether the user's expectations would lead to a closed proof, the user needs to change the proof state according to his expectations and proceed further in the proof to inspect the resulting states.

The possibilities to change the state are modifying the underlying proof problem (i.e., changing the source code or the specification), applying calculus rules interactively or introducing formulas into the proof, e.g., using the cut rule in KeY to both –use and prove a user-specified lemma. In other systems users may use lemmas for this purpose. If the user chooses to modify the proof input artifacts (e.g., by editing parts of the specification) he has to abandon the current proof and start the proof process over again to test whether these changes lead to progress in the proof search. With our concept of proof exploration we want to support users in introducing hypotheses about the proof problem into the current proof state by starting an *exploration mode* from the current proof state and test the introduced hypotheses without fully abandoning the proof task and starting over again. To achieve this we allow the user to make changes to the current proof obligation using exploration actions in a special exploration mode and progress in the proof search in the usual way. If the changed proof state leads to a (partially) closed proof, the user gained information about what to change in the actual proof problem to have a proof. A central part of our concept is to inform the user about the origin of formulas and terms he has changed using exploration actions. This information is important for the user to determine which parts of the proof artifacts need to be changed for a successful proof.

9.2.1. Reasons and Corrective Actions for Unfinished Proof Attempts

To be able to construct appropriate exploration actions, we first need to reconsider the reasons for unfinished proof attempts and the actions users currently perform to find and remedy the cause for a failed proof attempt.

Recall from Chapter 5 that the reasons for a program verification system to fail in the automatic proof search are either a mismatch between the program and its specification, consequently the proof problem is not provable, or the proof system needs user input in the form of quantifier instantiations, lemmas or other rule applications.

In case the program does not fulfill its specification, the user has to distinguish whether the specification is correct but the program contains a defect, or the program is implemented correctly and the cause for an unfinished proof attempts is the requirement specification. The specification may contain a precondition that is too weak or a postcondition that is too strong for the program to satisfy. Additionally, the auxiliary specification may need adjustments, i.e, the loop invariant may be too weak or too strong for the proof situation. As our use case, we will focus on the verification of a single method against its contract, without considering any further class invariants. The concepts presented are, however, not restricted to this use case.

To modify the specification, users of the KeY system edit the source file which contains the source code and the corresponding specification – possible modification actions for these proof input artifacts according to our use case are to *add, delete* or *modify* formulas in the pre- and postcondition as well as in the loop invariant.

When a formula is added to the precondition, the user adds information about the pre-state to the proof obligation and therefore restricts the program states for which the proof obligation must hold. Vice-versa, when the user removes a formula in the precondition more states are covered in the proof and the proof may become more difficult.

In contrast to editing the precondition, adding formulas to the postcondition may result in a proof with more cases to prove. Removing formulas in the postcondition may result in less cases to be handled in the proof.

If the user suspects the cause for an unfinished proof attempt in a defect of the program, the user changes the statements in the program according to the hypothesis about the defect.

Modifying the specification or the program requires the user to restart the proof process in order to prove that the possibly modified program fulfills the possibly modified specification.

9.2.2. Exploration Actions

Our goal for the exploration concept is to provide the user with actions on the sequent level that capture the intent of the previously mentioned corrective actions without the need to fall back on modifications of the proof input artifacts.

One of the challenges for the user when modifying the proof state on sequent level (e.g. by applying calculus rules or exploration actions) is to relate the formulas and terms on the sequent to their origins even after the application of sequent calculus rules has deconstructed or modified the original proof obligation. Each of the resulting formulas on a sequent originates either from the proof input artifacts, from implicit assumptions (e.g., about data structures in the program), from user input when applying rules interactively (especially the cut rule and instantiations of quantifiers) or from instantiations or skolemizations automatically generated by the proof system.

The aforementioned actions on the program can be converted to changes on the sequent. We will group the possible changes into the following three categories:

(A) adding formulas to the antecedent or succedent,

(B) removing formulas from the antecedent or succedent, and

(C) editing formulas or terms in the antecedent or succedent.

In the following we will describe all three categories in more detail.

(A) Adding Formulas In the following, let Γ, Δ be sets of JavaDL formulas and f, g formulas in Java DL.

In general, adding arbitrary formulas to the antecedent is unsound. It may be then possible to derive propositions that can not result from the original proof obligation.

However, when looking at the *cut*-rule in the sequent calculus, we can use the following similarity: One of the resulting branches of the cut-rule corresponds to adding the cut-formula to the antecedent (depicted by the rule 9.2), the other resulting branch corresponds to adding the cut-formula to the succedent (depicted by the rule 9.1).

$$\mathit{AddToSuccedent}\ \frac{\Gamma \Rightarrow f, \Delta \qquad (\Gamma, f \Rightarrow \Delta)}{\Gamma \Rightarrow \Delta} \tag{9.1}$$

$$\mathit{AddToAntecedent}\ \frac{(\Gamma \Rightarrow f, \Delta) \qquad \Gamma, f \Rightarrow \Delta}{\Gamma \Rightarrow \Delta} \tag{9.2}$$

Thus, it would be possible to implement the exploration actions in a sound way by using the existing cut-rule and hiding the second branch, i.e., the branch that does not correspond to the desired action, from the user in the exploration mode. We denote this hiding by enclosing the branch to hide in parenthesis in our rules (e.g., in the rule 9.1, the right branch is hidden).

(B) Deleting Formulas Deleting arbitrary formulas from the antecedent concerns the completeness of the calculus.

The sequent calculus for JavaDL, as it is implemented in the KeY system, offers two rules that can be used for the case of *deleting* formulas: the hiding rules or also called weakening rules. The *weakening-right* or *HideRight* rule hides a formula in the sequent's succedent (see the rule 9.4), the *weakening-left* or *HideLft* rule, hides a formula in the seuqent's antecedent (see the rule 9.3). The weakening rules visually remove selected formulas from the sequent and also hide them from the automatic strategies, but they do not delete the formulas. Users are always able to introduce them back on to the sequent.

$$\mathit{HideLeft}\ \frac{\Gamma \Rightarrow \Delta}{\Gamma, f \Rightarrow \Delta} \tag{9.3}$$

$$\mathit{HideRight}\ \frac{\Gamma \Rightarrow \Delta}{\Gamma \Rightarrow f, \Delta} \tag{9.4}$$

(C) Editing Formulas or Terms We do not only want to allow users to delete or add formulas to the seuqent but also edit formulas or terms. As with adding arbitrary formulas to the sequent also editing formulas on the sequent is in general unsound. However, the effect can be captured using the cut-rule and a weakening rule (see rule 9.5). Consider changing a formula f to a formula g in the antecedent. We can use the cut rule to introduce g onto the sequent and use the weakening-left rule to hide the formula f. Similar to the addition of formulas, the second branch can be hidden from the user in the exploration mode to not clutter the screen.

$$\cfrac{\cfrac{(\Gamma, f \Rightarrow g, \Delta) \qquad \Gamma, g \Rightarrow \Delta}{\Gamma, f \Rightarrow g, \Delta \qquad \Gamma, f, g \Rightarrow \Delta}}{\Gamma, f \Rightarrow \Delta} \tag{9.5}$$

The same argumentation can be used for the succedent by using the weakening-right rule instead of weakening-left.

If a user changes a subterm in a formula there is a second possibility to model this behavior, which we have depicted in Eq. 9.6. Let $p(t_1, \ldots, t_n)$ be an arbitrary formula with subterms t_i. Using the *cut*-rule, the equality of t_1 and t_j can be introduced to the sequent and, using the rules to substitute subterms in a formula according to an equality (depicted in Eq. 9.6 with the step *applyEq*) before applying the *weakening* rule (depicted in Eq. 9.6 with the step *weak*), we have modeled the same behavior as when using the cut-rule with the changed formula. The only difference is that it remains for the user to show the equality of the subterms to be sound and not the equality of the whole formula, which may be in some cases easier to show.

$$\cfrac{\cfrac{\cfrac{(\Gamma, p(t_1, \ldots, t_n) \Rightarrow t_1 = t_j, \Delta) \qquad \Gamma, p(t_j, \ldots, t_n), t_1 = t_j \Rightarrow \Delta}{(\Gamma, p(t_1, \ldots, t_n) \Rightarrow t_1 = t_j, \Delta) \quad \Gamma, p(t_1, \ldots, t_n), p(t_j, \ldots, t_n), t_1 = t_j \Rightarrow \Delta}\ (\mathit{weak})}{(\Gamma, p(t_1, \ldots, t_n) \Rightarrow t_1 = t_j, \Delta) \qquad \Gamma, p(t_1, \ldots, t_n), t_1 = t_j \Rightarrow \Delta}\ (\mathit{applyEq})}{\Gamma, p(t_1, \ldots, t_n) \Rightarrow \Delta}\ (\mathit{cut}) \tag{9.6}$$

9.3. The Exploration Mode

One major design decision for the interaction concept is to encapsulate proof exploration actions in an *exploration mode* that must be invoked explicitly by the user. Introducing different modes into a system is a source for mode confusion and it is generally better to avoid explicit modes if possible.

However, exploring the proof state is an activity that we believe is conceptually different from proving and needs special support, but uses the same or similar actions. Furthermore, we want the users to be able to use the system as far as possible in their usual way. As a consequence, we have to especially focus on presenting the system status to the user in an appropriate way. We assume that if the user explicitly invokes the exploration mode, he should be aware of the situation he is changing to.

The alternative would be to let the user perform the proof exploration in an external window, in order to emphasize the mode change. This however, means that the integration into the existing user interface and workflow would be disturbed.

For reaching this fundamental decision, the idea of proof exploration was presented to a few KeY users in an informal meeting and the functionality was discussed in detail. The following two significant opinions in the meeting influenced our decisions: firstly, the users wanted to be prevented from accidentally accessing the exploration actions in the regular proof as there was the fear that the proof may be compromised. Secondly, the usual way of using the system should not be disturbed. We believe that a user study is necessary to explore whether the decision for an integrated mode was the better choice.

To not disturb the usual workflow for KeY users, we allow the exploration actions to be accessed in the exploration mode in the same way calculus rules are accessed – using the usual direct manipulation actions on the terms and formulas on the sequent.

Although the exploration rules can be realized using existing calculus rules, we chose new names for the rules that match their purpose in the exploration mode, i.e., instead of displaying the cut-rule as such to the user, we display an action named *Add formula to the antecedent*, which technically can be realized using the *cut*-rule.

Furthermore, in our concept applying the rules in the exploration mode has two significant differences to applying the same underlying rules outside the exploration mode: Firstly, if a second branch for justification is created, this branch is hidden in the proof tree as long as the user does not choose to show the second branch. Secondly, the resulting branch is marked such that the automatic proof search strategies do not include the branch without an explicit user-invocation.

The rules performing exploration actions with side-effects for the view are deactivated outside the exploration mode in order to prevent the user from applying rules that hide details in the proof tree and therefore to prevent mode confusion.

Additionally, in the exploration mode a variety of supporting features for the user are essential: (1) retrieving information about the origin of formulas or terms that have been changed, (2) allowing to revert explorative actions, (3) allowing to add the hidden branches to the current view, and (4) enable the automatic proof strategies to apply rules to these branches and to get a visualization where, in the proof tree, exploration actions have been applied.

Furthermore, the user should be able to get an overview over the exploration actions he has applied, e.g., in an additional window or tab.

If the user manages to close all non-hidden branches, the proof itself is not closed, as the hidden branches remain to be proven. To avoid mode confusion, the user needs to be informed in this case about the open hidden branches. Furthermore, the system should support the user in this situation by an easy access to the hidden branches and potentially also a strategy that tries to prove the hidden branches automatically, such as the macro `Close Provable Goals Below` in KeY.

In our concept for the devised proof exploration it is essential to provide information about the origin of formulas and terms to the user. This information should contain the origin information relative to the proof input artifacts, for example, either by giving a textual description such as "the formula `x>=0` results from the precondition of the method `split()`" or by highlighting the corresponding information in KeY's source code view.

The origin information is necessary at the end of a proof exploration phase where the user has collected enough information about the current proof to know which next step in the proof process is promising. One outcome of the exploration phase may be that a formula needs to be added, deleted or modified. In the case of an edit or deletion operation the user is interested in the origin of this formula in the input artifacts to know where to perform a change. In many cases this information can be computed by tracing the direct parent relation in the proof.

In case the user has added a formula F to an arbitrary proof state, depending on F, there may not be one single origin for F but rather many different origins of the terms F is composed of. Giving origin information for F to the user in this case can (in general) only be an approximation. This issue is exacerbated by the fact that terms used in F might not be present in the original proof obligation but were produced as part of the symbolic execution (e.g., for the program `x = x + y + 1`, symbolic execution in KeY introduces a fresh variable `x_0` to store the intermediate result of the addition).

9.4. Interaction in the Exploration Mode

In the following we will present the different concrete activities we consider in our concept and their realization in a prototype for the KeY system. For the activities we distinguish between the *regular proof mode* and the *exploration proof mode*.

9.4.1. Interplay: Exploration Mode and Regular Proof Mode

We consider *regular proof search*, *proof exploration* and *resolve exploration* as the main activities which users perform in the context of proof exploration. In Fig. 9.1 we have depicted the intents and actions that are used to switch from one activity to another. We will first describe the activities and will proceed with the actions that can be used to change between the activities.

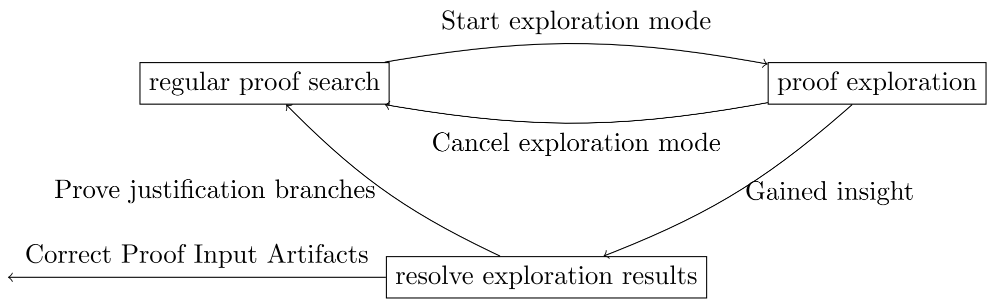


Figure 9.1.: Activities for the exploration of the proof state

Regular proof search. The proof process starts with regular proof search performed by the user under the assumption that the program meets its specification (and that the latter is provided in a form suitable for the automatic proof search strategies) with the single intent of completing the proof. This process includes inspection and changes of the proof input artifacts, actions in the verification system such as manual rule applications or invoking automatic strategies, as well as proof comprehension activities.

There are different situations that may lead the user to stop the regular proof search activity: The user may discover a proof state with open goals which the proof system is not able to close automatically within the resource bounds specified by the user. Another reason to end regular proof search may be that the user discovers a part of the proof obligation that does not coincide with his mental model about the proof state. Common to both situations is that, at the end of the regular proof search, the user needs to comprehend the proof state and decide how to proceed.

Proof exploration. Compared to regular proof search, the proof exploration activity has the main goal to provide further information to the user in order to complete his mental model, e.g., by confirming or falsifying assumptions made by the user. In the best case, after proof exploration, the user is (a) convinced that the program meets its specification (possibly after correcting the proof input artifacts), and (b) his mental model matches the information provided in the current proof situation.

One of the situations where proof exploration might be helpful is thus when the user suspects that an open goal is not provable, because the propositions in the goal do not coincide with the user's conception of the proof state and the proof problem. Actions we consider in this activity are modifying the sequent by adding, deleting and editing formulas or terms. After modification of the proof state, we assume the user to proceed in the proof process with similar actions as in the regular proof search, possibly alternating with modification actions, until the user has gained sufficient insight about the situation. We call this insight *exploration result* in the following. The result may be that the user has closed the exploration branches or that the exploration branches stay open and the user gained knowledge about the proof problem. For our concept, we consider that the user does not change the proof input artifacts in the proof exploration activity.

Resolve exploration results. If the user has gained insight into the proof problem using proof exploration activities, he can now exploit this insight. For this, the user must have access to all performed exploration steps.

Depending on the outcome of the exploration activities, there are different possible next steps: if an exploration action provided no further insight or produced a proof state that is more difficult to prove than the original proof obligation, the user can *revert* the exploration step. Another possibility is to further inspect and retrace a promising exploration step, e.g., to determine the origin of a formula that was modified during the proof exploration to be able to make the necessary correcting changes in the proof input artifact – or to inspect the justification proof steps and branches for a certain property to be able to use the information in the remaining proof process.

Switching activities. For our interaction concept, we consider the user to first perform actions of the regular proof search activity and switch activities by using a *Start exploration mode* action. Such a start action can, for example, be implemented using a context-menu entry or a toggle button to switch modes. If the user is performing exploration actions, the user may leave the exploration mode explicitly by using a special *cancel exploration mode* action. In this case, there needs to be support to completely revert the actions of the proof exploration activity. At the same time, the user needs be informed about reverting the exploration actions, such that the user is still able to cancel this operation. If the user leaves the proof exploration mode after gaining insight into the proof problem with an exploration result, the user may perform activities to resolve the exploration actions. If the user performs actions to prove the justification branches after the exploration branches have been closed, the user can switch back to the regular proof mode. If the user abandons the proof and corrects the proof input artifacts according to the gained insight, the user can restart the proof process.

9.4.2. Proof Exploration in Action

In the following, we describe an exemplary workflow for our proof exploration concept supported by screenshots of a prototypical implementation.

In general, the proof exploration rules may be applied in any proof state that contains open goals. Here, we describe the interaction in the exploration mode using a simple example. We chose this example, as the sequents of the open goals are simple enough to easily demonstrate the possible interactions. Usually, the sequents of a regular proof problem are more complex and contain more formulas.

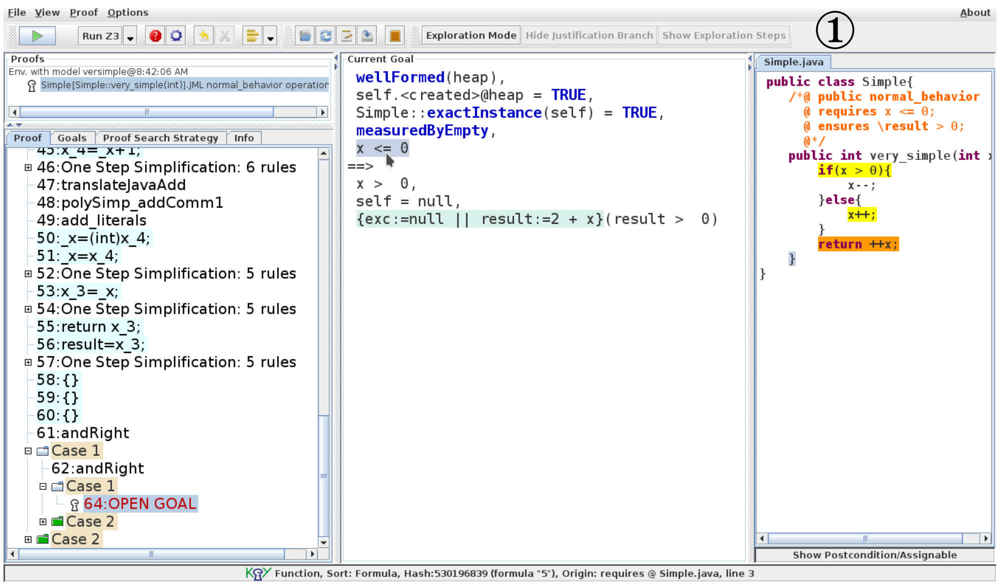
Let us consider a user wants to prove that the program in listing 9.1 adheres to the simple specification stating that if the program is started in a state in which the variable `x` is greater or equal to 0, the result of the method is greater than 0 in the post state.

The user loads this program and specification into the KeY system and tries to prove it correct in the *regular proof mode*. Using the macro `Autopilot` the user retrieves a partial proof with one open goal, which is depicted in Fig. 9.2.

Listing 9.1: Simple Java program with specification in JML.

```
/*@ public normal_behavior
  @ requires x <= 0;
  @ ensures \result > 0;
  @*/
public int very_simple(int x){
    if(x > 0){
        x--;
    }else{
        x++;
    }
    return ++x;
}
```

The path to the open goal in the proof tree reveals that the proof state corresponds to the path through the program where the `else` branch is taken. In this simple example, the user may already suspect that the formula from the precondition ($x <= 0$) is wrong and change it to ($x >= 0$).



Env. with model versimple@8:42:06 AM
Simple[Simple::very_simple(int)].JML normal_behavior operatior

```
45:x_4=_x+1;
46:One Step Simplification: 6 rules
47:translateJavaAdd
48:polySimp_addComm1
49:add_literals
50:_x=(int)x_4;
51:_x=x_4;
52:One Step Simplification: 5 rules
53:x_3=_x;
54:One Step Simplification: 5 rules
55:return x_3;
56:result=x_3;
57:One Step Simplification: 5 rules
58:{}
59:{}
60:{}
61:andRight
Case 1
  62:andRight
  Case 1
    64:OPEN GOAL
  Case 2
Case 2
```

```
 wellFormed(heap),
 self.<created>@heap = TRUE,
 Simple::exactInstance(self) = TRUE,
 measuredByEmpty,
 x <= 0
==>
 x >  0,
 self = null,
 {exc:=null || result:=2 + x}(result >  0)
```

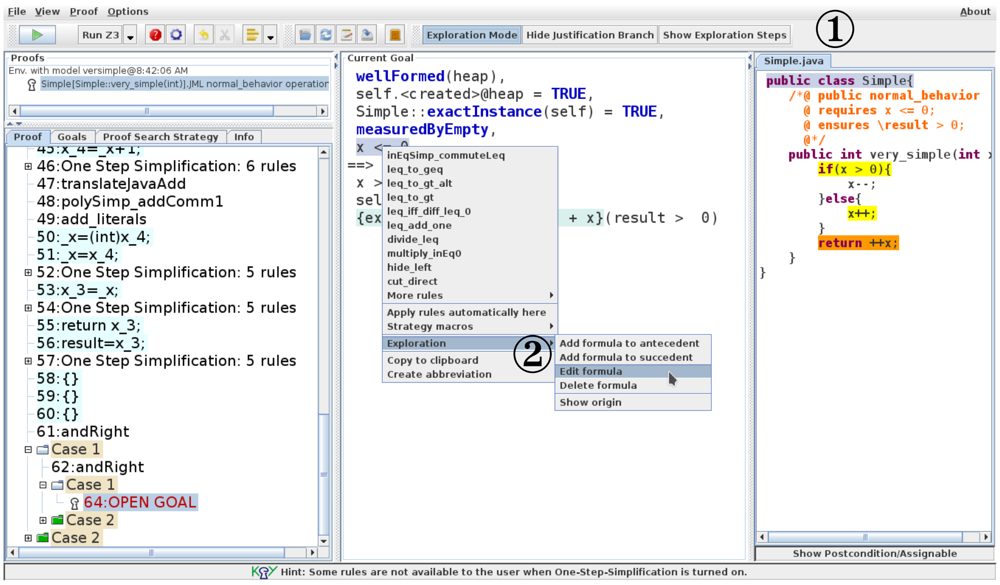
```
public class Simple{
   /*@ public normal_behavior
     @ requires x <= 0;
     @ ensures \result > 0;
     @*/
   public int very_simple(int x
       if(x > 0){
           x--;
       }else{
           x++;
       }
       return ++x;
   }
}
```

Function, Sort: Formula, Hash:530196839 (formula "5"), Origin: requires @ Simple.java, line 3

Figure 9.2.: Sequent after using the macro `Autopilot`. In the toolbar ① the exploration mode can be invoked.

