Adding Text-Based Interaction to a Direct Manipulation Interface for Program Verification – Lessons Learned[*]

Sarah Grebing
sarah.grebing@kit.edu

An Thuy Tien Luong
an.luong@student.kit.edu

Alexander Weigl
weigl@kit.edu

Karlsruhe Institute of Technology (KIT)

Interactive program verification is characterized by iterations of unfinished proof attempts. For proof construction, many interactive program verification systems offer either text-based interactions, in using a proof scripting language, or a form of direct manipulation interaction. We have combined a direct manipulation program verification system with a text-based interface to leverage the advantages of both interaction paradigms. To give the user more insight when scripting proofs we have adapted well-known interaction concepts from the field of software debugging. In this paper we report on our experiences in combining the direct manipulation interface of the interactive program verification system KeY with a text-based user interface to construct program verification proofs, leveraging interaction concepts from the field of software-debugging for the proof process.

1 Introduction

Proving complex properties of programs requires user guidance, which can come in the form of program annotations or user interaction during proof construction. Providing the guiding information that allows a verification system to find a proof is, in general, an iterative process of repeated failed attempts. To support the iterative nature of proof construction, text-based as well as direct manipulation interfaces were developed for interactive program verification systems. Direct manipulation (Sec. 3.1) has its strengths, e.g., in the immediate feedback for actions and the visible system state for the user, while text-based interaction (Sec. 3.2) is advantageous when repetition and textual manifestation of actions is needed.

Both interaction paradigms have their advantages and disadvantages (cf. [11]), and demand different requirements onto the user interface. The representation of objects that are being manipulated are central in direct manipulation interfaces and need to be made visible for selection and application of actions. In text-based interfaces the objects themselves are often not shown directly to the user and object manipulation and information retrieval are done via actions which are encoded textually.

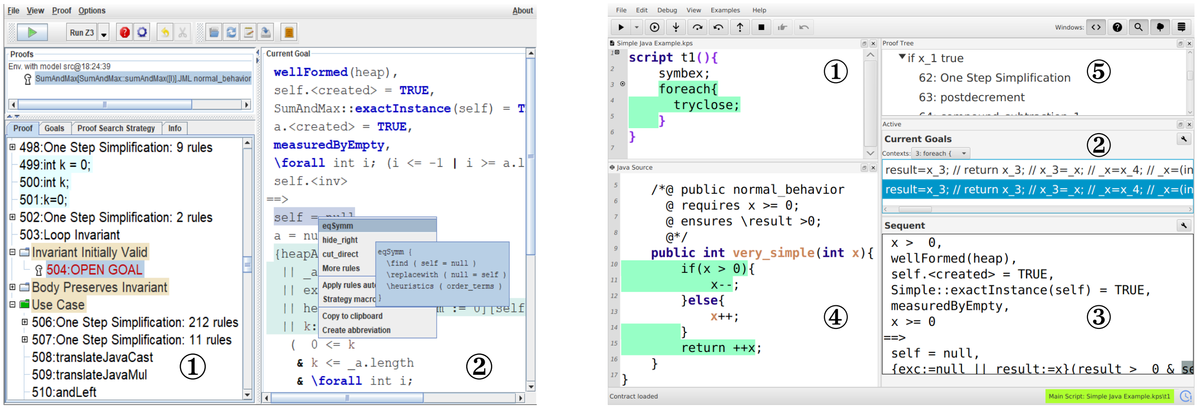
We report on our experiences in developing a text-based interface on top of the already existing direct manipulation interface of the verification system KeY for Java programs together with lessons we have learned from our first evaluation of the implementation. The evaluated implementation is the offline and replay debugger PSDBG, available at `http://formal.iti.kit.edu/psdbg`.

2 Preliminaries: The KeY system

The KeY system [1] is an interactive verification system for Java programs on source code level. It is based on a sequent calculus for Java Dynamic Logic (Java DL). The KeY system was successfully applied to verify real world Java programs, e.g., implementations of Timsort [8] and Dual-Pivot Quicksort [5].

[*] Acknowledgments: We would like to thank Bernhard Beckert and Mattias Ulbrich for fruitful discussions.

Submitted to:
UITP 2018



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 )
}
```

```
script t1(){
    symbex;
    foreach{
      tryclose;
    }
}
```

```
    /*@ public normal_behavior
      @ requires x >= 0;
      @ ensures \result >0;
      @*/
    public int very_simple(int x){
        if(x > 0){
            x--;
        }else{
            x++;
        }
        return ++x;
    }
}
```

```
▾if x_1 true
   62: One Step Simplification
   63: postdecrement
```

```
result=x_3; // return x_3; // x_3=_x; // _x=x_4; // _x=(in
result=x_3; // return x_3; // x_3=_x; // _x=x_4; // _x=(in
```

```
 x >  0,
 wellFormed(heap),
 self.<created> = TRUE,
 Simple::exactInstance(self) = TRUE,
 measuredByEmpty,
 x >= 0
==>
 self = null,
 {exc:=null || result:=x}(result >  0 & se
```

(a) (b)

Figure 1: The user interfaces of KeY (a) and PSDBG (b)

KeY constructs an explicit proof object, i.e., all proof steps and rule applications are available to the user at any time in addition to the current open goals. The user interface contains two different views on the proof state, the proof tree (①), and the node view (②).

The proof tree is a more high-level view onto 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. The node view is a more detailed view and can be obtained by selecting a proof node. In this view the user sees the proof obligation stored in the proof node as a textual representation of the sequent.

In program verification, many large proof states have to be presented to the user, containing both information on the symbolic program state and information on logical rule applications. In KeY these steps are represented in the proof tree, as symbolic execution steps and calculus rule applications.

Proof construction is performed only by using direct manipulation (see Fig. 1a). The user points to a term of the sequent in the node view and selects it by clicking, applicable calculus rules are shown and the user can choose the appropriate rule. If the rule application requires more parameters, an input dialog with drag-and-drop support is presented. Additionally, KeY offers several proof search strategies (also called *macro* steps). Furthermore, 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.

2.1 The Proof Process in KeY

The typical workflow of KeY is depicted in Fig. 3: Initially, the user provides a Java program, together with a specification formulated in the Java Modelling Language [9] (step 1a). The result of an automatic proof search (step 2) is (a) the successful verification of the program or (b) either a counterexample or an open proof with goals that remain to be shown. In the latter case, the user may interact directly with KeY (step 3a) by interactively applying calculus rules (e.g., 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.

2.2 KPS: A Proof Scripting Language For KeY

We developed KPS [3], a proof scripting language for the KeY system. An example script is depicted in Fig. 2. The basic constructs in KPS are *selectors* (cf. line 5), to select proof goals in the proof state, and