The user now has two possibilities: Firstly, find the corresponding formula in the proof input artifacts, change it accordingly, reload the problem and redo all interactive steps.

The other possibility is to switch to the *exploration mode* (in the toolbar in Fig. 9.2 ①) and use the offered action *Edit formula*, which can be accessed via the context menu when selecting a term or formula, similar to rule applications, in case the exploration mode is switched on (see Fig. 9.3).



Env. with model versimple@8:42:06 AM
Simple[Simple::very_simple(int)].JML normal_behavior operation

```
45:x_4=_x+1;
46:One Step Simplification: 6 rules
47:translateJavaAdd
48:polySimp_addComm1
49:add_literals
50:_x=(int)x_4;
51:_x=x_4;
52:One Step Simplification: 5 rules
53:x_3=_x;
54:One Step Simplification: 5 rules
55:return x_3;
56:result=x_3;
57:One Step Simplification: 5 rules
58:{}
59:{}
60:{}
61:andRight
Case 1
  62:andRight
  Case 1
    64:OPEN GOAL
  Case 2
Case 2
```

```
wellFormed(heap),
self.<created>@heap = TRUE,
Simple::exactInstance(self) = TRUE,
measuredByEmpty,
x <= 0
==>
x >
sel
{ex                  + x}(result >  0)
```

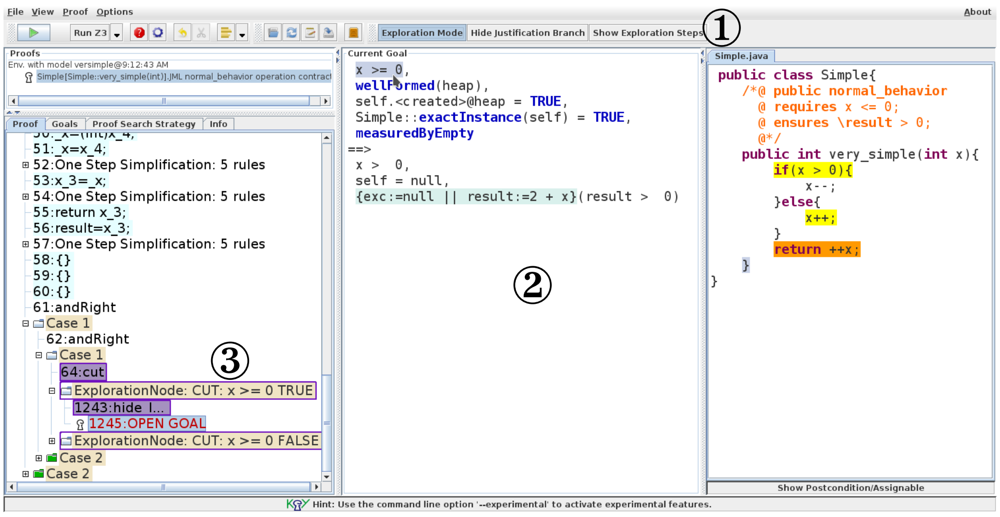
```
public class Simple{
  /*@ public normal_behavior
    @ requires x <= 0;
    @ ensures \result > 0;
    @*/
  public int very_simple(int x
      if(x > 0){
          x--;
      }else{
          x++;
      }
      return ++x;
  }
}
```

Hint: Some rules are not available to the user when One-Step-Simplification is turned on.

Figure 9.3.: Invocation of proof exploration actions via the context menu ②. This action can only be accessed if the exploration mode is switched on in the toolbar ①.

Using the action *Edit Formula* results in a change of the formula on the sequent (see Fig. 9.4 ②) and a second branch with the proof obligation to justify the change (see Fig. 9.4 ③). This second branch is hidden from the user, if the corresponding option has been chosen. Hiding the justification branch is only a view option, and in the underlying proof, the branch is still available. Hence, the proof can only be fully closed if the justification branch can be closed as well. Therefore, as it is only a view option, the user can always toggle the option and see all hidden branches.

After performing exploration actions, the user can proceed in the verification process using his preferred way of proof construction. In the example, the proof of the changed goal leads to a closed goal (see Fig.9.5, the goal with the number 1258). The justification branch is still open, as shown in the screenshot (Fig.9.5, the goal with the number 1244). The user now gained the information that changing the formula ($x \leq 0$) at the point where it is introduced would close the proof. To find this location the user may select the goal before the exploration action, for example, by using the possibility to view all exploration steps and directly jump to the changed node (Fig.9.6 (a)), and click on the changed formula. He retrieves a context menu with the option *Show Origin*. Selecting this functionality presents the user a new window con-



```
Env. with model versimple@9:12:43 AM
  Simple[Simple::very_simple(int)].JML normal_behavior operation contract
```

```
50:_x=(int)x_4;
51:_x=x_4;
52:One Step Simplification: 5 rules
53:x_3=_x;
54:One Step Simplification: 5 rules
55:return x_3;
56:result=x_3;
57:One Step Simplification: 5 rules
58:{}
59:{}
60:{}
61:andRight
Case 1
  62:andRight
  Case 1
    64:cut
    ExplorationNode: CUT: x >= 0 TRUE
      1243:hide l...
      1245:OPEN GOAL
    ExplorationNode: CUT: x >= 0 FALSE
  Case 2
Case 2
```

```
 x >= 0,
 wellFormed(heap),
 self.<created>@heap = TRUE,
 Simple::exactInstance(self) = TRUE,
 measuredByEmpty
==>
 x >  0,
 self = null,
 {exc:=null || result:=2 + x}(result >  0)
```

```
public class Simple{
    /*@ public normal_behavior
      @ requires x <= 0;
      @ ensures \result > 0;
      @*/
    public int very_simple(int x){
        if(x > 0){
            x--;
        }else{
            x++;
        }
        return ++x;
    }
}
```

Hint: Use the command line option '--experimental' to activate experimental features.

Figure 9.4.: Proof state and sequent ① after editing the formula using an exploration action. Invoking the exploration action results in two new proof branches ③, one that contains the changed sequent and a justification branch.

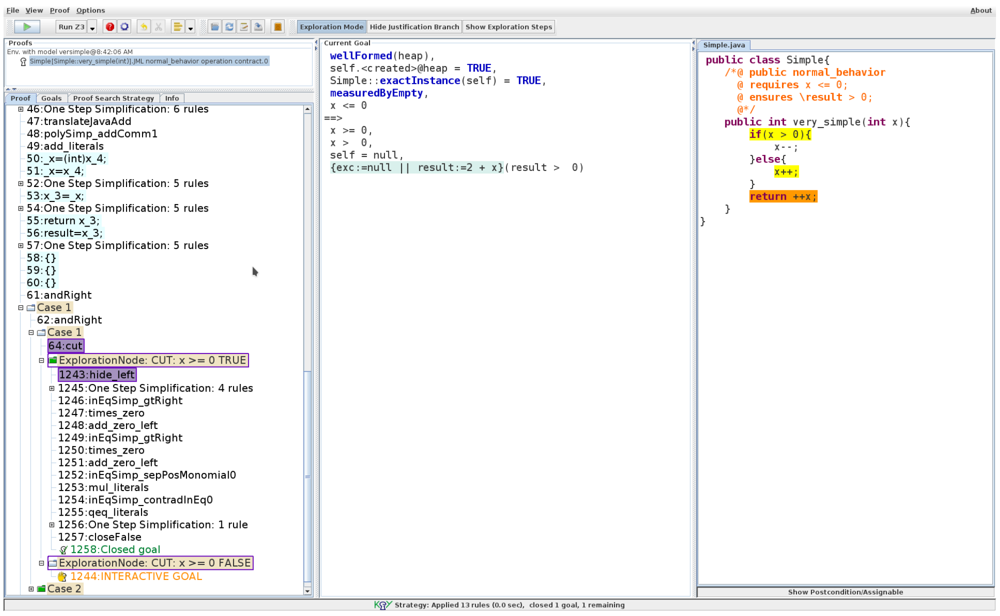
taining information about the selected formula (see Fig. 9.6 (b)). The user can now see that the formula results from the precondition (in line 3 of the Java file `Simple.java`, as shown in Fig. 9.6 (b) ②. He can now change the precondition in the source file, load the problem again and try out the automatic proof strategy. In our example, this change leads to a closed proof.

The user also has the possibility to try to prove the justification branch, which is possible in some, but not all cases. In our example, this is not possible, as the justification branch still contains the wrong assumption about the range of x and we cannot conclude $(x >= 0)$ from the formulas in the antecedent. Further techniques such as the generation of counterexamples can be used in such a case to try to reveal such a contradiction.

9.4.3. presentation of Additional Information in the Exploration Mode

As shown in the example workflow in Sect. 9.4.2, we have presented two windows which the user can access in the exploration mode: the window containing all performed exploration steps and the window containing the origin information about selected terms and formulas. In the following, we present these two windows in more detail.

Exploration Steps View. To allow the user to gain an overview over all performed exploration actions and to easily navigate to the nodes on which exploration actions have been applied to, we devise a view containing all exploration actions of a current



```
Env. with model versimple@8:42:06 AM
  Simple[Simple::very_simple(int)].JML normal_behavior operation contract.0
```

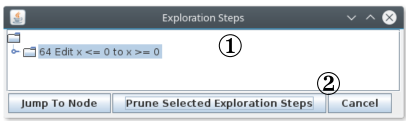
```
⊞ 46:One Step Simplification: 6 rules
  47:translateJavaAdd
  48:polySimp_addComm1
  49:add_literals
  50:_x=(int)x_4;
  51:_x=x_4;
⊞ 52:One Step Simplification: 5 rules
  53:x_3=_x;
⊞ 54:One Step Simplification: 5 rules
  55:return x_3;
  56:result=x_3;
⊞ 57:One Step Simplification: 5 rules
  58:{}
  59:{}
  60:{}
  61:andRight
⊟ Case 1
    62:andRight
  ⊟ Case 1
      64:cut
    ⊟ ExplorationNode: CUT: x >= 0 TRUE
        1243:hide_left
      ⊞ 1245:One Step Simplification: 4 rules
        1246:inEqSimp_gtRight
        1247:times_zero
        1248:add_zero_left
        1249:inEqSimp_gtRight
        1250:times_zero
        1251:add_zero_left
        1252:inEqSimp_sepPosMonomial0
        1253:mul_literals
        1254:inEqSimp_contradInEq0
        1255:qeq_literals
      ⊞ 1256:One Step Simplification: 1 rule
        1257:closeFalse
        1258:Closed goal
    ⊟ ExplorationNode: CUT: x >= 0 FALSE
        1244:INTERACTIVE GOAL
  ⊞ Case 2
```

```
 wellFormed(heap),
 self.<created>@heap = TRUE,
 Simple::exactInstance(self) = TRUE,
 measuredByEmpty,
 x <= 0
==>
 x >= 0,
 x >  0,
 self = null,
 {exc:=null || result:=2 + x}(result >  0)
```

```java
public class Simple{
    /*@ public normal_behavior
      @ requires x <= 0;
      @ ensures \result > 0;
      @*/
    public int very_simple(int x){
        if(x > 0){
            x--;
        }else{
            x++;
        }
        return ++x;
    }
}
```

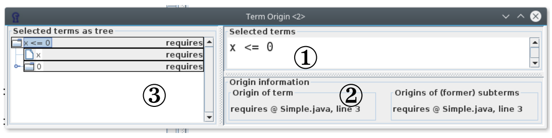
```
Strategy: Applied 13 rules (0.0 sec),  closed 1 goal, 1 remaining
```

Figure 9.5.: State after closing the exploration branch using the automatic strategies of KeY. The justification branch is still open, as the automatic is prevented from touching this branch by default.



```
64 Edit x <= 0 to x >= 0
```

(a)



```
x <= 0    requires
  x       requires
  0       requires
```

```
x  <= 0
```

```
requires @ Simple.java, line 3
requires @ Simple.java, line 3
```

(b)

Figure 9.6.: Fig. (a) shows the information window about the exploration steps. If more than one exploration step has been performed the tree view ① shows which exploration steps would be affected if the user chooses to prune ②, i.e., revert exploration steps. The user can also jump to the proof node on which the exploration step has been performed ② or discard the window ③. Fig. (b) shows the origin window, where the user can retrieve information about the origin of formulas and terms. The window contains the selected formula ①, the origin information for the selected part of the formula ② and the term structure of the selected formula with corresponding origin information ③.

proof. We devise to present the exploration actions in a tree structure. For a proof node n on which a proof exploration action has been applied to, a node e is added to the exploration tree. For each child node n_i of n to which an exploration action has been applied to, a child node e_i is added to the exploration tree. Reversing exploration actions can be performed in a sound way by pruning the proof at the node to which the exploration action is applied to. Pruning a proof tree at a node n means to the whole subtree below the node n. Pruning however, also affects the child nodes to which exploration actions have been applied. They are reverted as well. We therefore devise to present exploration actions in a tree structure such that users are aware that reverting an exploration action for a node also reverts other exploration actions that have been applied to child nodes.

Origin Information View. To enable users to inspect the origin of terms and formulas after exploration actions, we devise a view in our concept that contains origin information. In this view we consider the user to be able to view the term respectively formula structure together with the semantic origin of each subterm respectively subformula. As semantic origin information we devise to allow the user to trace back symbols to the proof input artifacts. More specifically this is the line where the symbol originates from together with a qualitative information from which JML-clause the symbol originates from, e.g., if a formula is part of the precondition of a method, the origin is the *ensures* clause. This information is not always possible: there are cases where formulas and terms are added during the transformation steps between the proof input artifacts and the proof obligation. We denote the origin of these elements by being *implicit conditions*. For implicit conditions no source code location can be computed. Therefore, the user can only get the information that the condition is added during the transformation step. An example for an implicit condition is that certain objects are created or that certain data structures are not null.

9.5. Related Work

When considering mathematical education, Schoenfeld [Sch92] identified proof exploration as one of six activities within the problem-solving process during proof construction. Furthermore, the interview which is reported on in [Mel94], the mathematician Gert Faltings mentioned that if he does not know how to proceed with the proof, it is a try-and-error approach, which shows that there are situations where some kind of exploration is performed during proof construction. One example for an exploration action in mathematical proofs in our view is to try out initial values for an induction, to see whether already the base case contradicts the hypothesis.

Means to explore proofs in verification systems are for example to explore which cases the proof system is able to prove automatically, i.e., possibilities to perform automatic proof search, and if the search does not lead to a closed proof, revert the steps performed by the system automatically. Such proof search strategies are for example already implemented in the KeY system as the macro `Close Provable Goals Below`. Other possibilities, for example present in different verification systems (e.g., in Dafny, KeY or Isabelle/HOL) is the possibility to generate counterexamples. A counterexample for a proof state or proof verification condition allows users to explore the proof state more target-oriented, often with concrete values for variables. Some systems also give feedback to the user in terms of the input artifacts (e.g., the annotation or the source code) which path or which condition the system was not able to show. These feedback mechanisms allow users to get hints where to look for errors.

Another way of proof exploration present in many verification systems is the possibility to *try out* different proof search strategies, and being able to revert them either manually, or by the system. This kind of exploration gives insight into which parts of a proof needs attention and which parts can be proven automatically. In Isabelle/HOL, for example it is common to use the tool sledgehammer, which tries out different lightweight proof strategies, on the original proof obligation. This reduces the proving effort for simpler proof obligations for the user.

Users can also explore proofs and proof states in verification systems, such as in KeY or KeYmaeraX by selecting formulas and terms and view applicable rules in a proof state. Furthermore, these systems provide a preview for rule applications. This allows users to explore which rules may seem promising to apply in a proof state to proceed with the proof.

The semantic origin of formulas for a sequent calculus proof, similar to our approach, can also be accessed in the Symbolic Execution Debugger (SED) [HBH18; Hen16] for KeY. The purpose is to provide information which parts of the method's contract could not be proven, which should also give users hints which parts of the proof input artifacts may need attention.

9.6. Conclusion and Future Work

We have presented a concept for proof exploration in a verification system with a sequent calculus that allows users to explore the proof state by modifying parts of the sequent without having to leave the current proof situation. After an informal presentation of the concept to KeY users, the exploration mode was prototypically realized in the KeY system as integration into the usual workflow. The prototype described in this chapter is influenced by the feedback given by the users during this informal presentation. It remains for future work to fully realize the concept. Especially navigation to the exploration nodes and pruning exploration actions needs to be implemented. The origin view developed in the scope of the proof exploration is currently realized as new view for the KeY system. It remains for future work to integrate this view into the view containing the annotated source code.

In this chapter, we contributed with an example workflow for a proof using proof exploration actions as considered in our concept. As a subsequent step, the realization has to be presented to KeY users for further feedback and for the use in case studies.

Besides performing case studies where proof exploration can be used, a larger user study should be conducted to explore whether the interaction concept covers the user's needs as well as mental models. This also includes whether the exploration mode as considered in our concept leads to mode confusion. It needs to be determined, whether users would use proof exploration and in which situations. Furthermore, also the presentation of the origin of selected formulas as we have presented it in this chapter needs to be evaluated. It needs to be investigated whether the presentation is comprehensible for users.

Moreover, in our first prototype we have chosen to keep the origin information for each subterm and add further origin information in the course of proof construction. We need to evaluate whether the additional information may confuse users if deep in the proof process. For a final version of the presentation of the origin of formulas, the information also has to be presented in the view containing the annotated program, such that users see the formulas in their original context. In our concept presented in Chapter 10, we consider presenting the origin information in the original context.

10. A Seamless Interaction Concept For Interactive Program Verification

Contents

One goal of this thesis and the topic of this chapter is to develop an interaction concept for interactive program verification that supports users in keeping the gap between the mental model of the proof and the actual tool-supported proof small.

One could argue that choosing a representation such that there is no gap would be the perfect solution. The user just provides the requirement specification and the proof system would find the proof automatically. For trivial properties and trivial programs, this is possible. With increasing complexity of the program and the properties the proof cannot be found automatically anymore and user intervention is required.

Auto-active verification systems follow that argumentation and allow users to only interact in terms of the program by providing the requirement specification and additional auxiliary annotations, such as assertions to guide the proof search. This approach is already suitable to reduce the aforementioned gap by hiding proof details from the user and allowing interaction only on the level of the artifacts that serve as input for the proof process. Many verification problems can be handled auto-actively. Other tools, such as the KeY system, also incorporate means to allow for auto-active proof guidance, e.g., using explicit assertions in the source code or block-contracts. For some non-trivial programs and properties, as well as when annotations are insufficient for the proof system to find a proof, users need to gain more insight into the proof process. One example is the addition of the *SMT-Inspector* to the VCC tool after experiencing that users need to see the SMT-encoding in some cases [Bor14]. As a result, the auto-active approach suffers from a so called *leaky abstraction*, where internals of the system that should be hidden from the user have to be revealed.

Increased size and complexity of the verification target lead to increased complexity of the proof problem. One reason for this is that for a proof, the dependencies between different software modules have to be taken into account (e.g., call hierarchies between different methods). Specifications have to be chosen in a way that they establish the specification for a specific target (e.g., the method contract for a method), but also support the successful proof of other depending specifications in the system (e.g., the method contract for a calling method). Users performing program verification tasks need to keep track of these dependencies for a successful proof task completion as the dependencies may become relevant during proof construction.

Often several iterations are necessary to write the specification: Users may make errors or miss properties in the first attempts of writing the specification. To correct the specification users need feedback about the cause of an unfinished proof attempt. In some cases this can be given on the level of the input artifacts, however there exist cases where users need to gain insight into the internal proof process to understand the reason for the unfinished proof attempt.

Our hypothesis is that one major bottleneck in proof construction in state-of-the-art systems is the proof comprehension and exploration part. Without providing methods to support these tasks advancing proofs gets extremely challenging for users.

In this chapter we present a concept for seamless program verification based on our observations in the user studies and on the usability principles described or referenced in Chapter 2.3.2. For the concept we consider the two user-tasks – proof exploration and proof comprehension – from the beginning on as main focus and as an essential part for proof construction. We integrate aspects found in related systems which aim to aid the user in these tasks.

Our concept follows the assumption that the user of the proof system has to be aware of proof internals, like in the interactive approach, but the interaction is also supported on the language of the input problem, like in the auto-active paradigm. We therefore propose that both interaction styles have to be available to the user. Unlike in each of the two approaches (auto-active and interactive) the user is able to interact on different proof artifacts and on different representations of the same problem and to *seamlessly* switch between the representations, with only little effort in gaining orientation.

During the proof, our solution allows choosing which interaction style to follow for each part of the proof: the user may use the auto-active approach (as a more black-box style), and if necessary he may switch to the proof representation of the problem in order to continue with proof construction on the more internal level (i.e., in a white-box style). To enable a seamless change in the representations, the dependencies between different parts of proof artifacts are made explicit in the system and visible to the user.

In addition, we support the exploration phase by including an exploration mechanism such as the one presented in Chapter 9 that allows the user to inspect different parts of the proof artifacts and apply analysis methods to them.

In summary, we present an interaction concept that allows users:

- to choose the representation of the proof state (e.g., a logical representation or in terms of the annotated program) and the kind of interaction style for proof construction according to the user's preferences and the proof situation at hand,
- to seamlessly switch the proof state representation when another one seems more informative,
- to easily recognize relations between information artifacts displayed to the user among the different representations, and
- to focus on complex proof subtasks, leaving trivial subtasks to the proof system.

10.1. The Structure of Verification Tasks

The correctness of a software system can depend on different *aspects* of the system (e.g., termination, functional behavior or information-flow). Although each of these aspects can be part of the same specification, it may be advisable to individually verify the parts of the specification in order to keep the verification task and proof size small. We assume that while performing the verification task for one aspect, the focus of the user will be on that part of the specification that contains statements about the specific concern (as introduced in Sect. 3.5). In the concept presented in the following, we will concentrate on the aspect *functional behavior* of a software system S, where Req is the requirement specification formalizing the functional behavior of S.

The activities a user performs in the process to verify the concern $\langle \mathrm{Req}, S\rangle$, can be categorized in activities that either advance proofs (e.g., writing a specification or applying calculus rules) or that help the user in comprehending the current proof situation (also called proof state) in order to decide the next proof step or proof advancing activity.

Orthogonal to this classification, this work is further based on the hypothesis that activities in the verification process can be grouped into two classes: activities on the *local level* and activities on the *global level* – each class requires the user to consider specific concerns and parts of the system. What belongs to the global and the local level of the verification process is not only user-dependent, but also depends on the structure of the verification concern the user is currently working on.

We will call the part of the system/(sub)system and the specification the user in interested in at a given time the *user focus*. Examples for activities on the global level are concerned with the requirement specification, e.g. formalizing or changing the requirements. Activities on the local level are concerned with the auxiliary specifications and advancing the proof, e.g., providing or changing the loop invariant, and proving that the loop invariant implies the post condition. To prove that an implementation conforms to its requirement specification with a particular proof system, auxiliary specifications may be required to give the used proof system necessary information for the proof search. Our hypothesis is that the user performs activities on both levels and switches between both levels during the course of the verification process.

10.2. Description of Our Concept

Our concept aims to support the user in the proof process, particularly in the proof construction and proof comprehension, by providing different *projections* of the proof problem and proof state, and by allowing all three interaction paradigms common in interactive program verification systems (as described in 3.6). Our concept also supports the verification task by means for *abstraction* of as well as *focusing* on details of the proof problem and by breaking down the proof task into smaller *subtasks* and providing means to *select* these subtasks. Additionally, our concept conforms to some of the usability principles presented by [Eas98] for usable theorem provers and is further influenced by the user studies presented in part II of this thesis and by our experiences with verification systems, such as KeY.

Our goal is to design an interactive deductive program verification system that provides the right means of interaction in each proof situation. The complexity of the problem is reduced by concentrating on a subset of dependencies at a time and by providing the required information (resp. hiding unnecessary information) whenever possible during the verification process.

One essential building principle is that the user needs different context information both to comprehend the proof situation and to advance the proof – which context information is needed depends on what kind of concern is in user focus and also depends on the level on which the user is performing activities (i.e., on the global or local level).

One of the challenges of providing the user with the right amount of information are existing dependencies between (a) the different parts of the system and (b) between the requirement specification and auxiliary specification currently in user focus. The large amount of proof states, which each may become large in size as well, contribute to the aforementioned challenge.

From our observations in the user studies, we derived the hypothesis that users of interactive program verification systems need both an overview over the system and the bigger picture of the proof task and a way to focus on specific parts of the proof problem. One exemplary observation from the user study is that users of the KeY system searched for the relation between the program and the proof state currently presented by the system. Additionally, the huge proof states lead to situations where users missed important formulas on the sequent that gave hints about defects in the program code or insufficient specifications. The overview over the proof was often obtained by adjusting the view onto the proof tree in a way that only branching nodes were visible. Users then navigated to the open goals to see in which case the proof stays open and then focused on the sequents themselves. Users also often used the branching labels as guides and source for information.

One part of our concept is to support the user in switching between the different representations of proof artifacts by allowing to step-wise focus on details of the different artifacts. This step-wise focusing should help the user in keeping the overview and dependency information gained from one representation and transferring this knowledge to the alternative representation.

In contrast to other verification systems, such as KeY, the user interface in our concept covers the whole process of proving programs correct: Within the same verification tool, and thus user interface, the user provides the program code with specification given as annotations and also performs the program correctness proof.

As a starting point for the verification process, the user can generate the proof verification conditions from this input and let SMT-solvers try to prove these verification conditions automatically. This step already divides the proof problem into smaller chunks which we believe are better manageable by users. Additionally, using lightweight tools to separate trivial proof problems from complex problems that need additional user input helps the user to focus. When selecting a proof verification condition, the user can now seamlessly change the views from an overview over the whole proof problem to the logical encoding of the verification condition. We include two interaction styles on the logical encoding in our concept: a *direct manipulation* style, where users can select terms and formulas and retrieve a view with applicable rules and a *script-based* style, where users can textually encode the rule applications in a *proof script* for the proof verification condition.

The proof script is central on the logical level and serves different purposes: it serves as the representation of the proof structure and allows navigating through the proof by clicking onto the respective statement in the script. It furthermore persists the proof and logs the user interactions: Actions performed using the direct manipulation style are textually encoded and added to the proof script. The script also serves as a mean to advance the proof by allowing users to textually add proof commands to the script and executing it.

10.2.1. Projections: Multiple Views onto the Proof Problem

To prove the requirement specification Req of a software system S users switch between the global and the local level of the concern $\langle \mathrm{Req}, S \rangle$ iteratively. In addition, the user may change between the verification of different concerns if Req has to be adapted. The requirement specification may also serve as auxiliary specification in the verification of other concerns.

The goal of our concept is to allow a seamless change from the global to the local level which allows inspecting dependencies between the different components of the verification target and the relations between the levels. As a prerequisite, the dependencies between the concerns need to be made visible to the user. We assume that the user has a different *user focus* on the available components on the different levels therefore different context information can be necessary and should be shown to the user. This idea adheres to the usability principle of *anticipation*, which makes the claim that "all information and tools needed for each step in the process" [Tog14] should be provided to the user.

To achieve the seamless change between the artifacts our concept devises different projections on the proof problem to be presented to the user as different views. These views support users on the level they are currently working on as well as take into account that users shift their focus during the proof process.

Presenting multiple views of the proof state and allowing meaningful operations on those views are considered as two usability principles for theorem provers that support the user in the task of theorem proving. The different views should help the user to form different models of the proof task, to be able to choose the most appropriate one for deciding the next goal-directed action [Eas98].

We consider the following views as being essential for the task of proving a software system correct:

- a view showing the proof input artifacts, i.e, the source code and its annotations,
- a view showing the system and proof structure as well as the proof progress overview,
- a view that focuses on the logical representation of a single PVC with the possibility to construct a deductive proof for this PVC.

Source Code View

In our user study presented in Chapter 5, participants accessed the annotated source code in different phases of the process and with different intents and actions on these artifacts. The goal of the source code view is to allow the users to access these artifacts at any time in the proof process and to allow for the desired actions on these artifacts. At the beginning of the proof process, to prove the correctness of a software system, users may use the source code to formulate the requirement and possibly the auxiliary annotations. The annotated source code serves then as input for the verification system. During the proof process, users may inspect the program and the annotations and also modify both, if necessary.

There are different possibilities to display the program and its annotations to the user: both artifacts – program and annotations – can be shown in a combined view or shown separately, but also vary the amount of annotations that are shown can vary (e.g., showing all annotations as provided by the user, only certain parts or no annotations at all).

Showing both artifacts in a combined view can already be found in auto-active verification systems such as Dafny or VCC. The idea behind a combined view is that the annotations and the source code are closely related: firstly, the annotations state properties about the program and program locations and secondly, the position of the annotations, especially of assertions, in the program and the relative position among the other existing annotations matter for the generation of proof verification conditions. A further advantage of this choice is that the maintenance of annotations is simplified in case the user changes the source code, the annotations can be directly adjusted as well. The disadvantage of such a combined view is, however, that the source code itself may get cluttered by annotations if the complexity of the properties or the program increases. Projects in which large software systems have been verified, for example the Hyper-V project within VerisoftXT, report on the maximum ratio of one to three comparing lines of code with lines of annotation [Bau+12]. One way to address the issue of a source code cluttered with annotations is to separate the annotations and the source code in two different views. This separation would provide more *visual clarity*, as in each view only elements of one domain are represented. This may reduce the user's cognitive load. However, as the position of the annotations matters, additional references to the source code locations have to be added to the annotations. This choice would have the advantage that the two artifacts with different formalisms would be separated and inspected in isolation by the user. However, if the user needs the relation to the program location the user has to switch the view and navigate to the referenced location.

To avoid this context switch, a third possibility is to provide a view onto the requirement specification in a separate view and a view onto the source code combined with the location-dependent auxiliary annotations. The advantage of this view is that the location-dependent auxiliary annotations are shown at the point in the source code where they belong to, at the same time the requirement specification is separated such that visual clarity of the view is increased. This solution also does not guarantee a full clutter-free source code and parts of the auxiliary annotations are developed using information from the requirement specification. As a result users may still needs to switch contexts.

There is a fourth possibility, which is present in static analysis tools: only show the source code and no annotations, as they are implicitly given. The advantage is that the source code is not cluttered at all. The properties that are being checked in static analyzers are lightweight, compared to general properties in program verification tasks and do not require (auxiliary) annotations, as they result from the structures present in the programs. For proving user-defined, possibly complex properties however, this is not possible as the prover needs information in the form of annotations on what to prove.

During the proof process, the users have different intents when accessing the source code and the annotations as visible in the user study, which pose different requirements onto the possible actions on the view. The intents of the users can be divided into intents to *manipulate* the artifacts and to *inspect* the artifacts.

Intents to access the source code and the annotations include: the comprehension of the proof problem or the proof state, the search for mistakes in the proof input artifacts, and the construction of the annotations and the program. Adding auxiliary annotations for proof construction can also be an intent for proof construction.

As support for inspection, visual support such as syntax highlighting and line highlighting should be present in the view. More importantly, though is a possibility to inspect the relation between elements of the proof state and the proof input artifacts to support the comprehension tasks.

If the user has found an error in the proof input artifacts, wants to proceed in the proof process or extend the proof input artifacts, supporting features similar to those in text editors or IDEs are necessary. This support includes functionalities that allow the modification of the source code and the annotations (e.g., copy-paste or deletion features) and features for navigation through the source code. Furthermore, support for writing source code and annotations in the form of auto-completion and checking for syntax errors while writing should be provided to the user when interacting with this view to provide immediate feedback to the user and prevent errors on the syntactic level.

System and Proof Overview

As introduced in Chapter 3.5, the software systems that are proven correct can be units or modules with dependencies on each other resulting in dependencies between the specifications.

Not only the systems to be verified have an internal (hierarchical) structure but also the corresponding concerns are structured by being composed of several proof verification conditions. This structure helps to divide the large proof problem into smaller sub-problems, which may support users in dealing with the overall proof task. Due to the aforementioned dependencies and hierarchical relations, it is crucial for the user to gain an overview to build up a mental model about the problem to prove such that decisions about the next actions to take can be made. Furthermore, the information about dependencies is necessary to gain an overview about the parts that influence each other and need to be kept track of during the proof process. For a successful proof of a concern, all PVCs generated for this concern have to be proven, respectively the conjunction of the PVCs has to be valid. As the number of PVCs may become quite large, means to keep track of the progress of their proofs is part of the global overview.

We devise a view that supports the user in gaining a *global* overview over the concern and the proof task. This view should allow for activities to progress in the proof process on a more global level, as well as provide means to navigate from the global overview to the individual proofs for a concern, its PVCs and the proof obligation that logically encodes a single PVC. Mainly, hierarchical structures need to be presented in this view to the user; this presentation should allow for a step-by-step refinement from the abstract top-level entities down to the more detailed entities, similar to zooming into details of a picture.

Interactions that users should be able to perform in this view are browsing activities for the hierarchical structures and the dependencies, as well as to apply general proof search strategies to all or a part of the PVCs without the need to know the internal details of the proof. If the automatic is already able to prove the concern there is no need for the user to look into the proof if he is not interested in the details. This estimation allows the user to determine those PVCs that need further attention, either because interaction is needed or annotations need to be adjusted. We have observed in our user study that some participants used the automatic proof search strategies of KeY to determine which parts need a more detailed inspection or interaction and which parts can be handled by the automatic proof search.

The challenges within this view are twofold: firstly, presenting all information without overwhelming the user and, secondly, allowing for suitable and meaningful interactions.

In this view the hierarchy information of the proof artifacts has to be presented and the overview over the proof progress. One example of navigating hierarchical structures which may serve as a guide for navigating proof structures are file systems. Here, two solutions prevail that can be combined in two adjacent views: (1) show the file system structure as a tree structure to select the level in the tree which should be inspected further. Here, the path to the selection is a prominent information presented to the user. (2) The second solution is to let the user navigate through the file system by displaying one level in the tree in detail and navigation is performed by selecting an element of the level to *zoom* into the next level. Often, in file explorers both views are combined. If only solution (2) is visible, the user still gets the path information by a so-called *breadcrumb-bar*. If this breadcrumb-bar is visible, the tree structure provides redundant information to the user.

To not overwhelm users with too much detail, solution (1) allows for the possibility to fold or unfold the layers of the tree.

Both solutions solve the issue of displaying the information about hierarchical dependencies (e.g., the dependency between a PVC and its proof obligation), however, the information about dependencies that become apparent during proof construction (e.g., call dependencies and usage of different requirement specifications) and the overview over the proof progress still needs to be presented.

In the user study we were able to observe that the participants applied a general proof search strategy to the proof obligation in KeY at the beginning of the proof process with an intent to gain an overview of which sub-problems may need interaction and attention. This lead us to the conclusion to consider in our concept that the activity to apply a general proof search strategy to the concern should be possible in this view. A possibility to achieve this is by selecting the system S in the *system and proof overview* and retrieve a list of possible strategies to apply to it. The same should then also be applicable for the single PVCs in this view. The result of the verification attempt should then be presented in this view as well, such that users do not need to switch views.

To gain an overview over the proof progress, one of the simplest information that can be displayed is a number that indicates the number of PVCs already closed amongst all PVCs. This information, however, may be too coarse. Furthermore, as the proof problem is divided into smaller tasks, at least the information about the progress of the smaller tasks should be presented as well. In the KeY system, for example the number of open goals is shown in the status bar, after the application of the prover's strategies and users can access a view that lists all open goals.

Another possibility is to present a list of all verification (sub)tasks with an indication about their progress, this is for example present in verification systems such as Why3 [FP13]. The list view has the advantage that it is possible to present more details of the (sub)tasks than just a number indicating the progress. To give an overview over the nature of the proof of a PVC we also consider presenting the number of open branches of a proof for a PVC in this overview. This enables the user to gain an overview over the possible complexity of the single proof tasks.

Typical activities we consider in the system and proof overview are the *selection* of proof artifacts for detailed inspection, *selection* of proof tasks to work on and *browsing activities* to build a mental model about the problem and system structure.

10.2.2. Logical and Proof Construction View

In auto-active verification systems, users need to come up with auxiliary annotations for which often insight into the proof process and the logical encoding is needed [BBK11;

Bor14]. We therefore propose a view that enables users to gain insight into the logical encoding (PVC) and the deduction steps performed, if they are not able to solve the problems on the program level or need more details on the proof problem. Users should be able to focus on one verification task in isolation.

The notation of the logical encoding should be as close as possible to the input artifacts, e.g., names of identifiers such as fields in the program should not be renamed by deduction and transformation steps. This would adhere to the principle of *consistency*. In cases where renaming of identifiers cannot be avoided, a possibility to trace back to the original version of an entity should be provided. In the user study the renaming issue and therefore the retracing of the origin of symbols was addressed by participants.

As with the two other views also on this view proof construction should be possible. A natural choice is to allow for a direct manipulation interaction style, similar to the interaction in the KeY system, by pointing and clicking onto terms and formulas and retrieving possible rule applications for the selected position.

As users should be prevented from performing actions that may prove to be disadvantageous, i.e., where users have a wrong assumption about the effect of an action, it is essential that users should be able to observe the result of a rule application before actually applying it to advance the proof. This information should be given on different levels of granularity, both on a more abstract level, to give a rough estimate about the rule's effect (e.g., by showing how many branches result from the rule application) and on a more detailed level, to provide an insight into how the proof would evolve if the rule would be applied.

To support users in choosing the most promising rule among a list of applicable rules, the rules shown to the user should be sorted. One possibility would be to sort the rules according to the number of branches their application would produce or whether their application would close the proof branch. A proof with many branches is often conjoint with more work for the user. Another possibility would be to sort the rules according to complexity measures for the formulas in the proof state, i.e., if a rule simplifies a formula in terms of its length or number of symbols.

Additionally, to be able to persist the interaction we devise that also a sub view of the logical view should contain the possibility to perform the proof using textual interaction. We will call this sub view *script-view*.

Both interaction styles should be usable interchangeably and in alternation, to allow for different user preferences. Interactions performed using direct manipulation should automatically extend the proof script. This also allows that the actions performed using direct manipulation are reversible, as users just have to delete the corresponding statement in the script.

Besides the single PVCs, also the current overall proof state needs to be presented to the user, for example for inspection. Proofs may branch, e.g., because of rules that create case distinction, and therefore, means to depict a possibly branching proof has to be accessible in this view which should at least allow for navigation through the proof performed so far. We devise that proof exploration has to be supported in this view using the concept presented in Chapter 9, as support for detailed inspection and to decide whether the logical argument corresponds to the user's mental model of the current situation.

On the logical view, proof construction takes place on the most detailed level. Here, the user focuses on the individual propositions that are either assumed or need to be proven for the corresponding program state. We argue that formulas with similar origins should be displayed close to each other, e.g., by grouping formulas resulting form the precondition of a program together (similarly for the formulas that need to be proven).

Additionally, to keep the relation to the origin, formulas that contain program expressions should be depicted as long as possible in the way they have been introduced by the user. This may also be reflected by using the same fonts and names.

From experiences and observations in using the KeY system, the logical representation may become lengthy. For this we devise that the user should have possibilities to hide formulas, abbreviate them and to abstract from a concrete set of formulas, by e.g., giving it a name. Furthermore, grouping and sorting of formulas has to be supported, such that users are able to arrange the view in a way that fits their needs and expectations.

10.2.3. Relations between Proof Artifacts

Up to now, we have addressed the difficulties that arise with the system's and problem's structure and dependencies on each view individually. However, dependency relations also exist between the different proof artifacts across the different proposed views. These dependencies are often implicit and users need to keep them in mind while proving or invest resources to search for these dependencies. One essential building block of our concept is thus to make these *hidden dependencies*, as they are called by Blackwell and Green [BG07], visible to the user.

An example for a hidden dependency was observable in our user study (in Ch. 5) where the participants searched for relations between the different proof artifacts and also expressed that the support for finding these relations could be improved in the target of evaluation. Furthermore, participants asked for a highlight of conditions that could not be proven in the proof input artifacts, similar to the way auto-active systems support users in finding the reason for a failed proof attempt.

We will go into detail about the relations across the different aforementioned views in the following.

Concerning the verification target, each implementation in the *source code view* has a representation in the *system overview* that shows its location in the system hierarchy.

Having the information about call contexts of subsystems, the user may gain an overview about the state in which subsystems are called. This information is necessary for the requirement specifications of the subsystems.

For an overview over the proof task, not only the proof progress is important but also the information about dependencies between the proof obligations, the proof input artifacts and the PVCs. Furthermore, also the use of general lemmas for the proof of the PVCs is a necessary dependency information which should be accessible in the overview. The information about lemmas is also necessary to get an overview about what can be (re)used during the proofs.

The PVCs in the *system and proof overview* have a relation to parts of the properties of a subsystem, i.e., a PVC has a relation to a path through the program and the annotations that either hold on this path or should be proven for this path. A PVC also has a direct relation to the *logical view* where the logical representation is a formalization of the PVC. The two relations together induce the relation between the individual formulas in the logical representation and the annotated program. Formulas in the *logical view* can have their origin from statements in the program or the annotations. Here, the origin relation can be on different abstraction levels: the detailed information is the position of the property in the annotations or source code, a more abstract information would be the rough estimation in which part of the program or annotation the origin is from (e.g., from the precondition). Another origin of formulas may be from specific rule applications, such as the cut-rule. Here the rule and its position in the script should be presented as origin information.

If the *script view* and the view containing the logical representation are considered to be two different views, relations between both views exist as well. During the proof progress each statement in the script corresponds to a logical representation. The logical representation depicts the current proof state of a PVC. If a rule is applied to the current state, the logical representation changes. This process reveals another relation, a relation between a rule application, a statement in the proof script and the logical representation of the proof state, e.g., a node in a proof tree.

It is crucial for user support to make the relations visible. As the number of relations may become large, it is advisable to only display this information upon user request.

Furthermore, we devise to support “zooming-in” from the abstract overview to the detailed representation, which was also observable in the user study. This especially means to support the user in switching between the different views and keeping track of the dependencies that exist in one view but also the dependencies between the views. This can be supported by positioning views sequentially, i.e., starting with the most abstract view (proof and project overview) down to the detailed logical view, and thus restricting the work-flow to the “zooming-in” process. This restriction ensures that important relations are always visible and can be inspected. Further support could be to allow the user to trace the origin of element across the different views.

10.3. A Concretization of the Concept

In the following, we will present our choices for a concrete user interface that adheres to the concept presented in the previous sections. Accompanying our descriptions we will present screenshots of a prototypical implementation. Those parts not yet implemented are demonstrated via mock-ups.

A central design choice, as stated in Sec. 10.2.3, is the sequential layout of the views (as depicted in Fig. 10.1) to support the “zooming-in” activity. Only two adjacent views are shown next to each other at once, on the one hand for the user to be able to relate the contents of both views to each other, and on the other hand not to clutter the screen and overwhelm the user with the full information at once to reduce the cognitive load of users. Each view together with the relations to its neighboring view is described in detail in the following.