```
1  script simpleExample(){
2    impRight;
3    impRight;
4    impLeft;
5    cases{
6        case match `==> !p, ?Z`:
7           notRight; auto;
8        default: auto;
9    }}
```

Figure 2: Example script for KPS

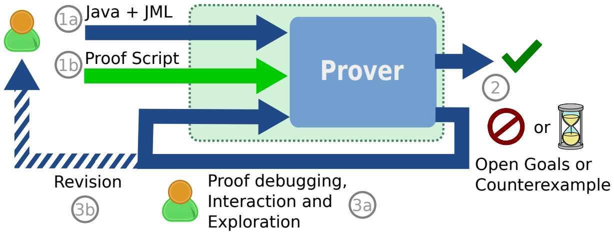


Figure 3: Classical Proof process in KeY extended with scripts

mutators (cf. lines 2 or 3) to apply proof commands to the selected goals. Selectors can contain *match expressions* (cf. line 6), that match on constituents of a proof goal, i.e., on parts of the sequent (cf. line 6), as well as on meta information such as branching labels. For more details on the language concept we refer the reader to [4].

3 Common Interaction Paradigms in Verification System

Like most user interfaces today, verification systems combine different interaction paradigms. However, for proof construction, text-based or direct manipulation is the prominent interaction style.

3.1 Direct Manipulation

In the direct manipulation paradigm 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” [16].

In deductive program verification the task domain consists of the proof object, e.g., as proof tree, and its constituents. Actions to manipulate the objects include proof commands (calculus rules, invocations of automatic strategies) and actions to manipulate the visual representation of the objects.

Direct manipulation interfaces adhere to the usability principle of *recognition rather than recall*: in program verification, rather than remembering all available calculus rules, the user may select the position where an action should be performed and retrieve suggestions from the proof system about applicable rules, i.e., the actions can be context-sensitive. In program verification systems also drag-and-drop actions are possible, e.g., for quantifier instantiation. An additional aid for the user can be tool-tips that show the result of applying the selected rule. This feature supports users in making a promising decision for the progress of the proof process and should prevent them from unrecoverable errors, which follows Nielsen’s usability heuristic of *error prevention* [13].

Another advantage is that the user sees the immediate result of the selected action, which also contributes to a visible system status. In program verification this advantage is leveraged as follows. After the application of calculus rules the proof is updated, e.g., by adding proof nodes to the proof tree and by adjusting the node view to show the newly created proof nodes.

An advantage of the reversibility of actions is the *easy recoverability from errors*. Together with lightweight rule applications this feature enables proof exploration.

The downside of direct manipulation is the amount and types of objects of the task domain that need to be represented: a large amount of objects require a lot of space on the screen, the same holds for large objects that can not be depicted in a dense form [16]. This property gives rise to issues such as that the user is overwhelmed with information and has difficulties finding the right object to manipulate. This issue is especially prominent in program verification proofs with their large proof states, mainly resulting from the encoding of the program states. The size of the proof tree, representing all applied proof steps so far, increases with a progressing proof. For user support projections of the proof tree are available (e.g., hiding intermediate proof nodes). The proof tree and its possibilities to change the view to focus on specific steps also counteracts the general disadvantage of the direct manipulation paradigm where the history of actions is not easily accessible. The proof process using a verification system is characterized by alternating automatic and interactive proof steps – applying automatic steps may result in a large new subtree that is added to the proof tree – counteracting this advantage. Finding the orientation and the proof node on which the proof command was applied to can be challenging for the user.

Another general disadvantage of direct manipulation is a missing shortcut for repeatedly applying actions to several objects in the task domain. In program verification proofs, this issue is especially prominent, as proof goals are often similar and require the same proof steps for a successful proof.

3.2 Text-Based Interaction

The *command-line interaction* (CLI) forms the basic principle for classical text-based interaction, where users formulate actions as textual commands. The *script-based interaction* is an extension to the CLI interaction where the user can use control flow structures to combine commands to more complex actions. In the case of interactive verification, actions can be either view manipulating or proof state manipulating. Visual feedback is only presented after fully executing the script. Instead of repeated manual user interactions, an interpreter decides, based on the proof and the script state, which actions to take.

Efficient interaction with the system by experienced users and the support for repetitions of actions are advantages of this interaction style. Expressing proof plans in advance and capturing the history of proof commands during proof construction is possible. The script is the representation of the proof.