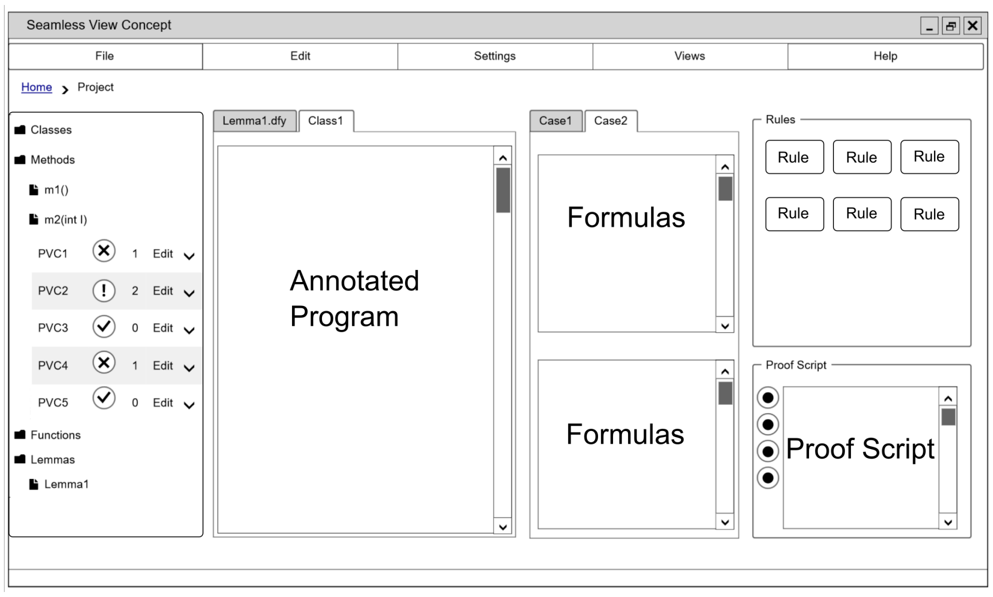


Figure 10.1.: Abstract sequential arrangement of views for the seamless view concept

10.3.1. System and Proof Overview

The *system and proof overview* is the starting point for every verification attempt. The main goal is that the user gains an overview over the verification target, the properties that should be verified as well as the verification progress.

The user should be able to switch from the abstract context information to the more focused single verification tasks using this view.

Exploring the project is one of the major activities in this view: the user should be able to browse the system structure, dependencies of the different subsystems and proof artifacts in order to identify the next verification tasks similar to the project browsing in an integrated development environment. This browsing activity should support the user in building a mental model of the system and the proof task.

We have decided to depict the hierarchical structure of the components of the verification target in a tree view as usually found in IDEs. Besides these (sub)systems of the verification target, we also include lemmas written by the user in this tree view.

The tree structure has to be collapsible and expandable to allow the user to either gain a more abstract overview or to be able to focus on single components of the system. To also already gain an overview over the verification progress we chose to depict the number of proven verification conditions next to the system's components.

The dependencies between constituents of a system are not all shown at once, but can be accessed by the user upon request, to maintain the clear arrangement for the user. To allow inspecting call dependencies in this view, we devise to use a context menu-entry that can be accessed when right-clicking onto the (sub)system. Using this functionality should allow users to retrieve a view similar to the call-hierarchy view of IDEs where the call dependencies are listed and can be selected for further inspection.

If the user selects a subsystem in the list view, the view changes: If a subsystem is selected that can be further divided, its subsystems are shown. If the user chooses a subsystem that cannot be decomposed further into subsystems, the concerns, if more than one exists, are added as successor nodes to the subsystems in the view, i.e., they are expanded in the tree view. When selecting a concern, the corresponding PVCs are shown as leaves in the tree structure. The PVCs are represented in a list view as depicted in the left part of the Figure 10.2. An example for a PVC in this figure is `sort/InitInv[kInBounds]` which denotes the verification condition to prove that the loop invariant with the label `kInBounds` holds initially.

To support the user in relating a PVC to the annotated program we have chosen to use identifiers for each PVC that contain the path information of the symbolic execution path they represent, followed by an identifier that captures the PVCs purpose. If an annotation is a composition of different properties a string identifying the part of the annotation is added as suffix. Furthermore, we also support the user in adjusting and naming the identifiers for the PVCs by allowing to label the annotations. This label is then used in the PVCs identifier instead of a default numbering that identifies the annotation's position among the other annotations.

Example 10.1. Consider the case where it has to be shown that a loop invariant initially holds as an example. Here, the name of the corresponding PVC ends with the string `InitInv`. An example for a user defined label that is used in the name of a PVC can be seen in Fig. 10.2: the label `kInBounds` of the selected PVC can be found in the loop invariant annotation in line 72.

If the user needs details either about the system itself or the PVCs we have chosen to use the same representation that is used in file explorers, where the contents of a selected file is depicted in a second window. In our case, the details are shown in the *source code view* in the context of the selected system.

To allow for an overview over the proof progress, for proof inspection and proof construction already on the abstract system level, we included further means for interacting with the PVCs in the tree structure. Next to the PVCs, the users are able to see an indicator of the proof status of a PVC, the number of proof branches in a proof for a PVC and a possibility to select the PVC for proof construction on the local level.

In the left part of Figure 10.2 next to the PVCs an icon is presented showing the state of the proof for the PVC. A green check mark indicates that the proof is closed, a red exclamation mark indicates an open proof and a yellow warning sign indicates an outdated proof.

Next to the subsystem `method sort` the number of PVCs for the proof of the requirement specification of `sort` is shown together with the number of already proven PVCs as indication about the proof state for the proof of the requirement specification of `sort`.

10.3.2. The Source Code View

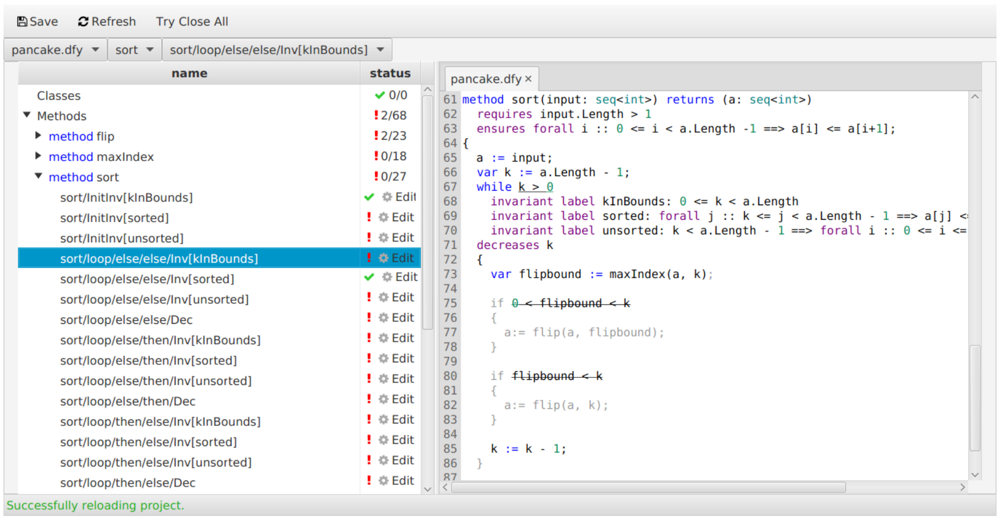
We have chosen to show the program and its annotations in one file in a tabbed pane in the source code view (see Fig. 10.2 right side), despite the disadvantage of a cluttered source code. The reason for this decision is that our goal is to support the user in finding and inspecting relations and dependencies between the different proof artifacts. It is then a natural choice to present the two proof input artifacts together, to illustrate the close relationship between the proof input artifacts by presenting them close to each other, following the proximity principle from the Gestalt Principles [Wer23]. Users are supported in developing the auxiliary specification relative to the requirement specification by directly seeing both types of specifications, together with the program at the same time and they can directly place annotations at the program location, where they should be checked.

The tabbed pane furthermore allows opening and inspecting different parts of the system's implementation. We have included syntax highlighting for the source code and the annotations as a visual guidance for the user. Support for editing the source code and the annotations is also present in the form of syntax checking with a visual highlight of the location containing a syntax error. The usual supporting actions for text editing, such as *copying* and *pasting* text fragments can be accessed using the shortcuts common in text editors.

As soon as users edit the proof input artifacts in the source code view, the contents of *all* other views are disabled as the information presented in these views is outdated. This decision has been made to maintain a consistent state of all proof artifacts. In many cases it would be possible to compute exactly which artifacts are affected by the change in the background and disable only the invalidated parts, this may however disturb the user while editing, i.e., the user may make small mistakes, such as typing the wrong symbol, while editing and each correction would lead to a new computation of affected artifacts. We therefore chose to not disturb the user and recompute all artifacts after the user has finished editing and saved the changes. The assumption behind this decision is that users indicate through the save changes action that they are either reasonably sure that the annotation is the right one, or that they want the system to check the annotation.

Depending on the kind of annotation the user is working on, he needs either information about the local states or about the global context. For example, if the user is editing the contract of a method, he might be more interested in the context this method is called in. If the user is trying to prove a loop invariant he is more focused on the invariant cases and their states and not necessarily on the global call context of the method.

Relevant information that users might be interested in are thus call-contexts of (sub-)systems or usage contexts of annotations, i.e., the locations in the system where an annotation may be used to prove the requirement specification of a calling method, while editing or inspecting the annotated source code. Presenting this information in full detail would clutter the screen, and we therefore consider it as an action in this view that can be invoked by right-clicking onto the system's declaration. This results in a new window displaying the context information to the user. Using an



name	status
Classes	✓ 0/0
Methods	! 2/68
method flip	! 2/23
method maxIndex	! 0/18
method sort	! 0/27
sort/InitInv[kInBounds]	✓ Edit
sort/InitInv[sorted]	! Edit
sort/InitInv[unsorted]	! Edit
sort/loop/else/else/Inv[kInBounds]	! Edit
sort/loop/else/else/Inv[sorted]	✓ Edit
sort/loop/else/else/Inv[unsorted]	! Edit
sort/loop/else/else/Dec	! Edit
sort/loop/else/then/Inv[kInBounds]	! Edit
sort/loop/else/then/Inv[sorted]	! Edit
sort/loop/else/then/Inv[unsorted]	! Edit
sort/loop/else/then/Dec	! Edit
sort/loop/then/else/Inv[kInBounds]	! Edit
sort/loop/then/else/Inv[sorted]	! Edit
sort/loop/then/else/Inv[unsorted]	! Edit
sort/loop/then/else/Dec	! Edit

```
method sort(input: seq<int>) returns (a: seq<int>)
  requires input.Length > 1
  ensures forall i :: 0 <= i < a.Length -1 ==> a[i] <= a[i+1];
{
  a := input;
  var k := a.Length - 1;
  while k > 0
    invariant label kInBounds: 0 <= k < a.Length
    invariant label sorted: forall j :: k <= j < a.Length - 1 ==> a[j] <
    invariant label unsorted: k < a.Length - 1 ==> forall i :: 0 <= i <=
  decreases k
  {
    var flipbound := maxIndex(a, k);

    if ~~0 < flipbound < k~~
    {
      a:= flip(a, flipbound);
    }

    if ~~flipbound < k~~
    {
      a:= flip(a, k);
    }

    k := k - 1;
  }
```

Successfully reloading project.

Figure 10.2.: Arrangement of *system and proof overview* (left) next to the *source code view* (right). In the system and proof overview a PVC is selected and the corresponding path through the program is highlighted in the source code view.

extra window for this action has the advantage that users can keep the information next to the source code as long as they need it and are able to move the information to the position most useful for them.

We consider the context information in this new window to be presented similarly to the presentation of call hierarchies in common IDEs: the entries are sorted by their access type (i.e., writing or reading) for systems together with the system where the access is taking place. When accessing usage information for annotations we can distinguish two cases: either the annotation of a called system is of interest or the usage information about the annotations currently in user focus is of interest. Both types of information should be accessible by using context menu entries in the source code view. The presentation should be different, however: while for the latter case the requirement annotation should be shown to the user, for the first case all locations where the system is called need to be listed. All context information should be presented in tree views, where details can be collapsed if not needed to achieve visual clarity and to support structuring of the accessed dependencies. To further support a more detailed inspection, the selection of an entry in this new window should result in opening a new tab in the source code view containing the selected system and its annotations.

10.3.3. The Interplay between System and Proof Overview and Source Code View

Our goal with the arrangement of *system and proof overview* and *source code view* next to each other was that the user is able to see all PVCs that need to be shown for a concern and is able to inspect the implementation and annotations of a (sub)system on one screen.

Selecting a system in the system and proof overview results in the presentation of the corresponding implementation in the source code view. Users retrieve a presentation of the PVCs for a system and a PVC's relation to the program control structure by selecting a PVC. Upon selection, not only the corresponding implementation is shown but also those statements are highlighted that correspond to the symbolic execution path for the PVC, as well as annotations that are assumed or need to be shown for this path (see Fig. 10.2). We chose to gray out the remaining context if a PVC is selected instead of hiding it completely. Thereby users still have context information available, e.g., about the control flow structures of the systems, if needed for building up their mental model. Selecting lemmas in the system and proof overview also results in a new tab added to the source code view in which details about the lemma are shown and can be edited.

The combination of both views allows users to switch between the global level and the local level of a concern and provides means to access necessary context and dependency information. On the global level means to access and edit requirement specifications are present. Users can choose to use lightweight tools on a concern to already try to prove the program against these requirements, without being overwhelmed with details. On the local level, users can edit auxiliary annotations, as well as inspect single PVCs in their context for editing them. Switching between both levels is supported by displaying hierarchical context information and allowing users to access further contextual information, such as information about the usage of annotations in a call context, upon request. Interactions in this combination of views include retrieving dependency information, editing annotations, and performing proofs.

Proofs in these two views are *auto-active* proofs, i.e., the user adds annotations and lets a prover discharge resulting PVCs. However, in contrast to purely auto-active systems, we allow the user to gain a more fine-grained control over the proof construction. Proofs are performed by selecting either the system (then the process is similar to a pure auto-active proof process) or a PVC in the *system and proof overview* and choosing to either directly call a lightweight prover, or use a general proof strategy, which can be user-defined, to discharge the verification conditions. As the user can choose which kind of proof strategy to use we allow for more flexibility in the proof construction compared to pure auto-active systems.

10.3.4. Logical and Proof Construction View

The *logical and proof construction view* (shown in Fig. 10.3) contains different parts: a view showing the logical representation of a PVC (which we will simply refer to as *logical view*) in a form similar to a sequent and a *proof construction view* that contains a proof script and details on applicable proof rules and strategies.

The intention of combining both views is to support the user in the task of proof construction for a single PVC by using direct manipulation and script-based interaction interchangeably. We have combined both sub-views to one view as the artifacts shown in the views are closely related. They all support the user in constructing a deductive proof on the most detailed representation of a proof state.

The *logical view* contains the logical representation of a selected PVC if no proof has been performed yet. During proof construction for a PVC the logical view shows the current open goals of the proof. Conditions that are assumed in the proof state are grouped above the turnstile symbol ($\vdash$) in the topmost part of the view and the conditions that need to be shown are grouped at the bottom. The syntactical representation of the conditions is chosen to be as close as possible to their syntactic representation in the annotated program.

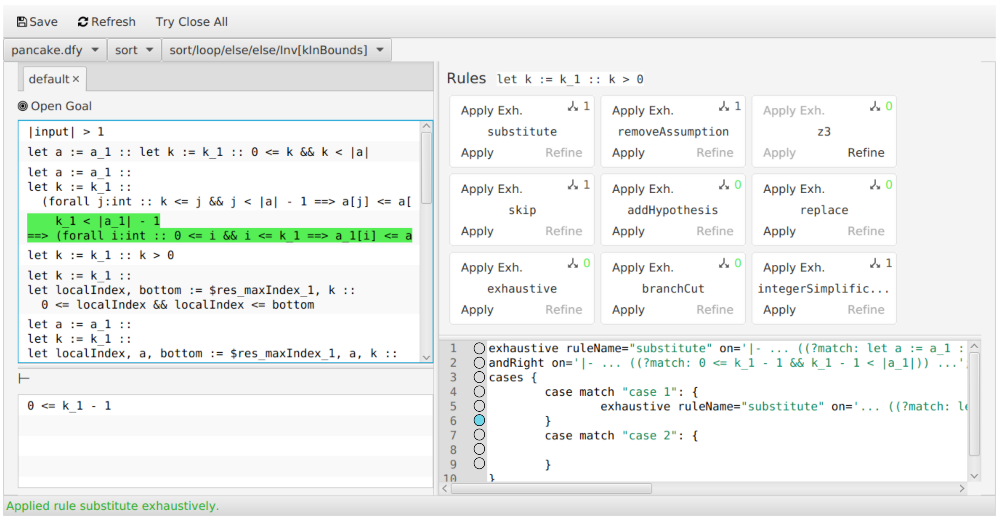
To help the user with inspecting the logical view on the proof state, we consider in our concept to provide interactions to arrange the formulas in the view. To allow users to structure the representation according to their needs, it should be possible to rearrange and group formulas. The grouped entities can be given a user-defined name and moved within the representation to different locations.

A proof for a PVC may branch, for example, because of a case distinction in the proof. A user then has to be able to access all proof branches. To identify and refer to the different branches, rules provide branching labels that are attached to the newly created branches. If a proof for a PVC has more than one branch, the logical view is tabbed, and in each tab the open goal of a branch is presented. The name of the tab corresponds to the branching label in the proof and allows the user to easily identify the open goal of a branch. Users can give labels to user-defined lemmas and rules to be able to use them for the identification of proof branches during proof construction.

Assignments and other state updates in the program are translated to let expressions as seen in the example state in Fig. 10.3. During proof construction, these let expressions are applied to formulas by replacing the left-hand side of the declaration of the let expression with the right-hand side of the declaration in the scope of the let expression. Let expressions may be nested and more than one let expression may be applied to one formula. For the user to be able to reconstruct the origin and update history of a formula to which (possibly nested) let expressions have been applied to, we devise an additional view that shows the different application steps of let expressions for a given formula in a structured list view. Alternatively, the user would also be able to retrace the origin using the navigation possibilities offered by the script view, we however chose to allow for an additional inspection possibility. For visual clarity, we provide this view as overlay if the user selects the corresponding formula.

In the logical and proof construction view, proof construction can be performed using the two interaction styles direct manipulation and text-based interchangeably, allowing flexibility in the interaction.

Direct Manipulation Interaction. If the user chooses to use direct manipulation for proof construction, all applicable rules and lemmas for a selected position are shown above the proof script (in Fig. 10.3, the right upper panel), in the *rule grid*.



```
|input| > 1
let a := a_1 :: let k := k_1 :: 0 <= k && k < |a|
let a := a_1 ::
let k := k_1 ::
  (forall j:int :: k <= j && j < |a| - 1 ==> a[j] <= a[
    k_1 < |a_1| - 1
==> (forall i:int :: 0 <= i && i <= k_1 ==> a_1[i] <= a
let k := k_1 :: k > 0
let k := k_1 ::
let localIndex, bottom := $res_maxIndex_1, k ::
  0 <= localIndex && localIndex <= bottom
let a := a_1 ::
let k := k_1 ::
let localIndex, a, bottom := $res_maxIndex_1, a, k ::
```

```
0 <= k_1 - 1
```

```
let k := k_1 :: k > 0
```

```
exhaustive ruleName="substitute" on='|- ... ((?match: let a := a_1 :
andRight on='|- ... ((?match: 0 <= k_1 - 1 && k_1 - 1 < |a_1|)) ...'
cases {
        case match "case 1": {
                exhaustive ruleName="substitute" on='... ((?match: le
        }
        case match "case 2": {

        }
}
```

Applied rule substitute exhaustively.

Figure 10.3.: The *logical and proof construction view* showing the partial proof for a PVC. The script shows a proof state with two proof branches, each branch can be selected using the tabs in the tabbed view or the corresponding `case` in the script, by selecting the line or the circle next to the line.

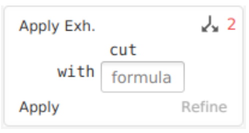


Figure 10.4.: The rule application panel for the *cut* rule.

The rules are shown as panels (*rule application panels*) with different interactive elements and labels (an example is shown in Fig. 10.4). A rule always has a name (in Fig. 10.4, the rule's name is cut) and the number of proof branches it produces when applied to the selected position (the right upper corner of Fig. 10.4). The rule application panels have different interactions available. If a rule needs parameters, for example the cut rule that needs the cut-formula or a quantifier instantiation rule that needs terms that are used for instantiations, the rule application panels contain a form fillin interaction for the parameters below their name.

There are different modes in which a rule may be applied. Examples are whether a rule may be applied only once, or as long as it is applicable to the selected position. A further mode is to apply a rule globally everywhere it is applicable in the logical representation. The application mode, and other settings, which may be rule-specific can be selected by the user in the drop-down menu in the upper left corner of the rule application panel (in Fig. 10.4, the mode is selected where a rule is applied to one position as long as applicable).

In the lower part of the rule application panel two buttons are placed. One button that allows applying the rule (the lower left corner of Fig. 10.4) and one that refines a rule (the lower right corner of Fig. 10.4). Refining a rule updates the branching information of the respective rule, by invoking a lightweight prover, such as an SMT solver, in the background which tries to close the branches resulting from the rule's application. In case the prover was able to close the proof branch, the branching information is updated with the number 0 to indicate a closed branch. This allows the user to select rules that are likely to yield a closed proof or less proof branches to work on.