The downside of script-based interaction is the violation of the principle *recognition rather than recall* for applying proof commands. The syntax of a script language has to be learned before being able to use the system. This disadvantage becomes apparent when considering the huge amount of available proof commands, from which not all are applicable in all proof situations. Features like auto-completion and visual highlights for syntactic errors try to mitigate this disadvantage.

The user also needs support in identifying the correct syntax for referring to the part of the object that is manipulated. The textual representation of terms needs to be unambiguous and comprehensible, which is an issue in program verification due to large terms.

Writing the script happens in advance and requires prediction of upcoming proof states, in order to predict the next applicable proof step. When applying automatic strategies that may result in a large number of proof goals, the complexity of this prediction increases.

4 Integrating Direct Manipulation and Text-Based Interaction

To integrate text-based interaction into KeY, we developed the proof scripting language KPS for proof construction [4]. The basic idea behind this approach is to leverage the advantages of both interaction paradigms for interactive program verification. Allowing the user to construct proofs using both, direct manipulation and text-based interaction, adheres to the usability principle of *flexibility* (identified by [7] as being important for theorem provers). More precisely the integration 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.

By introducing this text-based interaction style proof construction becomes similar to regular software debugging, as there is an analogy between finding bugs in programs and finding the cause for a failed proof attempt. This analogy allowed us to adapt well-known concepts from software debugging for the user interface and the proof process [4].

KeY already provides the proof tree and each sequent as different views onto the proof state. Now adding text-based interaction style adds another view – the proof script. Challenges in displaying the different views to the user include that the views each have a clear purpose that is recognizable by the user. 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 consistency of views to follow the usability principle of *least surprise*.

We claim it is a central usability aspect to achieve consistency between the views, 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. 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. The invariant should also lead to a consistent view for the user while stepping through the proof script. Violations 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 detail.

Different Projections of the Proof State As already identified as guideline for theorem prover interfaces, multiple views support the user in the complex task of theorem proving [7]. We also devise to have different projections of the proof state, such that users can visually focus on their preferred representation of information in the current proof situation. Like in software debugging systems, where a potentially complex and large program state is presented to the user in different views, we also developed views that show different projections of the proof state. 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 and also close them, if they clutter the screen.

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. 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 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 (Fig. 1b) 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 [7], as with a separation of interwoven information the user can now choose the relevant information by choosing the preferred view. The standard view on the proof state in our concept 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 depiction of the proof obligation of the current 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.

With the different views present, changes need to be propagated to all views, to maintain the principle of *consistency*. The proof tree as well as the goal list have to be advanced when the script is executed. Additionally, if the user uses direct manipulation for proof construction the actions performed there need to be reflected in the script text.

Focussing 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. We consider the proof script as the central proof state representation which serves as a more abstract representation of the proof tree. Details, such as the application of single calculus rules by invoking 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 focussing on details, we adapt stepping functionalities from software debugging, where users can step into methods of a programs execution in case detailed insight is needed. In our approach users can step into proof commands to retrieve more details about the performed proof steps. In the case of macro steps in KeY, this stepping can result in a view that contains a whole subtree, containing all steps that were performed by invoking the macro step. Additionally, users can inspect the sequent of each proof node upon request.