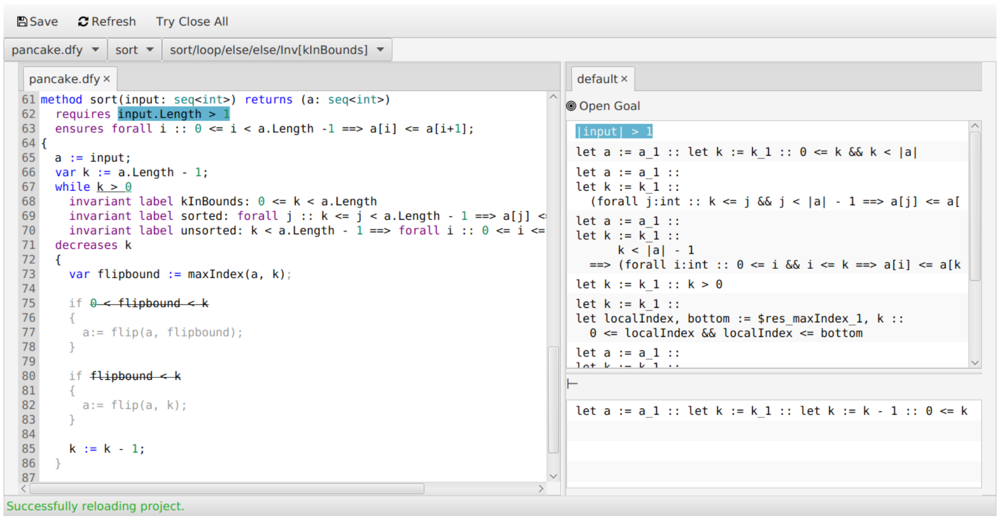
We allow users accessing the result of a rule application before applying it, for two reasons: firstly, to prevent users from accidentally applying a series of rules that may be complex to reverse: a user may rate a proof state to be a promising intermediate state towards a successful proof. After applying a series of rules the proof may be guided into a direction that does not seem goal-oriented anymore. Finding back to the state rated as promising may require to revert a whole subproof. The second reason is to allow users inspecting the result of a rule application to determine whether applying a specific rule is a promising step in the proof process. Selecting a rule application panel without applying the rule, shows its results in the context of the logical representation. Formulas that will be deleted are highlighted in red and formulas that are added or changed are highlighted in green. This information also remains in the logical view after the application of a rule, such that users are able to directly inspect the differences while navigating a proof.

Further support to choose the right rule is given by sorting the rules in the rule grid according to their estimated effect on the complexity of the resulting proof. Rules which close the proof are shown first, followed by rules that result in only one branch. Branching rules with several branches are put at the end of the list of rules. We chose this order for the rules, as the more branches a rule produces the more interaction the user may perform in order to close the proof.

Script-Based Interaction. As already described in Chapter 8.2, direct manipulation has the disadvantage of not persisting the user interactions and the issue that previous performed actions cannot be easily accessed. We counteract this disadvantage by allowing the user to use both styles interchangeably and by generating the proof script from the direct manipulation interactions of the user – manually applying a rule results in adding this interaction to the proof script as proof command.

The scripting language in the script view is a subset of the language presented in Chapter 7, without the repetition statements. We excluded the repetition statements as we chose to pass parameters to the rules that indicate whether they should be applied repeatedly. Furthermore, when adding rules from direct manipulation interactions, a matching expression is generated that encodes the position in the proof goal to which the rule should be applied to and it is persisted as parameter for the proof command representing that rule in the script.

The proof script reflects the proof structure and allows for means to navigate a proof for a PVC. As introduced, proofs can branch, and the applied rules generate branching labels to name the branches. These branches are reflected in the script by adding a `cases` statement containing `case` statements with matching expressions over the labels.



```
method sort(input: seq<int>) returns (a: seq<int>)
  requires input.Length > 1
  ensures forall i :: 0 <= i < a.Length -1 ==> a[i] <= a[i+1];
{
  a := input;
  var k := a.Length - 1;
  while k > 0
    invariant label kInBounds: 0 <= k < a.Length
    invariant label sorted: forall j :: k <= j < a.Length - 1 ==> a[j] <
    invariant label unsorted: k < a.Length - 1 ==> forall i :: 0 <= i <=
  decreases k
  {
    var flipbound := maxIndex(a, k);

    if ~~0 < flipbound < k~~
    {
      a:= flip(a, flipbound);
    }

    if ~~flipbound < k~~
    {
      a:= flip(a, k);
    }

    k := k - 1;
  }
```

```
|input| > 1
let a := a_1 :: let k := k_1 :: 0 <= k && k < |a|
let a := a_1 ::
let k := k_1 ::
  (forall j:int :: k <= j && j < |a| - 1 ==> a[j] <= a[
let a := a_1 ::
let k := k_1 ::
      k < |a| - 1
  ==> (forall i:int :: 0 <= i && i <= k ==> a[i] <= a[k
let k := k_1 :: k > 0
let k := k_1 ::
let localIndex, bottom := $res_maxIndex_1, k ::
  0 <= localIndex && localIndex <= bottom
let a := a_1 ::
```

```
⊢
let a := a_1 :: let k := k_1 :: let k := k - 1 :: 0 <= k
```

Successfully reloading project.

Figure 10.5.: The *source code view* (left) and the *logical view* (right) showing the path corresponding to a selected PVC and the logical representation of the PVC. The relation between the selected formula in the logical view and its origin in the annotations is highlighted.

In some cases, users may want to inspect the proof performed so far, similar to the way it is possible in systems like KeY or KIV. Instead of presenting a proof tree to navigate the current proof, the script can be used to navigate through the proof of a PVC. To achieve this navigation feature, next to each statement symbols are added that serve as interactive handles. Selecting the handles before or after statements (in Fig. 10.3 the circles next to the line numbers of the script) in the proof script shows the state before or after the execution of the script statement. If the user chooses to advance the proof by using the proof script, textual interaction is possible in the script view. Basic support such as syntax highlighting and syntax checking is available in this view. Similar to changing the annotated source code the script is not executed in the background but rather the user has to invoke the execution, to prevent the execution of an incomplete script.

10.3.5. Interplay between the Source Code View and the Logical and Proof Construction View

Our goal is to support the user in relating the proof input artifacts to the logical representation for the deductive proof construction by placing the source code view and the logical and proof construction view next to each other and providing further means to inspect the relations between the representations. Presenting both views in full detail may however clutter the screen, we therefore chose to only show the logical view next to the source code view (as depicted in Fig. 10.5).

This view serves as pre-stage to the proof construction view and provides means for the user to first relate the relevant parts of the proof input artifacts to the logical representation before fully switching to the logical and proof construction view.

In the source code view the user can inspect the path through the program for the selected PVC as well as all annotations that are considered for the proof of the PVC. In the logical view the user can inspect the formulas that result from the transformation of the system to its PVCs. Although special care is taken to represent the formulas syntactically close to the representation in the proof input artifacts, this is not always possible. Especially with progressing proof construction the formulas may change significantly. To still allow for a relation between both representations, the user can select a formula or sub term in the logical representation and use a shortcut to retrieve the origin information in the context of the proof input artifact, if available. Our realization of this feature is depicted in Fig. 10.5.

From the pre-stage users can change to the full logical and proof construction view, where the origin relation is available within the proof respectively along the proof states. In this view users additionally retrieve information about the proof step that is responsible for introducing the selected formula in the current representation in the logical view, if the proof is already progressing. This is realized by highlighting the corresponding statement in the proof script. This then allows users to select the script statement and inspect the state changes in the logical view. As origin relation we consider a similar concept as the one presented for our user study (in Chapter 5). In the user study, participants wanted a relation to the proof input artifacts and a more fine-grained origin relation compared to the one presented to them in our user study.

The first origin of a formula may be in the proof input artifacts. We show this information in the logical and proof construction view by annotating formulas with abstract labels such as “precondition”. These labels can be accessed using the context menu in the logical view.

10.4. Conclusion and Future Work

In this chapter we have presented a concept for a user interface for a seamless interactive program verification process. The concept is based on results from our user study (presented in Chapter 5) as well as general usability principles and principles for theorem provers. One of the main goals of the concept is to support users in step-wise focusing on the different parts of the proof artifacts for inspection and proof construction, as well as the possibility to seamlessly switch to more abstract presentations if necessary. At the same time users are supported in the inspection of relations between the different proof artifacts. The concept also integrates the three interaction styles text-based, direct manipulation and auto-active such that they can be used interchangeably. These styles are the prominent interaction styles for proof construction in current theorem proving and verification systems.

A concretization of the concept for a user interface for a verification system is presented in this chapter as well. The user interface contains parts of the proof artifacts in different views. The arrangement of the views is chosen in a way to support users in step-wise focusing on more detailed parts of the proof artifacts. For this reason, always only two adjacent views are considered to be shown together to the user on the screen. The goal of the arrangement is to support the users in carrying over information about relations and dependencies between different proof artifacts from one view to the other, by trying to keep the cognitive load low.

Further support that users should invoke on demand for inspecting relations and dependencies is presented in the concretization of our concept as well. We did not only present the concepts, parts of the concept are implemented in an early prototype.

It remains for future work to integrate the missing features, like the additional view showing the call or usage dependencies, or proof exploration techniques for the logical view. Furthermore, following the user-centered design process, evaluations for the effectiveness in the form of case studies as well as user studies for the interaction concepts and the user experience should be performed to be able to enhance the presented concept and prototype. The user studies should also concentrate on evaluating whether our restriction in the work-flow may need enhancements for expert users, for example by adding means to directly jump from the system and proof overview to the logical view.

To evaluate whether the integration of the three interactions styles is enhancing the user support, we devise to perform a comparative evaluation using the prototype. For such an experiment the participants should grouped in to different groups. Each group should either be allowed to use only one interaction style or a combination of styles for proof construction. The respective views then have to be inaccessible by the corresponding group for the experiment. The tasks should be the same for all groups and the task completion time should be measured as well as the user experience using standardized questionnaires, such as UEQ [LHS08] or SUMI [Kir]. These questionnaires should be accompanied by a small number of open questions to explore room for improvement.

Room for improvement, which we already consider after first experiences with the first prototype concerns the parameter representation for the script language and the language constructs. One example is to add functions that allow matching terms and assigning different matching results to script variables. Also, technical issues, such as a incremental loading process that only reloads those parts of the system that are affected by changes, need to be developed to allow for an efficient use of the prototype.

10.5. Related Work

Different state-of-the-art verification systems inspired the development of our seamless verification concept.

The direct manipulation interaction was mainly inspired by the KeY system [Ahr+16]. KeY allows users to select a position in the sequent where a rule should be applied to. Users then get suggestions for applicable rules in a context menu and can retrieve a tool tip that contains information about changes to the position if applying the selected rule.

KeY also contains first support for inspecting relations between proof artifacts. In its current version, KeY depicts the relation between the proof input artifacts – the source code – and the proof state using an additional window. This window was implemented in response to our user study KeY. In this window the annotated program is presented and the statements are highlighted that have been symbolically executed for an open goal. However, modification of the annotated source code can at the moment only be done by using an external text editor and requires a restart of the verification process.

Another system that in particular inspired our system overview and the hierarchical structuring of the proof of a concern, e.g., by splitting a concnern into single PVCs, is the Why3 platform. In Why3 the concern is transformed into proof verification conditions, which are presented in a hierarchical structure to the user. It is possible to use different means to discharge the proof verification conditions. Besides the use of SMT solvers, it is possible to perform interactive verification using the Coq system [BC04]. The relation between the verification conditions and the proof input artifacts is presented by highlighting relevant statements in the proof input artifacts. The relation between the formula in Coq and the proof input artifacts get lost during transformation. Furthermore, the user needs to switch between whole systems and thus user interfaces, which requires the user to gain orientation and to perform a context switch, which may be cognitively challenging.

VCC [Dah+09] and Dafny [LW14] inspired our concept in two ways. Firstly, these tools allow for auto-active proof construction and provide feedback about the result of a verification attempt on the program level. In these systems with their integration into an IDE the relation to the proof input artifacts is directly visible for the user. Users retrieve information about conditions that do not hold on specific program locations and retrieve the values of variables for a program path for inspection. Secondly, we chose to use the Dafny language in our prototype. Dafny is suitable for real-world applications, without the language scope of an established programming language that needs to be backwards-compatible.

Systems that allow for different interaction styles in combination include KeYmaeraX, where the user is able to use direct manipulation for proof construction as well as text-based interaction interchangeably. Proof exploration is supported by using different functionalities: users can retrieve suggestions for rule applications and a detailed view containing information about state changes when applying a rule. Furthermore, counterexamples and simulation can be generated. The current proof state is represented in a tabbed view, where each tab contains the sequent of an open goal, similar to our concept. However, instead of showing a proof tree, in KeYmaerax, deduction paths are shown, which can be retrieved by step-wise expanding details of the path in a tab, starting at the open goal. This solution also allows focusing on the different steps on a proof branch. With progressing proof, this view may become cluttered as a deduction path may contain a large number of proof steps. Structuring of proof goals is also possible in KeYmaeraX: users can change the sequent view by hiding formulas.

The KIV verification system [Han+05; Bal+00] also presents different views with different purposes to the user. The user has access to the proof tree, the current goal, editors for the theorems and the specification as well as managing the theorem base. Each view should support users in specific tasks, thus each view contains actions and features that are necessary for the specific task, e.g., the current goal view shows the applicable rules and has context-sensitive support for rule application using direct manipulation.

Similar to the KeY system, the representation of the proof object in KIV is the proof tree which is the central structure for proof comprehension and proof construction. KIV allows for direct manipulation with context-sensitive rule suggestions, similar to KeY. KIV contains a correctness management that takes care of the dependencies affected by changes of theorems and lemmas supporting users in the iterative proof process. KIV is integrated into the IDE Eclipse, which allows to use the usual support for writing code. The project can then be opened in KIV allowing the access to the aforementioned views and actions.

Part IV.

Conclusion

11. Conclusions

Formal methods, such as program verification, can be used to ensure correct functionality and increase the quality of software systems. Program verification is performed by using a proof system to verify that an implementation ensures formally specified properties. The problem whether a program fulfills a non-trivial property is in general undecidable. However, recent improvements in the performance and capabilities of state-of-the-art verification systems enable that more programs and properties can be proven automatically.

For *complex* programs or *complex* properties this problem is still only solvable with a certain degree of user intervention. The user has to guide and control the proof process and find a suitable specification that is sufficient for the proof of the correctness of a software system in an iterative, time-consuming process. In order to improve the current state-of-the-art in interactive deductive program verification one question is thus from the user's perspective which aspects of interactive program verification makes this task complex and how can users be supported by the verification system?

Our first hypothesis for this thesis was that there exists a gap between the actual proof performed by a verification system and the user's mental model about the proof. To investigate the aspects relevant for the time-consuming verification process we followed a two stage process: in the first part of this thesis we have conducted two focus group discussions with users of the generic proof assistant Isabelle/HOL and with users of the KeY system to get first insights into the aspects that contribute to the complexity of the verification task. We explored the context of use in interactive verification systems, in particular we investigated the user interaction by first conducting focus group discussions. Based on first insights gained in the focus group discussions we conducted semi-structured interviews with practical tasks where we were able to observe user interaction in parts of the verification process.

In these user studies we were able to observe that users of KeY have many degrees of freedom to interact in the verification process. The aggregated sequence model that contains the activities of all participants has shown that there is not one single verification process, but the processes differ for each user. Despite these many degrees of freedom, we were able to identify common patterns and relevant activities in the verification process, such as inspection of the proof state and the proof input artifacts, as well as the automatic and the controlled proof search.

We were also able to find evidence for the gap between the user's model of the proof and the actual proof performed by the system in the focus group discussions, as well as in the interview sessions. To find the orientation in a program verification proof that was partly constructed automatically, users often tried to find the relation between the proof state and the proof input artifacts. Furthermore, users tried to retrace the proof that was conducted by the system. Both tasks can be time-consuming.

We were also able to observe the orientation process in detail, where participants first tried to gain an overview over the proof by hiding details of the proof tree and performed a kind of *zooming-in* process into details of the proof. Similar to the usability principle identified for interactive theorem provers by Easthaughffe [Eas98], we have seen the need to provide users with different views onto the proof state. In Chapter 8 and Chapter 10 we have instantiated this principle for program verification systems with explicit proof object.

In the second part of this thesis we have developed improvements for the user interaction in deductive interactive program verification systems based on the evaluation results. The developed improvements cover different aspects of the user interaction in the verification process.

Firstly, we have developed an interaction concept based on program verification systems that offer an explicit proof object to the user and employ direct manipulation interaction for proof construction. This interaction concept integrates the two prominent interaction styles for proof construction: script-based interaction and direct manipulation. The goal is to leverage advantages of one style (e.g., the control-flow possibilities in the script-based style) to be able to mitigate the disadvantages of the other style (e.g., the high effort for repetitive interactions in the direct manipulation style). To enable this integration, we have developed a general proof scripting language with a flexible selection concept for proof goals, based on the observations of actions during the user studies. Together with this language we have developed an interaction concept for the integration of both – script-based interaction into direct manipulation – by presenting an analogy between software debugging and analyzing open proof attempts. This analogy enabled us to adapt concepts from software debugging and analyzing open proof attempts and allowing similar degrees of freedom for the orientation in the proof as we were able to observe during the user study.

Based on further observations in the user studies, we developed another interaction concept that supports users in the *zooming-in* process for proof comprehension and proof construction. In this concept the goal is to enable interaction on different representations of the proof problem: (a) on the proof input artifacts by using the auto-active style, (b) on the generated proof verification conditions (PVC), by using direct manipulation and the possibility to select all or single PVCs and apply either a general proof search strategy or a user-generated script to prove the PVCs and (c) on the logical representation of the PVC by using direct manipulation and script-based user interaction interchangeably. Furthermore, in this concept the proof state is structured into multiple views and support to relate objects across the different views is devised. With the presented concept we followed the intent that users are supported in relating different proof input artifacts and at the same time can choose their preferred way of interacting with the verification system according to the current proof situation.

We were also able to observe different actions in the user study that can be characterized as being explorative, i.e., actions with the intent to gain knowledge about the current proof state that may be reverted afterward and actions that seemed like a *try-and-error* approach for different proof search strategies. Such actions served the purpose to form a hypothesis about the proof state or to determine the cause for a failed proof attempt, to be able to decide the next promising step in the proof process. One of these steps is to change parts of the proof input artifacts.

In the systems we examined, changing the proof input artifacts amounts to abandoning the current proof, changing the artifacts and starting a new proof attempt. The user therefore may need to reapply steps performed in the new proof attempt and try to gain the orientation again in the new proof. If the change of the proof artifacts was not the cause for the failed attempt, the user has invested effort and time. To support the user in testing hypotheses about the proof state and to minimize the number of reiterations of the verification process, we developed a concept for proof exploration in a verification system implementing a sequent calculus. In this concept users can alter the proof state according to their hypotheses about the state and proceed in the verification process without starting a new attempt.

We added different supporting features to this concept, such as the possibility to trace back the origin of terms and formulas, especially if their origin are the specification annotations. Further support is given by an access to the performed exploration actions, e.g., to be able to revert them if their application was not helpful. Special care has been taken that users are not able to perform unsound actions through usage of the exploration mode.

Concerning future work, the concepts presented in this thesis need to be fully realized and their effect on the usability of the verification system needs to be evaluated. It has to be further investigated whether the proof scripting language and the interaction concept for KeY supports all necessary actions for proof construction and proof comprehension. To complement the qualitative studies performed in this work, a larger *quantitative* user study, together with case studies should be performed to investigate potential room for improvement. The proof script debugger should then be fully integrated into the user interface of KeY, instead of being a standalone user interface. Furthermore, it should be investigated whether to embed KPS into a regular scripting or programming language to benefit from common features of these languages such as complex, iterable data structures.

The prototype implementing the seamless interaction concept should be finalized and the presented supporting features that are not included yet should be implemented. Using this implementation, qualitative evaluations should be performed to investigate further room for improvement and whether all necessary user interactions are realized. Furthermore, this prototype may then be used to perform user studies to compare the different interaction styles.

With the results presented in this thesis, we provided building blocks for an improved user support in interactive deductive program verification systems. We believe that our seamless interaction concept, together with a further developed proof exploration mode has the potential to provide expert users with a tool to effectively and efficiently perform interactive software verification.

In the interactive verification process, users often already have a coarse proof idea or sketch in mind when starting or continuing a verification task – maybe in form of a (partial) mental proof plan or several approaches they might pursue to advance the proof. We believe a usable verification system has to both support transforming this abstract proof sketch to suitable prover guidance (e.g., using proof scripts) and also to determine which of the conceived proof approaches are productive without incurring high user effort (e.g., using proof exploration activities).

Together with our concepts to improve on proof state visualization and user guidance, we argue that these improvements of the usability of verification systems may also lower the entry barrier for more novice users.

However, to enable that interactive deductive program verification is applicable in a large-scale industrial context, e.g., by software engineers, a usable verification system or specification language alone is not sufficient and further effort is necessary.

Similar to the software engineering process, a common verification process with a clearly structured approach for the user to follow has to be developed to divide the verification task in more manageable pieces. For the majority of the verification tasks that deal with uncomplicated programs and specifications, standard workflows have to be established to bring software verification closer to an engineering activity. One example for possible improvements in this direction include the creation of specification and proof pattern libraries to simplify these common verification tasks.

Additionally, industrial application of software verification would entail integrating the verification activity into the software development process, which will pose new challenges. Together with the size and complexity of industrial applications, creating a formally proven software system will necessarily be a collaborative effort. As a consequence, further support by the verification tools will be essential. Examples include methods to estimate the time and effort needed to verify a module to be able to plan and manage software projects that include verified components, or further support for the verification of software systems that helps with dependencies between modules of the system (e.g., estimates for the effect of changes in the specification of a module to the already completed verification tasks of other modules that rely on it).

We argue that establishing a structured verification process, together with the improved usability of verification tools based on state-of-the-art verification systems have the potential for a more widespread adoption of deductive verification methods for real-world applications.

Part V.

Bibliography

Bibliography

[AB91] Gregory D. Abowd and Russell Beale. “Users, systems and interfaces: A unifying framework for interaction”. In: *Proceedings of the British Computer Society Special Interest Group on Human-Computer Interaction.* Ed. by D. Diaper and N. Hammond. Cambridge University Press, 1991, pp. 73–87.

[AG16] Wolfgang Ahrendt and Sarah Grebing. “Using the KeY Prover”. In: *Deductive Software Verification – The KeY Book: From Theory to Practice.* Ed. by Wolfgang Ahrendt, Bernhard Beckert, Richard Bubel, Reiner Hähnle, Peter H. Schmitt, and Mattias Ulbrich. Cham: Springer International Publishing, 2016, pp. 495–539. ISBN: 978-3-319-49812-6. DOI: `10.1007/978-3-319-49812-6_15`.

[AH97] Myla Archer and Constance Heitmeyer. “Human-style theorem proving using PVS”. English. In: *Theorem Proving in Higher Order Logics.* LNCS 1275. Springer, 1997. ISBN: 978-3-540-63379-2. DOI: `10.1007/BFb0028384`.

[Ahr+14] Wolfgang Ahrendt, Bernhard Beckert, Daniel Bruns, Richard Bubel, Christoph Gladisch, Sarah Grebing, Reiner Hähnle, Martin Hentschel, Mihai Herda, Vladimir Klebanov, Wojciech Mostowski, Christoph Scheben, Peter H. Schmitt, and Mattias Ulbrich. “The KeY Platform for Verification and Analysis of Java Programs”. In: *Verified Software: Theories, Tools, and Experiments (VSTTE 2014).* Ed. by Dimitra Giannakopoulou and Daniel Kroening. Lecture Notes in Computer Science 8471. Springer-Verlag, 2014, pp. 1–17. ISBN: 978-3-642-54107-0. DOI: `10.1007/978-3-319-12154-3_4`.

[Ahr+16] Wolfgang Ahrendt, Bernhard Beckert, Richard Bubel, Reiner Hähnle, Peter H. Schmitt, and Mattias Ulbrich, eds. *Deductive Software Verification – The KeY Book: From Theory to Practice.* Vol. 10001. LNCS. Springer, 2016. DOI: `10.1007/978-3-319-49812-6`.

[Ait+95] Stuart Aitken, Philip Gray, Tom Melham, and Muffy Thomas. “A Study Of User Activity In Interactive Theorem Proving”. In: *Task Centred Approaches To Interface Design.* GIST Technical Report G95.2. Dept. of Computing Science, 1995, pp. 195–218.

[Ait+98] J. S. Aitken, P. Gray, T. Melham, and M. Thomas. “Interactive Theorem Proving: An Empirical Study of User Activity”. In: *J. of Symbolic Comp.* 25.2 (1998), pp. 263–284. ISSN: 0747-7171.

[Ait96] James Stuart Aitken. "Problem Solving in Interactive Proof: A Knowledge-Modelling Approach". In: *Proceedings of the European Conference on Artificial Intelligence 1996 (ECAI96): 335-339, Edited by W. Wahlster.* 1996, pp. 335–339.

[AL04] David Aspinall and Christoph Lüth. "Proof General meets IsaWin: Combining Text-Based And Graphical User Interfaces". In: *Electr. Notes Theor. Comput. Sci.* 103 (2004), pp. 3–26. DOI: `10.1016/j.entcs.2004.09.011`.

[AM00] J. Stuart Aitken and Thomas F. Melham. "An analysis of errors in interactive proof attempts". In: *Interacting with Computers* 12.6 (2000), pp. 565–586.

[Ann03] John Annett. "Hierarchical task analysis". In: *Handbook of cognitive task design.* CRC Press, 2003, pp. 41–60.

[Bal+00] Michael Balser, Wolfgang Reif, Gerhard Schellhorn, Kurt Stenzel, and Andreas Thums. "Formal System Development with KIV". In: *Proceedings of the Third Internationsl Conference on Fundamental Approaches to Software Engineering: Held As Part of the European Joint Conferences on the Theory and Practice of Software, ETAPS 2000.* FASE '00. Berlin, Heidelberg: Springer-Verlag, 2000, pp. 363–366. ISBN: 3-540-67261-3. URL: `http://dl.acm.org/citation.cfm?id=645368.650817`.

[Bau+12] Christoph Baumann, Bernhard Beckert, Holger Blasum, and Thorsten Bormer. "Lessons Learned From Microkernel Verification – Specification is the New Bottleneck". In: *SSV.* Vol. 102. EPTCS. 2012, pp. 18–32.

[BBG16] Bernhard Beckert, Thorsten Bormer, and Daniel Grahl. "Deductive Verification of Legacy Code". In: *7th International Symposium on Leveraging Applications of Formal Methods, Verification and Validation (ISoLA 2016).* Ed. by Tiziana Margaria and Bernhard Steffen. Vol. I: Foundational Techniques. LNCS 9952. Springer, Oct. 2016, pp. 749–765. DOI: `10.1007/978-3-319-47166-2_53`.

[BBK11] Bernhard Beckert, Thorsten Bormer, and Vladimir Klebanov. "Improving the Usability of Specification Languages and Methods for Annotation-based Verification". In: *9th International Symposium on Formal Methods for Components and Objects (FMCO 2010), State-of-the-Art Survey.* Ed. by Bernhard Aichernig, Frank S. de Boer, and Marcello Bonsangue. Vol. 6957. LNCS. Springer, Jan. 2011.

[BBP13] Jasmin Christian Blanchette, Sascha Böhme, and Lawrence C. Paulson. "Extending Sledgehammer with SMT Solvers". In: *J. Autom. Reasoning* 51.1 (2013), pp. 109–128. DOI: `10.1007/s10817-013-9278-5`. URL: `https://doi.org/10.1007/s10817-013-9278-5`.

[BC04] Yves Bertot and Pierre Castran. *Interactive Theorem Proving and Program Development: Coq'Art The Calculus of Inductive Constructions.* 1st. Texts in Theoretical Computer Science An EATCS Series. Springer-Verlag Berlin Heidelberg, 2004. ISBN: 978-3-540-20854-9.

[Bec+17] Bernhard Beckert, Jonas Schiffl, Peter H. Schmitt, and Mattias Ulbrich. "Proving JDK's Dual Pivot Quicksort Correct". In: *Verified Software. Theories, Tools, and Experiments – 9th International Conference, VSTTE 2017, Heidelberg, Germany, July 22-23, 2017, Revised Selected Papers.* Ed. by Andrei Paskevich and Thomas Wies. Vol. 10712. Lecture Notes in Computer Science. Springer, 2017, pp. 35–48. ISBN: 978-3-319-72307-5. DOI: `10.1007/978-3-319-72308-2_3`.

[Ben05] Catherine Plaisant Ben Shneiderman. *Designing the User Interface: Strategies for Effective Human-Computer Interaction.* Pearson, 2005.

[Ben10] D. Benyon. *Designing Interactive Systems: A Comprehensive Guide to HCI and Interaction Design.* Addison Wesley, 2010.

[Bey17] Dirk Beyer. "Software Verification with Validation of Results". In: *Tools and Algorithms for the Construction and Analysis of Systems.* Ed. by Axel Legay and Tiziana Margaria. Berlin, Heidelberg: Springer Berlin Heidelberg, 2017, pp. 331–349. ISBN: 978-3-662-54580-5.

[BG07] Alan Blackwell and Thomas R. Green. *A Cognitive Dimensions Questionnaire (V. 5.1.1).* `www.cl.cam.ac.uk/~afb21/CognitiveDimensions/CDquestionnaire.pdf`. 2007.

[BG12] Bernhard Beckert and Sarah Grebing. "Evaluating the Usability of Interactive Verification System". In: *Proceedings, 1st International Workshop on Comparative Empirical Evaluation of Reasoning Systems (COMPARE), Manchester, UK, June 30, 2012.* Vol. 873. CEUR Workshop Proceedings. CEUR-WS.org, 2012, pp. 3–17. URL: `http://ceur-ws.org/Vol-873`.

[BG15] Bernhard Beckert and Sarah Grebing. "Interactive Theorem Proving – Modelling the User in the Proof Process". In: *Workshop on Bridging the Gap between Human and Automated Reasoning - A workshop of the 25th International Conference on Automated Deduction (CADE-25).* Ed. by Ulrich Furbach and Claudia Schon. Vol. 1412. CEUR Workshop Proceedings. CEUR-WS.org, Aug. 2015. URL: `http://ceur-ws.org/Vol-1412`.

[BGB14a] Bernhard Beckert, Sarah Grebing, and Florian Böhl. "A Usability Evaluation of Interactive Theorem Provers Using Focus Groups". In: *Software Engineering and Formal Methods – SEFM 2014 Collocated Workshops.* Lecture Notes in Computer Science. 2014.

[BGB14b] Bernhard Beckert, Sarah Grebing, and Florian Böhl. "How to Put Usability into Focus: Using Focus Groups to Evaluate the Usability of Interactive Theorem Provers". In: *Proceedings Eleventh Workshop on User Interfaces for Theorem Provers, Vienna, Austria, 17th July 2014.* Ed. by Christoph Benzmüller and Bruno Woltzenlogel Paleo. Vol. 167. Electronic Proceedings in Theoretical Computer Science. Open Publishing Association, 2014, pp. 4–13. DOI: `10.4204/EPTCS.167.3`.

[BGU17] Bernhard Beckert, Sarah Grebing, and Mattias Ulbrich. "An Interaction Concept for Program Verification Systems with Explicit Proof Object". In: *Hardware and Software: Verification and Testing – 13th International Haifa Verification Conference, Haifa, Israel 13-15, 2017, Proceedings.* Vol. 10629. Lecture Notes in Computer Science. Springer, 2017, pp. 163–178. DOI: 10.1007/978-3-319-70389-3_11.

[BH98] Hugh Beyer and Karen Holtzblatt. *Contextual Design: Defining Customer-centered Systems.* San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1998.

[BK14] A. Butz and A. Krüger. *Mensch-Maschine-Interaktion.* De Gruyter Oldenbourg, 2014.

[BKW16] Bernhard Beckert, Vladimir Klebanov, and Benjamin Weiß. "Dynamic Logic for Java". In: *Deductive Software Verification – The KeY Book: From Theory to Practice.* Ed. by Wolfgang Ahrendt, Bernhard Beckert, Richard Bubel, Reiner Hähnle, Peter H. Schmitt, and Mattias Ulbrich. Cham: Springer International Publishing, 2016, pp. 49–106. ISBN: 978-3-319-49812-6. DOI: 10.1007/978-3-319-49812-6_3.

[BLS05] Mike Barnett, K. Rustan M. Leino, and Wolfram Schulte. "The Spec# Programming System: An Overview". In: *Proceedings of the 2004 International Conference on Construction and Analysis of Safe, Secure, and Interoperable Smart Devices.* CASSIS'04. Marseille, France: Springer-Verlag, 2005, pp. 49–69. ISBN: 3-540-24287-2, 978-3-540-24287-1. DOI: 10.1007/978-3-540-30569-9_3.

[BMR95] Alexander Borgida, John Mylopoulos, and Raymond Reiter. "On the Frame Problem in Procedure Specifications". In: *IEEE Trans. Software Eng.* 21.10 (1995), pp. 785–798. DOI: 10.1109/32.469460.

[BN10] Jasmin Christian Blanchette and Tobias Nipkow. "Nitpick: A Counterexample Generator for Higher-Order Logic Based on a Relational Model Finder". In: *Interactive Theorem Proving.* Ed. by Matt Kaufmann and Lawrence C. Paulson. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 131–146. ISBN: 978-3-642-14052-5.

[Bor14] Thorsten Bormer. "Advancing deductive program-level verification for real-world application: Lessons learned from an industrial case study". Karlsruhe, KIT, Diss., 2014. PhD thesis. 2014.

[BP06] Bernhard Beckert and André Platzer. "Dynamic Logic with Non-rigid Functions". In: *Automated Reasoning.* Ed. by Ulrich Furbach and Natarajan Shankar. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 266–280. ISBN: 978-3-540-37188-5.

[Bus98] Samuel R. Buss. *Handbook of Proof Theory.* Elsevier, 1998.

[Cap90] Stanley Caplan. "Using focus group methodology for ergonomic design". In: *Ergonomics* 33.5 (1990), pp. 527–533. URL: https://doi.org/10.1080/00140139008927160.

[Car03] John M. Carroll, ed. *HCI Models, Theories, and Frameworks: Toward a Multidisciplinary Science.* San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2003. ISBN: 9780080491417.

[Che01] James Cheney. *Project Report – Theorem Prover Usability.* Tech. rep. Report of project COMM 64. 2001. URL: `http://homepages.inf.ed.ac.uk/jcheney/projects/tpusability.ps`.

[CNM83] Stuart K. Card, Allen Newell, and Thomas P. Moran. *The Psychology of Human-Computer Interaction.* Hillsdale, NJ, USA: L. Erlbaum Associates Inc., 1983. ISBN: 0898592437.

[Cok11] David R. Cok. "OpenJML: JML for Java 7 by Extending OpenJDK". In: *NASA Formal Methods.* Ed. by Mihaela Bobaru, Klaus Havelund, Gerard J. Holzmann, and Rajeev Joshi. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 472–479. ISBN: 978-3-642-20398-5.

[Dah+09] M. Dahlweid, M. Moskal, T. Santen, S. Tobies, and W. Schulte. "VCC: Contract-based modular verification of concurrent C". In: *2009 31st International Conference on Software Engineering – Companion Volume.* May 2009, pp. 429–430. DOI: `10.1109/ICSE-COMPANION.2009.5071046`.

[Del00] David Delahaye. "A tactic language for the system Coq". In: *International Conference on Logic for Programming Artificial Intelligence and Reasoning.* Springer. 2000, pp. 85–95.

[Dix+04] Alan Dix, Janet Finlay, Gregory Abowd, and Russell Beale. *Human-computer Interaction.* Prentice-Hall, Inc., 2004.

[Eas98] Katherine A. Easthaughffe. "Support for Interactive Theorem Proving: Some Design Principles and Their Application". In: *User Interfaces for Theorem Provers (UITP 1998)* (1998).

[Ebn+17] Gabriel Ebner, Sebastian Ullrich, Jared Roesch, Jeremy Avigad, and Leonardo de Moura. "A metaprogramming framework for formal verification". In: *PACMPL* 1.ICFP (2017), 34:1–34:29. DOI: `10.1145/3110278`.

[Ell+05] Janice Elliott, Sara Heesterbeek, Carolyn J. Lukensmeyer, and Nikki Slocum. *Participatory Methods Toolkit: A practitioner's manual.* Tech. rep. King Baudoin Foundation, Flemish Institute for Science, and Technology Assessment (viWTA), 2005.

[Fer+01] Xavier Ferré, Natalia Juristo Juzgado, Helmut Windl, and Larry L. Constantine. "Usability Basics for Software Developers". In: *IEEE Software* 18.1 (2001), pp. 22–29.

[FP13] Jean-Christophe Filliâtre and Andrei Paskevich. "Why3 – Where Programs Meet Provers". In: *ESOP'13 22nd European Symposium on Programming.* Vol. 7792. LNCS. Rome, Italy: Springer, 2013. URL: `https://hal.inria.fr/hal-00789533`.

[Ful+15] Nathan Fulton, Stefan Mitsch, Jan-David Quesel, Marcus Völp, and André Platzer. “KeYmaera X: An Axiomatic Tactical Theorem Prover for Hybrid Systems”. In: *CADE*. Ed. by Amy P. Felty and Aart Middeldorp. Vol. 9195. LNCS. Springer, 2015, pp. 527–538. DOI: 10.1007/978-3-319-21401-6_36.

[Ful+17] Nathan Fulton, Stefan Mitsch, Brandon Bohrer, and André Platzer. “Bellerophon: Tactical Theorem Proving for Hybrid Systems”. In: *Interactive Theorem Proving - 8th International Conference, ITP 2017, Brasília, Brazil, September 26-29, 2017, Proceedings*. Ed. by Mauricio Ayala-Rincón and César A. Muñoz. Vol. 10499. Lecture Notes in Computer Science. Springer, 2017, pp. 207–224. ISBN: 978-3-319-66106-3. DOI: 10.1007/978-3-319-66107-0_14. URL: https://doi.org/10.1007/978-3-319-66107-0%5C_14.

[GLW18] Sarah Grebing, An Thuy Tien Luong, and Alexander Weigl. “Adding Text-Based Interaction to a Direct-Manipulation Interface for Program Verification – Lessons Learned”. In: *13th International Workshop on User Interfaces for Theorem Provers (UITP 2018)*. Ed. by Mateja Jamnik and Christoph Lüth. To appear. 2018.

[GM93] M. J. C. Gordon and T. F. Melham, eds. *Introduction to HOL: a theorem proving environment for higher order logic*. Cambridge University Press, 1993. ISBN: 0-521-44189-7.

[Gog99] Joseph Goguen. “Social and Semiotic Analyses for Theorem Prover User Interface Design”. In: *Formal Aspects of Computing* 11 (1999), pp. 11–272.

[Gou+15] Stijn de Gouw, Jurriaan Rot, Frank S. de Boer, Richard Bubel, and Reiner Hähnle. “OpenJDK’s Java.utils.Collection.sort() Is Broken: The Good, the Bad and the Worst Case”. In: *Computer Aided Verification - 27th International Conference, CAV 2015, San Francisco, CA, USA, July 18-24, 2015, Proceedings, Part I*. 2015, pp. 273–289. DOI: 10.1007/978-3-319-21690-4_16.

[Gou+17] Stijn de Gouw, Frank S de Boer, Richard Bubel, Reiner Hähnle, Jurriaan Rot, and Dominic Steinhöfel. “Verifying OpenJDK’s Sort Method for Generic Collections”. In: *Journal of Automated Reasoning* (Jan. 1, 2017). ISSN: 1573-0670. DOI: 10.1007/s10817-017-9426-4. published.

[GU16] Daniel Grahl and Mattias Ulbrich. “From Specification to Proof Obligations”. In: *Deductive Software Verification – The KeY Book: From Theory to Practice*. Ed. by Wolfgang Ahrendt, Bernhard Beckert, Richard Bubel, Reiner Hähnle, Peter H. Schmitt, and Mattias Ulbrich. Cham: Springer International Publishing, 2016, pp. 243–287. ISBN: 978-3-319-49812-6. DOI: 10.1007/978-3-319-49812-6_8.

[Han+05] Dominik Haneberg, Simon Bäumler, Michael Balser, Holger Grandy, Frank Ortmeier, Wolfgang Reif, Gerhard Schellhorn, Jonathan Schmitt, and Kurt Stenzel. “The user interface of the KIV verification system – a system description”. In: *Proceedings of the User Interfaces for Theorem Provers Workshop (UITP 2005).* 2005.

[Har96] John Harrison. “HOL Light: A tutorial introduction”. In: *Formal Methods in Computer-Aided Design.* Ed. by Mandayam Srivas and Albert Camilleri. Berlin, Heidelberg: Springer Berlin Heidelberg, 1996, pp. 265–269. ISBN: 978-3-540-49567-3.

[HBH18] Martin Hentschel, Richard Bubel, and Reiner Hähnle. “The Symbolic Execution Debugger (SED): a platform for interactive symbolic execution, debugging, verification and more”. In: *International Journal on Software Tools for Technology Transfer* (Mar. 3, 2018). ISSN: 1433-2787. DOI: `10.1007/s10009-018-0490-9`. published.

[Hen16] Martin Hentschel. “Integrating Symbolic Execution, Debugging and Verification”. English. PhD thesis. Technische Universität Darmstadt, 2016.

[Hew+92] Thomas T. Hewett, Ronald Baecker, Stuart Card, Tom Carey, Jean Gasen, Marilyn Mantei, Gary Perlman, Gary Strong, and William Verplank. *ACM SIGCHI Curricula for Human-Computer Interaction.* Tech. rep. New York, NY, USA, 1992.

[HHB16] Martin Hentschel, Reiner Hähnle, and Richard Bubel. “An empirical evaluation of two user interfaces of an interactive program verifier”. In: *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering, ASE 2016, Singapore, September 3-7, 2016.* Ed. by David Lo, Sven Apel, and Sarfraz Khurshid. ACM, 2016, pp. 403–413. ISBN: 978-1-4503-3845-5. DOI: `10.1145/2970276.2970303`.

[Hui01] M. Huisman. “Reasoning about Java Programs in Higher Order Logic with PVS and Isabelle”. PhD thesis. University of Nijmegen, 2001.

[Hup14] Lars Hupel. “Interactive Simplifier Tracing and Debugging in Isabelle”. In: *Intelligent Computer Mathematics – International Conference, CICM 2014, Coimbra, Portugal, July 7-11, 2014. Proceedings.* Ed. by Stephen M. Watt, James H. Davenport, Alan P. Sexton, Petr Sojka, and Josef Urban. Vol. 8543. Lecture Notes in Computer Science. Springer, 2014, pp. 328–343. ISBN: 978-3-319-08433-6. DOI: `10.1007/978-3-319-08434-3_24`.

[ISO10] ISO. *ISO 9241-210:2010 – Ergonomics of human-system interaction – Part 210: Human-centred design for interactive systems.* Tech. rep. International Organization for Standardization, 2010.

[ISO18] ISO. *ISO 9241-11:2018 Ergonomics of human-system interaction – Part 11: Usability: Definitions and concepts.* Tech. rep. International Organization for Standardization, 2018.

[JIR99] Michael Jackson, Andrew Ireland, and G. Reid. “Interactive Proof Critics”. In: *Formal Aspects of Computing* 11.3 (1999), pp. 302–325.

[Kir] J. Kirakowski. *The Use of Questionnaire Methods for Usability Assessment.* URL: `http://sumi.uxp.ie/about/sumipapp.html`.

[Kir+15] Florent Kirchner, Nikolai Kosmatov, Virgile Prevosto, Julien Signoles, and Boris Yakobowski. "Frama-C: A software analysis perspective". In: *Formal Aspects of Computing* 27.3 (May 2015), pp. 573–609. ISSN: 1433-299X. DOI: `10.1007/s00165-014-0326-7`.

[Kle+09a] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. "seL4: Formal Verification of an OS Kernel". In: *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles.* SOSP '09. Big Sky, Montana, USA: ACM, 2009, pp. 207–220. ISBN: 978-1-60558-752-3. DOI: `10.1145/1629575.1629596`. URL: `http://doi.acm.org/10.1145/1629575.1629596`.

[Kle+09b] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. "seL4: Formal Verification of an OS Kernel". In: *ACM Symposium on Operating Systems Principles.* Big Sky, MT, USA: ACM, 2009, pp. 207–220.

[Kle+11] Vladimir Klebanov, Peter Müller, Natarajan Shankar, Gary T. Leavens, Valentin Wüstholz, Eyad Alkassar, Rob Arthan, Derek Bronish, Rod Chapman, Ernie Cohen, Mark Hillebrand, Bart Jacobs, K. Rustan M. Leino, Rosemary Monahan, Frank Piessens, Nadia Polikarpova, Tom Ridge, Jan Smans, Stephan Tobies, Thomas Tuerk, Mattias Ulbrich, and Benjamin Weiß. "The 1st Verified Software Competition: Experience Report". In: *Proceedings, 17th International Symposium on Formal Methods (FM).* Ed. by Michael Butler and Wolfram Schulte. Vol. 6664. LNCS. Materials available at `www.vscomp.org`. Springer, 2011.

[Kle+14] Gerwin Klein, June Andronick, Kevin Elphinstone, Toby Murray, Thomas Sewell, Rafal Kolanski, and Gernot Heiser. "Comprehensive Formal Verification of an OS Microkernel". In: *ACM Transactions on Computer Systems* 32.1 (2014), 2:1–2:70. DOI: `10.1145/2560537`.