Switching Interactions We combine two different interaction styles which can only be successful if the transition between them requires only little effort. 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. We chose for our approach to have two *modes* the user may switch: 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 is created which contains already precomputed 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 the proof exploration was not successful, or to add the script to the main script. This especially allows easy reversible actions and proof exploration. To maintain the invariant for a consistent state we have imposed the restriction that these scripts can not be added in loop constructs.

In the script-mode the proof script can be written and executed stepwise. 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.

Further Support for Proof Construction Adhering to the principle of *least effort*, which was instantiated for theorem prover interfaces by Völker [17], 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: the sequent matcher. In this window the user can enter a match expression and retrieves information which goals are matched by the entered expression.

5 Lessons Learned from Adding Text-Based Interaction to KeY

We performed first experiments using the KPS language within a first version of PSDBG to determine the effectiveness of the proof scripting language [10]. Based on our experiences in this experiment we have identified room for improvement of our approach and the implementation. In the following we introduce our experiment setup and summarize our experiences.

5.1 First Experiments Using KPS

The experiments aimed to asses the following aspects of KPS: (a) feasibility of deductive proofs, (b) stability of the script against changes in the program or specification, and (c) the conciseness.

(Relative) Feasibility refers to the possibility to formulate all proofs originally performed in KeY by using the proof commands offered by KPS. To estimate the feasibility of proofs we replayed existing large, non-trivial proofs of the Java standard library, i.e., the `compareMagnitude()` method of `BigInteger` class [15] and the `split()` method of the `DualPivotQuicksort` implementation [5]. The existing proofs we have replayed were originally performed using the point-and-click interface of KeY. While the proof of the `split()` method of the `DualPivotQuicksort` implementation can be successfully formulated in KPS, the transformation of the proof of the `compareMagnitude()` method of the `BigInteger` class revealed that not all proof commands were already implemented. Especially, issues in handling term referencing were revealed using this proof.

Stability indicates the degree of effects caused by minor changes to the program or its specification. To assess the stability of proofs scripts, minor changes, like renaming variables, simple rephrasing loop conditions, and repositioning of commutative terms, were made in the verified Java programs. Afterwards the saved scripts were re-executed to analyze the extent of the sustainability of the interaction in the scripts. The result of the experiments showed that proofs in KPS are more stable than KeY. Except for the case where variables were renamed all other changes lead to issues in reloading proofs in KeY, while in KPS proofs could still be reloaded after performing the changes.

Conciseness refers to the possibility to shorten KPS scripts in contrast to performed user interactions in KeY. It was possible to compress scripts during the experiment using multi-matching, which revealed that repetitive interactions can be compressed. However, the experiments revealed another issue: proof commands may require the term and its context (given as the formula on the sequent) to which the command should be applied to. In the first implementation these references had to be explicit, i.e., the whole subterm and formula have to be added to the script, leading to long scripts. Thus, the conciseness of the KPS is disputable as long as term matching expressions can not be used as parameters.

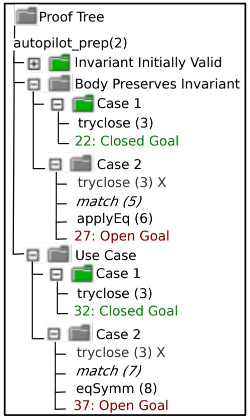
Using the user interface of PSDBG To evaluate how well proof construction using the interface of PSDBG is supported, we verified the sortedness of a variation of the BubbleSort algorithm, that sorts odd and even indices separately. The proof process is similar to the proof process in KeY (cf. Fig. 3) with some differences: the user first provides a proof script (step 1b, Fig. 3) in addition to the annotated Java program. For proof state inspection we used PSDBG breakpoints, the tree visualization and stepping functions to investigate failed proof attempts (step 3a), in addition to the functionalities already offered by the KeY system. The experiment showed that the PSDBG is still missing technical support to offer efficient and sufficient interactions. Proof exploration is therefore challenging for the user.

5.2 The Lessons Learned

The results of our first experiment showed that the scripting language concept developed for the KeY system is sufficient to reconstruct typical program verification proofs using KeY. In the following, lessons we have learned from the experiments are presented together with ideas for future work.

Breakpoints as Support for Orientation 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. 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. Our idea for this is sketched in Fig. 4, which shows a tree representation of the script combined with the proof tree. Nodes are statements from the script (containing a reference to their script line in parentheses) or branching nodes from the underlying proof tree. Proof commands that were not applicable are marked with an x.