[KM05] Bonnie Kaplan and Joseph Maxwell. "Qualitative Research Methods for Evaluating Computer Information Systems". In: Jan. 2005, pp. 30–55. DOI: `10.1007/0-387-30329-4_2`.

[KSD96] Gada Kadoda, Roger Stone, and Dan Diaper. "Desirable Features of Educational Theorem Provers: A Cognitive Dimensions Viewpoint". In: *Proceedings of the 11th Annual Workshop of the Psychology of Programming Interest Group.* 1996.

[Kuc14] U. Kuckartz. *Qualitative Inhaltsanalyse. Methoden, Praxis, Computerunterstützung.* Grundlagentexte Methoden. Beltz Juventa, 2014.

[LC05] Gary T. Leavens and Yoonsik Cheon. “Design by Contract with JML”. Draft, available from jmlspecs.org. 2005.

[Lea+13] Gary T. Leavens, Erik Poll, Curtis Clifton, Yoonsik Cheon, Clyde Ruby, David Cok, Peter Müller, Joseph Kiniry, Patrice Chalin, Daniel M. Zimmerman, and Werner Dietl. *JML Reference Manual.* Draft Revision 2344. May 2013.

[Lei10] Rustan Leino. “Dafny: An Automatic Program Verifier For Functional Correctness”. In: Microsoft Research, Apr. 2010. URL: `https://www.microsoft.com/en-us/research/publication/dafny-automatic-program-verifier-functional-correctness/`.

[LHS08] Bettina Laugwitz, Theo Held, and Martin Schrepp. “Construction and evaluation of a user experience questionnaire”. In: *Symposium of the Austrian HCI and Usability Engineering Group.* Springer. 2008, pp. 63–76.

[LLG16] Yuhui Lin, Pierre Le Bras, and Gudmund Grov. “Developing and Debugging Proof Strategies by Tinkering”. In: *Proceedings of the 22nd International Conference on Tools and Algorithms for the Construction and Analysis of Systems - Volume 9636.* New York, NY, USA: Springer-Verlag New York, Inc., 2016, pp. 573–579. ISBN: 978-3-662-49673-2. DOI: `10.1007/978-3-662-49674-9_37`.

[Low+96] Helen Lowe, Andrew Cumming, Michael Smyth, and Alison Varey. “Lessons From Experience: Making Theorem Provers More Co-operative”. In: *Proceedings 2nd Workshop User Interfaces for Theorem Provers.* 1996.

[Luo18] An Thuy Tien Luong. *Evaluation der Proof-Script-Sprache für KeY (in German).* Bachelor’s Thesis at Karlsruhe Institue of Technology. Mar. 2018.

[LW14] K. Rustan M. Leino and Valentin Wüstholz. “The Dafny Integrated Development Environment”. In: *Proceedings 1st Workshop on Formal Integrated Development Environment, F-IDE 2014, Grenoble, France, April 6, 2014.* Ed. by Catherine Dubois, Dimitra Giannakopoulou, and Dominique Méry. Vol. 149. EPTCS. 2014, pp. 3–15. DOI: `10.4204/EPTCS.149.2`.

[Mac13] I. Scott MacKenzie. *Human-Computer Interaction: An Empirical Research Perspective.* Elsevier, 2013.

[May00] Philipp Mayring. “Qualitative Content Analysis”. In: *Forum : Qualitative Social Research* 1.2 (June 2000). Online Journal, 1(2). Available at: `http://qualitative-research.net/fqs/fqs-e/2-00inhalt-e.htm` [Date of access: 04, 2014].

[May96] Philipp Mayring. *Einführung in die qualitative Sozialforschung – Eine Anleitung zu qualitativem Denken (Introduction to qualitative social research).* Weinheim: Psychologie Verlags Union, 1996.

[MB08] Leonardo de Moura and Nikolaj Bjørner. “Z3: An Efficient SMT Solver”. In: *Tools and Algorithms for the Construction and Analysis of Systems.* Ed. by C. R. Ramakrishnan and Jakob Rehof. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 337–340. ISBN: 978-3-540-78800-3.

[Mel94] Erica Melis. “How Mathematicians Prove Theorems”. In: *In Proceedings of the Annual Conference of the Cognitive Science Society.* Lawrence Erlbaum Associates, Publisher, 1994, pp. 624–628.

[Mer96] Nicholas A. Merriam. “Two Modelling Approaches Applied to User Interfaces to Theorem Proving Assistants”. In: *Proceedings of the 2nd International Workshop on User Interface Design for Theorem Proving Systems.* Department of Computer Science, University of York. 1996, pp. 75–82.

[Mey92] Bertrand Meyer. “Applying “Design by Contract””. In: *IEEE Computer* 25.10 (Oct. 1992), pp. 40–51.

[MH96] NicholasA. Merriam and MichaelD. Harrison. “Evaluating the Interfaces of Three Theorem Proving Assistants”. English. In: *Design, Specification and Verification of Interactive Systems ’96.* Ed. by Francois Bodart and Jean Vanderdonckt. Eurographics. Springer Vienna, 1996, pp. 330–346. ISBN: 978-3-211-82900-4. DOI: `10.1007/978-3-7091-7491-3_17`. URL: `http://dx.doi.org/10.1007/978-3-7091-7491-3_17`.

[MH98] Nicholas A. Merriam and Michael D. Harrison. “Making Design Decisions to Support Diversity in Interactive Theorem Proving”. In: *User Interfaces* 98 (1998), p. 112.

[MMW16] Daniel Matichuk, Toby Murray, and Makarius Wenzel. “Eisbach: A Proof Method Language for Isabelle”. In: *Journal of Automated Reasoning* 56.3 (Mar. 2016), pp. 261–282. ISSN: 1573-0670. DOI: `10.1007/s10817-015-9360-2`.

[MN90] Rolf Molich and Jakob Nielsen. “Improving a Human-computer Dialogue”. In: *Commun. ACM* 33.3 (Mar. 1990), pp. 338–348. ISSN: 0001-0782. DOI: `10.1145/77481.77486`.

[MP16] Stefan Mitsch and André Platzer. “The KeYmaera X Proof IDE - Concepts on Usability in Hybrid Systems Theorem Proving”. In: *Proceedings of the Third Workshop on Formal Integrated Development Environment, F-IDE@FM 2016, Limassol, Cyprus, November 8, 2016.* Ed. by Catherine Dubois, Paolo Masci, and Dominique Méry. Vol. 240. EPTCS. 2016, pp. 67–81. DOI: `10.4204/EPTCS.240.5`. URL: `https://doi.org/10.4204/EPTCS.240.5`.

[MP17] Stefan Mitsch and André Platzer. “The KeYmaera X Proof IDE - Concepts on Usability in Hybrid Systems Theorem Proving”. In: *Proceedings of the Third Workshop on Formal Integrated Development Environment, Limassol, Cyprus, November 8, 2016.* Ed. by Catherine Dubois, Paolo Masci, and Dominique Méry. Vol. 240. Electronic Proceedings in Theoretical Computer Science. Open Publishing Association, 2017, pp. 67–81. DOI: 10.4204/EPTCS.240.5.

[Nie93] Jakob Nielsen. *Usability engineering*. Academic Press, 1993. ISBN: 978-0-12-518405-2.

[Nie94] Jakob Nielsen. “Enhancing the Explanatory Power of Usability Heuristics”. In: *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. CHI ’94. Boston, Massachusetts, USA: ACM, 1994, pp. 152–158. ISBN: 0-89791-650-6. DOI: 10.1145/191666.191729.

[Nie95] Jacob Nielsen. *10 Usability Heuristics for User Interface Design*. 1995. URL: nngroup.com/articles/ten-usability-heuristics (visited on 12/01/2018).

[Nor13] D. Norman. *The Design of Everyday Things: Revised and Expanded Edition*. Basic Books, 2013. ISBN: 9780465072996.

[NPW02] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*. Vol. 2283. LNCS. Springer, 2002.

[OL07] Martin Ouimet and Kristina Lundqvist. *Formal Software Verification: Model Checking and Theorem Proving*. Tech. rep. Mar. 2007. URL: http://www.es.mdh.se/publications/1215-.

[ORS92] Sam Owre, John M. Rushby, and Natarajan Shankar. “PVS: A Prototype Verification System”. In: *Proceedings of the 11th International Conference on Automated Deduction: Automated Deduction*. CADE-11. London, UK, UK: Springer-Verlag, 1992, pp. 748–752. ISBN: 3-540-55602-8. URL: http://dl.acm.org/citation.cfm?id=648230.752639.

[OSF16] Steven Obua, Phil Scott, and Jacques Fleuriot. “ProofScript: Proof Scripting for the Masses”. In: *Theoretical Aspects of Computing – ICTAC 2016: 13th International Colloquium, Taipei, Taiwan, ROC, October 24–31, 2016, Proceedings*. Ed. by Augusto Sampaio and Farn Wang. Cham: Springer International Publishing, 2016, pp. 333–348. ISBN: 978-3-319-46750-4. DOI: 10.1007/978-3-319-46750-4_19.

[Pau10] Lawrence C. Paulson. “Three Years of Experience with Sledgehammer, a Practical Link between Automatic and Interactive Theorem Provers”. In: *Proceedings of the 2nd Workshop on Practical Aspects of Automated Reasoning, PAAR-2010, Edinburgh, Scotland, UK, July 14, 2010*. Ed. by Renate A. Schmidt, Stephan Schulz, and Boris Konev. Vol. 9. EPiC Series in Computing. EasyChair, 2010, pp. 1–10. URL: http://www.easychair.org/publications/paper/52675.

[Pau12] C. Paulin-Mohring. “Tools for Practical Software Verification, LASER 2011 summerschool, Revised Tutorial Lectures”. In: ed. by B. Meyer and M. Nordio. Lecture Notes in Computer Science 7682. Springer-Verlag, 2012. Chap. Introduction to the Coq proof-assistant for practical software verification, pp. 45–95.

[Pfe17] Wolfram Pfeifer. *Specifying and Verifying Real-World Java Code with KeY – Case Study java.math.BigInteger.* Bachelor's Thesis at Karlsruhe Institue of Technology. 2017.

[Pla10] André Platzer. *Logical Analysis of Hybrid Systems: Proving Theorems for Complex Dynamics.* Heidelberg: Springer, 2010. ISBN: 978-3-642-14508-7. DOI: `10.1007/978-3-642-14509-4`. URL: `http://www.springer.com/978-3-642-14508-7`.

[Pla12] André Platzer. "The Complete Proof Theory of Hybrid Systems". In: *Proceedings of the 27th Annual IEEE Symposium on Logic in Computer Science, LICS 2012, Dubrovnik, Croatia, June 25-28, 2012.* IEEE Computer Society, 2012, pp. 541–550. ISBN: 978-1-4673-2263-8. DOI: `10.1109/LICS.2012.64`. URL: `https://doi.org/10.1109/LICS.2012.64`.

[Pla17] André Platzer. "A Complete Uniform Substitution Calculus for Differential Dynamic Logic". In: *J. Autom. Reasoning* 59.2 (2017), pp. 219–265. DOI: `10.1007/s10817-016-9385-1`. URL: `https://doi.org/10.1007/s10817-016-9385-1`.

[Pla18] André Platzer. *Logical Foundations of Cyber-Physical Systems.* Switzerland: Springer, 2018. ISBN: 978-3-319-63587-3. URL: `http://www.springer.com/978-3-319-63587-3`.

[Pre+94] Jenny Preece, Yvonne Rogers, Helen Sharp, David Benyon, Simon Holland, and Tom Carey. *Human Computer Interaction.* Wokingham, England: Addison-Wesley, 1994.

[RU16] Philipp Rümmer and Mattias Ulbrich. "Proof Search with Taclets". In: *Deductive Software Verification – The KeY Book: From Theory to Practice.* Ed. by Wolfgang Ahrendt, Bernhard Beckert, Richard Bubel, Reiner Hähnle, Peter H. Schmitt, and Mattias Ulbrich. Cham: Springer International Publishing, 2016, pp. 107–147. ISBN: 978-3-319-49812-6. DOI: `10.1007/978-3-319-49812-6_4`.

[Sch16] Peter H. Schmitt. "First-Order Logic". In: *Deductive Software Verification – The KeY Book: From Theory to Practice.* Ed. by Wolfgang Ahrendt, Bernhard Beckert, Richard Bubel, Reiner Hähnle, Peter H. Schmitt, and Mattias Ulbrich. Cham: Springer International Publishing, 2016, pp. 23–47. ISBN: 978-3-319-49812-6. DOI: `10.1007/978-3-319-49812-6_2`.

[Sch83] Ben Schneiderman. "Direct Manipulation. A Step Beyond Programming Languages". In: *IEEE Transactions on Computers* 16.8 (Aug. 1983), pp. 57–69.

[Sch92] Alan H. Schoenfeld. "Learning to think Mathematically: Problem solving, metacognition, and sense making in mathematics". In: *Handbook of research on mathematics teaching and learning.* Ed. by Douglas A. Grouws. 1992, pp. 334–370.

[Tog14] Bruce Tognazzini. *First Principles of Interaction Design (Revised and Expanded).* 1987-2014. URL: `https://asktog.com/atc/principles-of-interaction-design/`.

[VE06] V. Vujosevic and G. Eleftherakis. "Improving Formal Methods' Tools Usability". In: *2nd South-East European Workshop on Formal Methods (SEEFM 05), Formal Methods: Challenges in the Business World, Ohrid, 18-19 Nov 2005.* Ed. by G. Eleftherakis. South-East European Research Centre (SEERC), 2006. ISBN: 960-87869-8-3.

[VH03] Petra Vogt and Sven Heinsen. *Usability praktisch umsetzen: Handbuch für Software, Web, Mobile Devices und andere interaktive Produkte.* Hanser, 2003.

[Völ03] Norbert Völker. "Thoughts on Requirements and Design Issues of User Interfaces for Proof Assistants." In: *User Interfaces for Theorem Provers (UITP 2003).* 2003.

[Wen12] Makarius Wenzel. "Isabelle/jEdit — a Prover IDE within the PIDE framework". In: *CoRR* abs/1207.3441 (2012). arXiv: `1207.3441`. URL: `http://arxiv.org/abs/1207.3441`.

[Wen18] Makarius Wenzel. "Isabelle/jEdit as IDE for Domain-specific Formal Languages and Informal Text Documents". In: *Proceedings 4th Workshop on Formal Integrated Development Environment, F-IDE@FLoC 2018, Oxford, England, 14 July 2018.* Ed. by Paolo Masci, Rosemary Monahan, and Virgile Prevosto. Vol. 284. EPTCS. 2018, pp. 71–84. DOI: `10.4204/EPTCS.284.6`. URL: `https://doi.org/10.4204/EPTCS.284.6`.

[Wen99] Markus Wenzel. "Isar - A Generic Interpretative Approach to Readable Formal Proof Documents". In: *Proceedings of the 12th International Conference on Theorem Proving in Higher Order Logics.* TPHOLs '99. London, UK, UK: Springer-Verlag, 1999, pp. 167–184. ISBN: 3-540-66463-7.

[Wer23] Max Wertheimer. "Untersuchungen zur Lehre von der Gestalt". In: *Psychologische Forschung: Zeitschrift für Psychologie und ihre Grenzwissenschaften* 4 (1923), pp. 301–350. URL: `http://vlp.mpiwg-berlin.mpg.de/library/data/lit38308/index_html?pn=8&ws=1.5`.

[WPN08] Makarius Wenzel, Lawrence C. Paulson, and Tobias Nipkow. "The Isabelle Framework". In: *Theorem Proving in Higher Order Logics (TPHOLs 2008).* Ed. by Ait Mohamed, Munoz, and Tahar. Vol. 5170. LNCS. Springer, 2008, pp. 33–38.

[Zil+13] Beta Ziliani, Derek Dreyer, Neelakantan R. Krishnaswami, Aleksandar Nanevski, and Viktor Vafeiadis. "Mtac: A Monad for Typed Tactic Programming in Coq". In: *SIGPLAN Not.* 48.9 (2013), pp. 87–100. ISSN: 0362-1340. DOI: `10.1145/2544174.2500579`.

Part VI.

Appendix

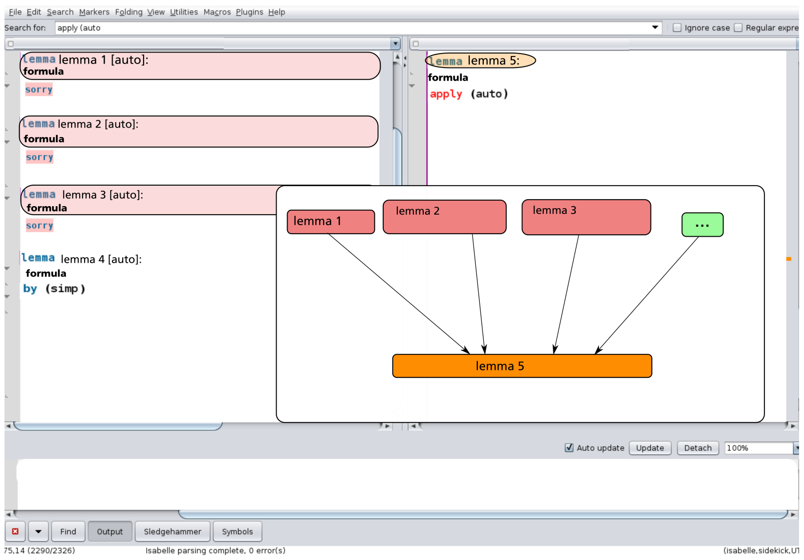
A. Appendix: Focus Groups

Documents related with the focus group discussion, such as the focus group scripts, can be found on the webpage `http://formal.iti.kit.edu/~grebing/SWC`. Examples for the visual cues in the discussions are presented in the following.

A.1. Examples for Visual Cues in the Focus Group Discussions

In the following, we present two examples for the visual cues shown during the focus group discussion for potential mechanisms. The full sequence of images can be found on the webpage `http://formal.iti.kit.edu/~grebing/SWC`.

A.1.1. Isabelle



Search for: apply (auto

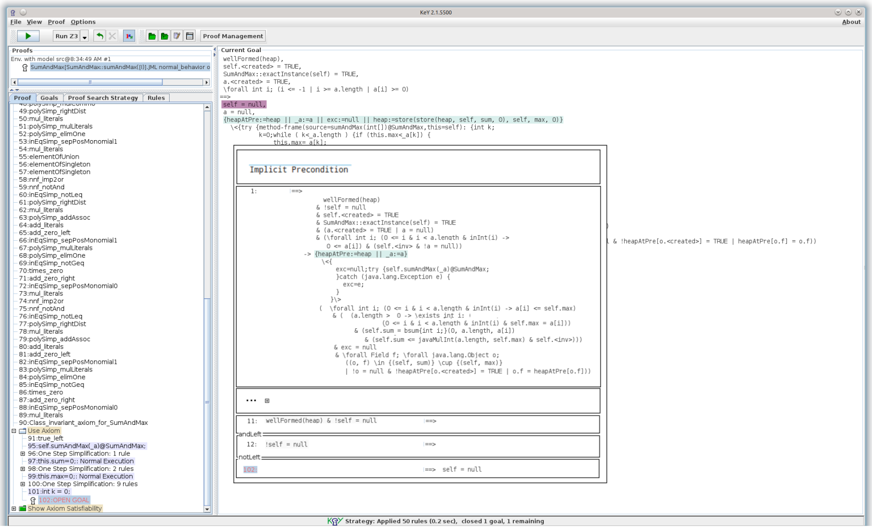
```
lemma lemma 1 [auto]:
formula
sorry

lemma lemma 2 [auto]:
formula
sorry

lemma lemma 3 [auto]:
formula
sorry

lemma lemma 4 [auto]:
formula
by (simp)
```

```
lemma lemma 5:
formula
apply (auto)
```

Isabelle parsing complete, 0 error(s)

Figure A.1.: A visual cue showing an idea for proof management in Isabelle. Users may select a lemma and retrieve a new window containing information about the lemmas used in the proof for the selected lemma. Furthermore, the proof status of lemmas is color-coded in this view.

A.1.2. KeY System



```
Env. with model src@8:34:49 AM #1
  SumAndMax[SumAndMax::sumAndMax([I)].JML normal_behavior o

49:polySimp_rightDist
50:mul_literals
51:polySimp_mulLiterals
52:polySimp_elimOne
53:inEqSimp_sepPosMonomial1
54:mul_literals
55:elementOfUnion
56:elementOfSingleton
57:elementOfSingleton
58:nnf_imp2or
59:nnf_notAnd
60:inEqSimp_notLeq
61:polySimp_rightDist
62:mul_literals
63:polySimp_addAssoc
64:add_literals
65:add_zero_left
66:inEqSimp_sepPosMonomial1
67:polySimp_mulLiterals
68:polySimp_elimOne
69:inEqSimp_notGeq
70:times_zero
71:add_zero_right
72:inEqSimp_sepPosMonomial0
73:mul_literals
74:nnf_imp2or
75:nnf_notAnd
76:inEqSimp_notLeq
77:polySimp_rightDist
78:mul_literals
79:polySimp_addAssoc
80:add_literals
81:add_zero_left
82:inEqSimp_sepPosMonomial1
83:polySimp_mulLiterals
84:polySimp_elimOne
85:inEqSimp_notGeq
86:times_zero
87:add_zero_right
88:inEqSimp_sepPosMonomial0
89:mul_literals
90:Class_invariant_axiom_for_SumAndMax
Use Axiom
  91:true_left
  95:self.sumAndMax(_a)@SumAndMax;
  96:One Step Simplification: 1 rule
  97:this.sum=0;: Normal Execution
  98:One Step Simplification: 2 rules
  99:this.max=0;: Normal Execution
  100:One Step Simplification: 9 rules
  101:int k = 0;
  102:OPEN GOAL
Show Axiom Satisfiability
```

```
 wellFormed(heap),
 self.<created> = TRUE,
 SumAndMax::exactInstance(self) = TRUE,
 a.<created> = TRUE,
 \forall int i; (i <= -1 | i >= a.length | a[i] >= 0)
==>
 self = null,
 a = null,
 {heapAtPre:=heap || _a:=a || exc:=null || heap:=store(store(heap, self, sum, 0), self, max, 0)}
   \<{try {method-frame(source=sumAndMax(int[])@SumAndMax,this=self): {int k;
          k=0;while ( k<_a.length ) {if (this.max<_a[k]) {
             this.max= a[k];
                                                     & !heapAtPre[o.<created>] = TRUE | heapAtPre[o.f] = o.f))
```

```
Implicit Precondition

 1:          ==>
                 wellFormed(heap)
               & !self = null
               & self.<created> = TRUE
               & SumAndMax::exactInstance(self) = TRUE
               & (a.<created> = TRUE | a = null)
               & (\forall int i; (0 <= i & i < a.length & inInt(i) ->
                  0 <= a[i]) & (self.<inv> & !a = null))
            -> {heapAtPre:=heap || _a:=a}
                 \<{
                     exc=null;try {self.sumAndMax(_a)@SumAndMax;
                     }catch (java.lang.Exception e) {
                       exc=e;
                     }
                   }\>
               (  \forall int i; (0 <= i & i < a.length & inInt(i) -> a[i] <= self.max)
                  & (  (a.length >  0 -> \exists int i:
                                (0 <= i & i < a.length & inInt(i) & self.max = a[i]))
                         & (self.sum = bsum{int i;}(0, a.length, a[i])
                            & (self.sum <= javaMulInt(a.length, self.max) & self.<inv>)))
                  & exc = null
                  & \forall Field f; \forall java.lang.Object o;
                     ((o, f) \in {(self, sum)} \cup {(self, max)}
                      | !o = null & !heapAtPre[o.<created>] = TRUE | o.f = heapAtPre[o.f]))

 ...

 11:  wellFormed(heap) & !self = null          ==>
andLeft
 12:  !self = null                             ==>
notLeft
 102:                                          ==>  self = null
```

Strategy: Applied 50 rules (0.2 sec), closed 1 goal, 1 remaining

Figure A.2.: One of the visual cues shown in the KeY focus group presenting an idea for showing the origin of formulas in KeY. The information about the origin of the highlighted formula is shown in a separate window.