```
Proof Tree
autopilot_prep(2)
├ ⊞ Invariant Initially Valid
├ ⊟ Body Preserves Invariant
│   ├ ⊟ Case 1
│   │   ├ tryclose (3)
│   │   └ 22: Closed Goal
│   └ ⊟ Case 2
│       ├ tryclose (3) X
│       ├ match (5)
│       ├ applyEq (6)
│       └ 27: Open Goal
└ ⊟ Use Case
    ├ ⊟ Case 1
    │   ├ tryclose (3)
    │   └ 32: Closed Goal
    └ ⊟ Case 2
        ├ tryclose (3) X
        ├ match (7)
        ├ eqSymm (8)
        └ 37: Open Goal
```

Figure 4: Condensed Proof Tree

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 [17]. 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 name and their surrounding formula. Our experiment showed that this representation leads to ambiguous matches, e.g., when a formula contained a subterm more than once. Using schematic expressions as parameters for proof commands, as considered in our concept, may amplify 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. Another disadvantage would be that the user needs to see 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 view support, the user has to be supported when inspecting terms or adding them to the proof script. 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 a resilient matching expressions are *deferred* in case the script is reloaded. 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. This suggestion mechanism has to be provided in the script (see also [17, 7]), 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. The first is that the user types parts of a command and gets suggestions for applicable commands. When chosen, the user gets visual highlights of terms where the command is applicable to and selects a suitable one by point-and-click, or using the keyboard. The second variation may be to select a term using direct manipulation to retrieve a list of applicable commands to choose from similar to KeY's current support. In case of more than one proof goal in the proof state, there has to be a support to construct the appropriate textual selector for the chosen goal. Additionally, to get context-sensitive support, the system needs to know the proof state in which the rules should be applied. Therefore, the proof has to be advanced to that state already while the user writes the script. 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.

To enable proof exploration, our approach includes the facility to use point-and-click in the interactive mode and to dispose the temporarily created script if not needed. The evaluated implementation only allowed undoing single interactively applied proof commands, and disposing whole interactive scripts is part of future work.

To mark the start of proof exploration using the script and for mass-reverting of proof commands, our approach considers user-defined persistent *safepoints* to be implemented into PSDBG for future work. A proof state can be saved using this action and recalled at a later stage, e.g., to dispose explorative steps.

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 [4] for more details). One feature, not yet present in KeY, is being currently implemented: a rewrite command that allows to replace a user-provided term by another term. The execution of this command would try to rewrite the term using KeY's calculus rule and if not possible result in 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.

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.

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 currently being developed. This functionality will also benefit from using differences in the proof objects.

6 Related Work

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 [7]. 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 focussing 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 [17]. 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 [17]. We added different ideas and a discussion for handling the reference to large formulas in program verification proofs.