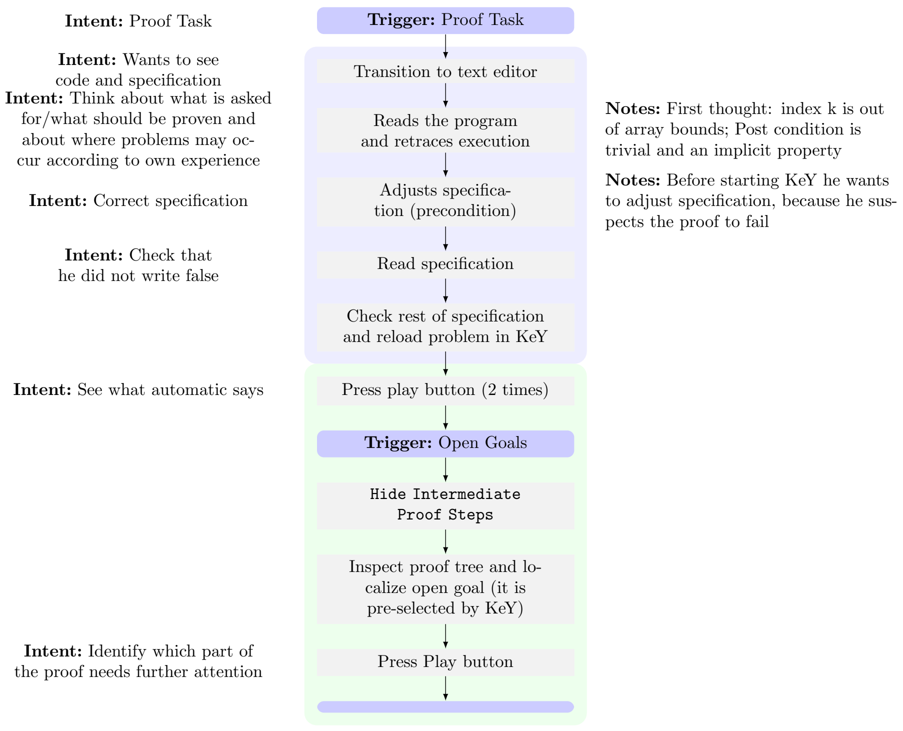
B. Appendix: User Study

The script in German as well as other material from the user study can be found on our webpage `http://formal.iti.kit.edu/~grebing/SWC`. Excerpts form the user study can be found in the following.

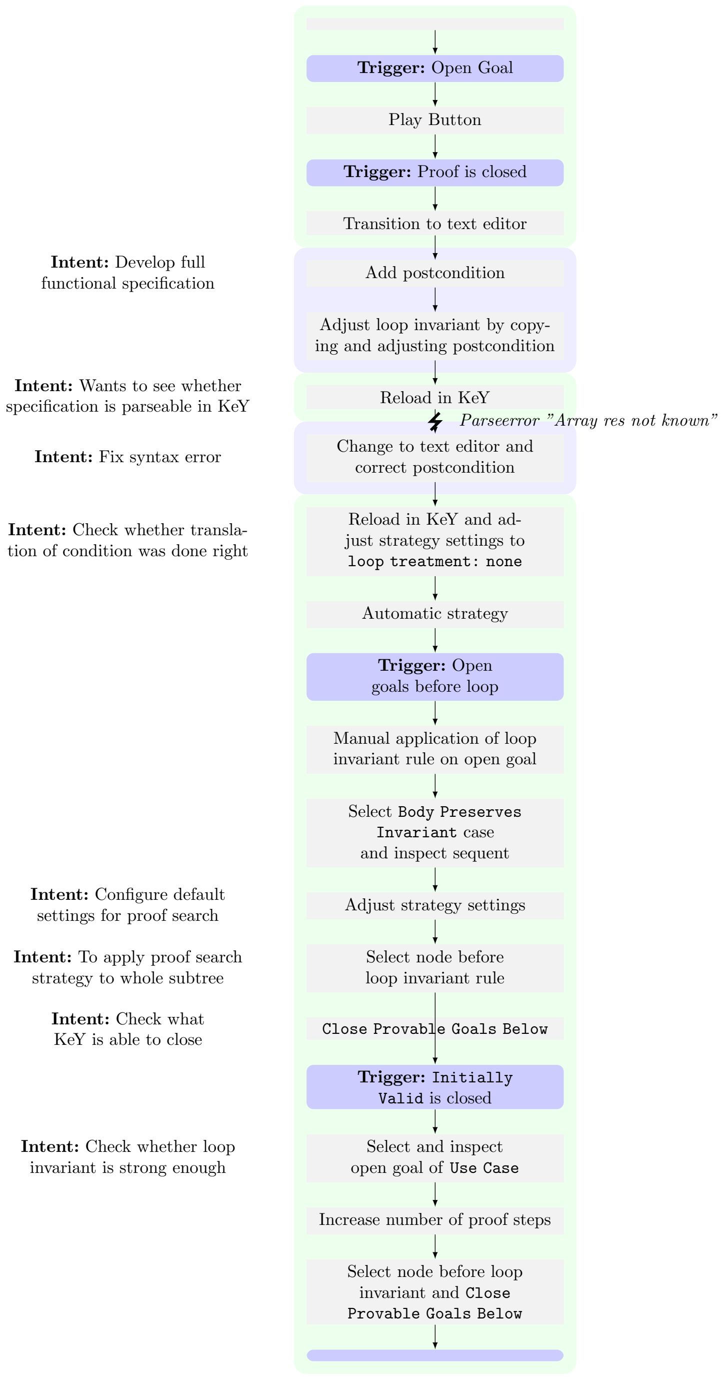
B.1. Proof Process

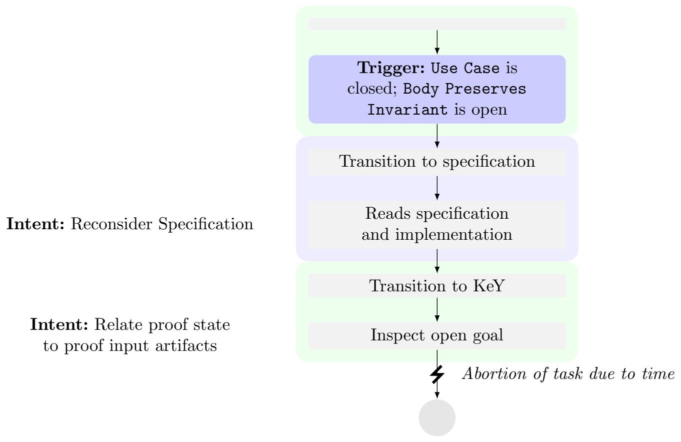
For the analysis of the practical tasks we developed sequence models of the participants' actions we observed during the user study. In the following one example for such a sequence model is shown. It depicts the actions of a participant while approaching the first practical task from the user study. Activities in the text editor are marked with a blue background, activities in KeY are shown with a green background.

B.1.1. Example Sequence Model



Intent: Proof Task
Trigger: Proof Task
Intent: Wants to see code and specification
Transition to text editor
Intent: Think about what is asked for/what should be proven and about where problems may occur according to own experience
Reads the program and retraces execution
Notes: First thought: index k is out of array bounds; Post condition is trivial and an implicit property
Intent: Correct specification
Adjusts specification (precondition)
Notes: Before starting KeY he wants to adjust specification, because he suspects the proof to fail
Intent: Check that he did not write false
Read specification
Check rest of specification and reload problem in KeY
Intent: See what automatic says
Press play button (2 times)
Trigger: Open Goals
Hide Intermediate Proof Steps
Inspect proof tree and localize open goal (it is pre-selected by KeY)
Intent: Identify which part of the proof needs further attention
Press Play button

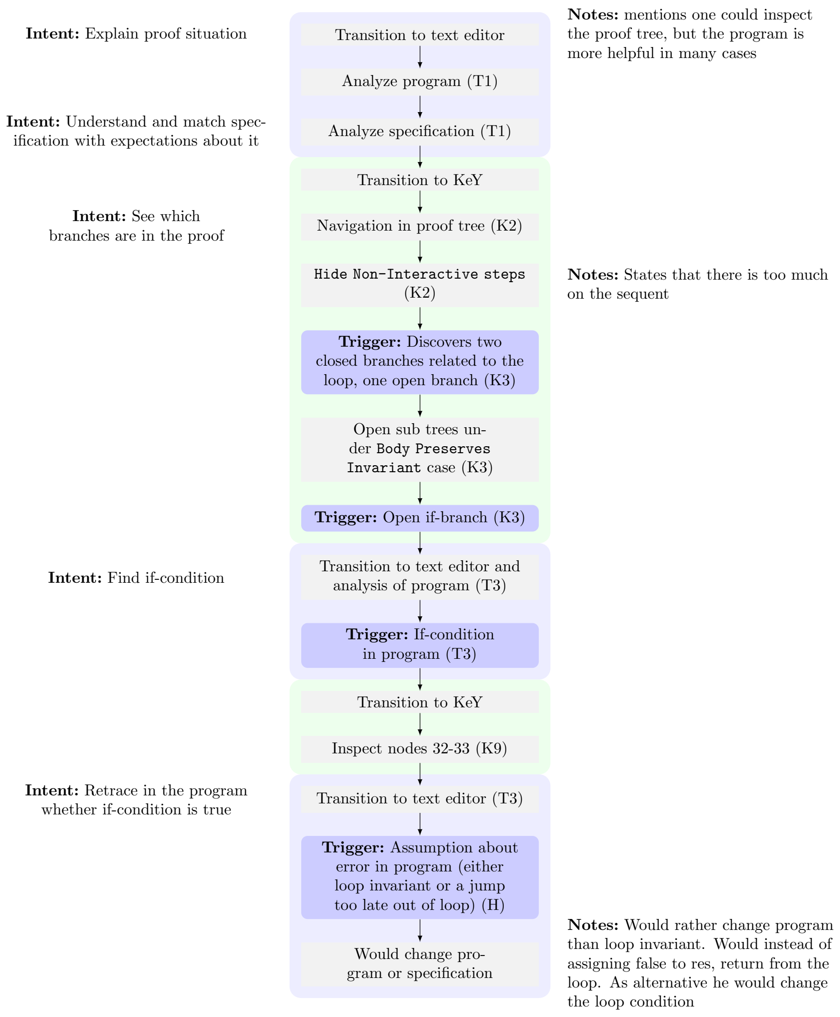

Trigger: Open Goal
Play Button
Trigger: Proof is closed
Transition to text editor
Intent: Develop full functional specification
Add postcondition
Adjust loop invariant by copying and adjusting postcondition
Intent: Wants to see whether specification is parseable in KeY
Reload in KeY
Parseerror "Array res not known"
Intent: Fix syntax error
Change to text editor and correct postcondition
Intent: Check whether translation of condition was done right
Reload in KeY and adjust strategy settings to loop treatment: none
Automatic strategy
Trigger: Open goals before loop
Manual application of loop invariant rule on open goal
Select Body Preserves Invariant case and inspect sequent
Intent: Configure default settings for proof search
Adjust strategy settings
Intent: To apply proof search strategy to whole subtree
Select node before loop invariant rule
Intent: Check what KeY is able to close
Close Provable Goals Below
Trigger: Initially Valid is closed
Intent: Check whether loop invariant is strong enough
Select and inspect open goal of Use Case
Increase number of proof steps
Select node before loop invariant and Close Provable Goals Below


Trigger: `Use Case` is closed; `Body Preserves Invariant` is open
Transition to specification
Reads specification and implementation
Intent: Reconsider Specification
Transition to KeY
Inspect open goal
Intent: Relate proof state to proof input artifacts
Abortion of task due to time

B.2. Orientation in the Proof Process

To gain orientation in the proof process we asked participants to explain a proof situation that was constructed using the automatic strategies of KeY. An example for a sequence model we were able to derive during data analysis is shown in the following.

B.2.1. Example Sequence Model



Intent: Explain proof situation
Transition to text editor
Notes: mentions one could inspect the proof tree, but the program is more helpful in many cases
Analyze program (T1)
Intent: Understand and match specification with expectations about it
Analyze specification (T1)
Transition to KeY
Intent: See which branches are in the proof
Navigation in proof tree (K2)
Hide Non-Interactive steps (K2)
Notes: States that there is too much on the sequent
Trigger: Discovers two closed branches related to the loop, one open branch (K3)
Open sub trees under Body Preserves Invariant case (K3)
Trigger: Open if-branch (K3)
Intent: Find if-condition
Transition to text editor and analysis of program (T3)
Trigger: If-condition in program (T3)
Transition to KeY
Inspect nodes 32-33 (K9)
Intent: Retrace in the program whether if-condition is true
Transition to text editor (T3)
Trigger: Assumption about error in program (either loop invariant or a jump too late out of loop) (H)
Would change program or specification
Notes: Would rather change program than loop invariant. Would instead of assigning false to res, return from the loop. As alternative he would change the loop condition

B.2.2. Aggregated Sequence Model

From the sequence models for the orientation task we derived an aggregated model which is shown in the following. In the aggregated sequence model we have abbreviated the activities in Fig. B.2. In the following the description for the abbreviations is shown.

Abbrv.	Activity
T1	gain insight into the proof problem
T2	review specification
T3	find relation to the current/observed proof situation
T4	localize errors/mistakes/issues in the proof input artifacts
K1	find clues/hints for the next action/step
K2	gain overview
K3	gain (more detailed) information about open goals (from labels, branching nodes)
K4	determine reason for open goal/rule out a reason for an open goal
K5	gain rough estimation about open goal
K6	localize/select (specific) open goals/nodes
K7	relate proof state to proof input artifacts
K8	retrace proof/observe changes by rule applications
K9	analyze specific node/sequent for details
K10	inspect proof search strategy settings
K11	change view settings
K12	perform mental deduction steps
H	form hypothesis

Figure B.1.: Descriptions of the abbreviations for activities in the aggregated sequence model in Fig. B.2

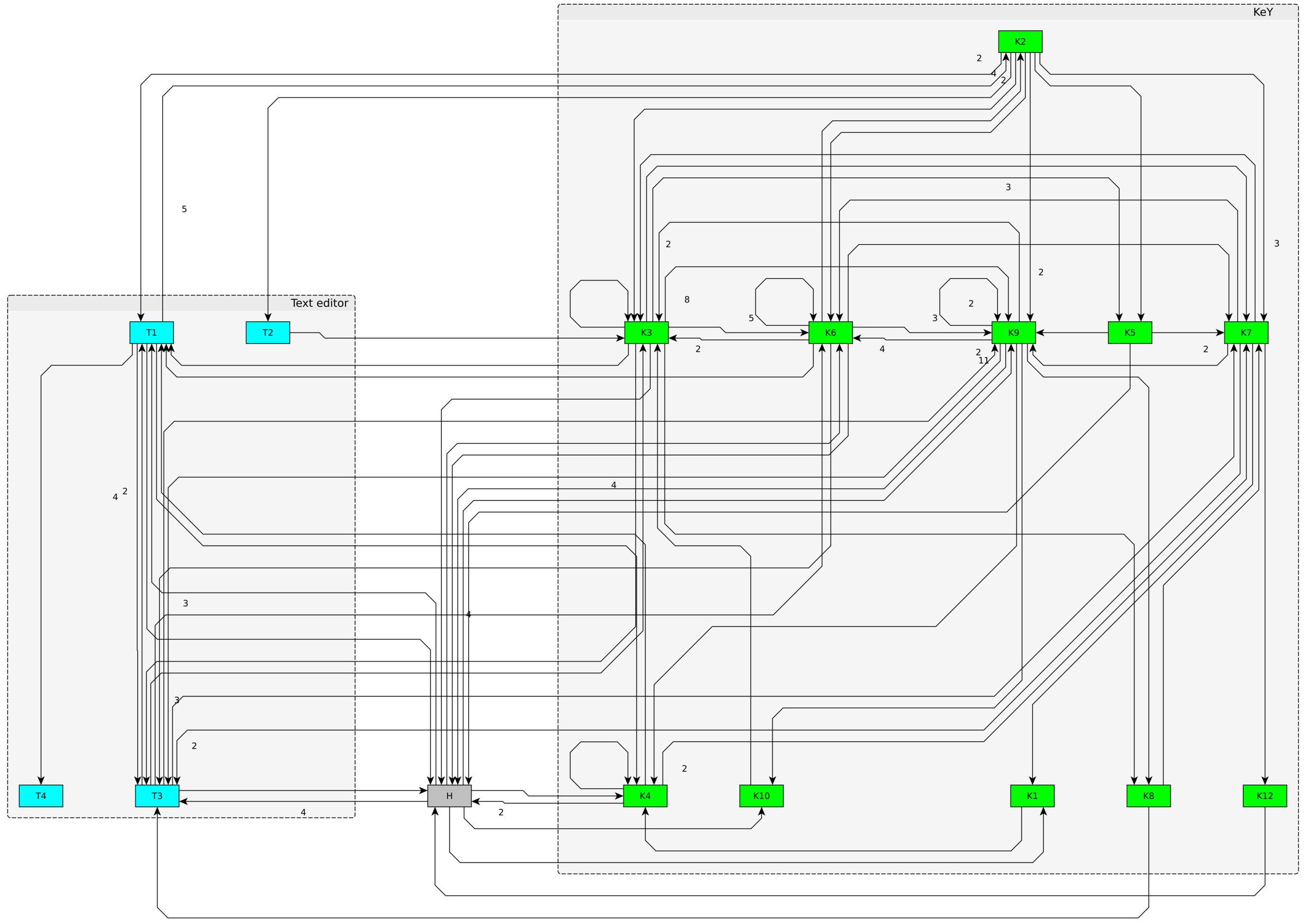
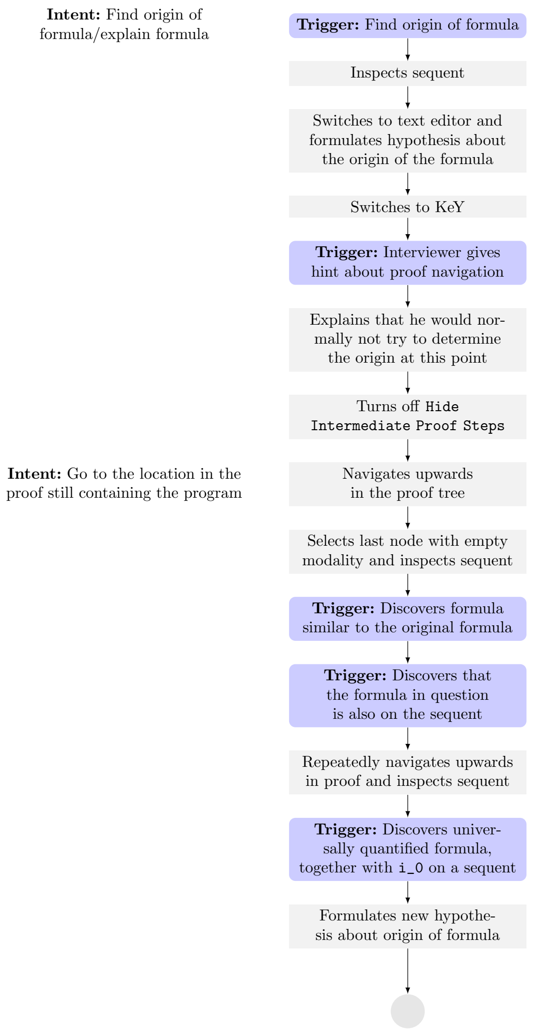


Figure B.2.: Aggregated Sequence Model for the tasks `split()` and `palindrome()`.

B.3. Origin of a Formula

We also observed how the participants search for the origin of a formula. In the following an example of a sequence model we were able to derive is shown.

B.3.1. Example Sequence Model


Intent: Find origin of formula/explain formula
Trigger: Find origin of formula
Inspects sequent
Switches to text editor and formulates hypothesis about the origin of the formula
Switches to KeY
Trigger: Interviewer gives hint about proof navigation
Explains that he would normally not try to determine the origin at this point
Turns off `Hide Intermediate Proof Steps`
Intent: Go to the location in the proof still containing the program
Navigates upwards in the proof tree
Selects last node with empty modality and inspects sequent
Trigger: Discovers formula similar to the original formula
Trigger: Discovers that the formula in question is also on the sequent
Repeatedly navigates upwards in proof and inspects sequent
Trigger: Discovers universally quantified formula, together with `i_0` on a sequent
Formulates new hypothesis about origin of formula