The authors of the guidelines [17, 7] contrarily discussed the benefit of having a visual proof tree. While it can serve as an aid for orientation and abstraction [7], also evidence for being not helpful was assembled in [17]. 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 [6] and Isabelle/HOL [14]. 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[1] 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 offer direct manipulation interaction and add text-based support afterwards. KeYmaeraX [12] and also our contribution are examples for such systems. 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. Open goals are presented in a tabbed view, where users can switch goals by selecting the appropriate tab. Navigating up and down in the proof is done using unfolding actions. Control-flow structures for exhaustively applying rules are present in the script language, however explicit multi-matching is missing.

The third approach for a combination of both styles was done by combining two existing interfaces of a general purpose theorem prover, such as the approach of PGWin for the theorem prover Isabelle [2]. 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.

7 Conclusion

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 have carried out first experiments by transforming non-trivial proofs saved in KeY's proof format to KPS proof scripts to evaluate our proof scripting language. Furthermore, we used PSDBG 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, handling term references in the script, the full support of all KeY proof commands, as well as the implementation of sophisticated features for adjusting our different views onto the proof state, e.g., in the proof tree. Functionalities adapted from software debugging such as using breakpoints and stepwise 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.

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.

[1] The addition for Coq is available at `https://askra.de/software/prooftree/`.

References

[1] Wolfgang Ahrendt, Bernhard Beckert, Richard Bubel, Reiner Hähnle, Peter H. Schmitt & Mattias Ulbrich, editors (2016): *Deductive Software Verification - The KeY Book: From Theory to Practice*. *LNCS* 10001, Springer, doi:10.1007/978-3-319-49812-6.

[2] David Aspinall & Christoph Lüth (2004): *Proof General meets IsaWin: Combining Text-Based And Graphical User Interfaces*. *Electr. Notes Theor. Comput. Sci.* 103, pp. 3–26, doi:10.1016/j.entcs.2004.09.011.

[3] B. Beckert, S. Grebing & A. Weigl (2018): *Debugging Program Verification Proof Scripts (Tool Paper)*. *ArXiv e-prints*.

[4] Bernhard Beckert, Sarah Grebing & Mattias Ulbrich (2017): *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*, *Lecture Notes in Computer Science* 10629, Springer, pp. 163–178, doi:10.1007/978-3-319-70389-3_11.

[5] Bernhard Beckert, Jonas Schiffl, Peter H. Schmitt & Mattias Ulbrich (2017): *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*, *Lecture Notes in Computer Science* 10712, Springer, pp. 35–48, doi:10.1007/978-3-319-72308-2_3.

[6] Yves Bertot & Pierre Castran (2004): *Interactive Theorem Proving and Program Development: Coq'Art The Calculus of Inductive Constructions*, 1st edition. Texts in Theoretical Computer Science An EATCS Series, Springer-Verlag Berlin Heidelberg.

[7] Katherine A. Easthaughffe (1998): *Support for Interactive Theorem Proving: Some Design Principles and Their Application*. *User Interfaces for Theorem Provers (UITP 1998)*.

[8] Stijn de Gouw, Jurriaan Rot, Frank S. de Boer, Richard Bubel & Reiner Hähnle (2015): *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*, pp. 273–289, doi:10.1007/978-3-319-21690-4_16.

[9] Gary T. Leavens, Albert L. Baker & Clyde Ruby (2006): *Preliminary Design of JML: A Behavioral Interface Specification Language for Java*. *SIGSOFT/SEN* 31(3), pp. 1–38.

[10] An Thuy Tien Luong (2018): *Evaluation der Proof-Script-Sprache für KeY (in German)*. Bachelor's Thesis at Karlsruhe Institue of Technology.

[11] N. Merriam & M. Harrison (1997): *What is wrong with GUIs for theorem provers*. In: *User Interfaces for Theorem Provers (UITP 1997)*.

[12] Stefan Mitsch & André Platzer (2016): *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.*, pp. 67–81, doi:10.4204/EPTCS.240.5.

[13] Jakob Nielsen (1994): *Enhancing the Explanatory Power of Usability Heuristics*. In: *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '94, ACM, New York, NY, USA, pp. 152–158, doi:10.1145/191666.191729.

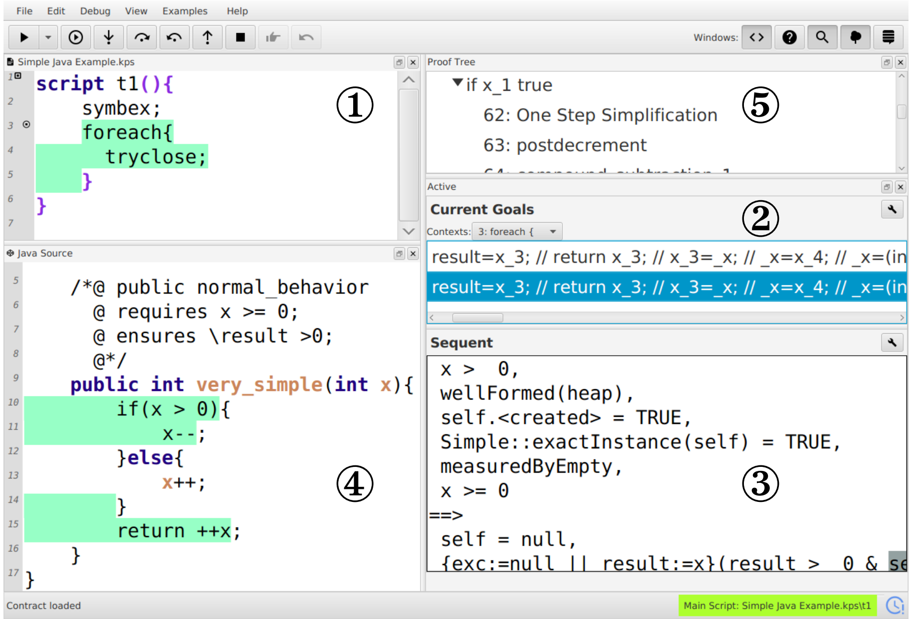
[14] Tobias Nipkow, Lawrence C. Paulson & Markus Wenzel (2002): *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*. *LNCS* 2283, Springer.

[15] Wolfram Pfeifer (2017): *Specifying and Verifying Real-World Java Code with KeY Case Study java.math.BigInteger*. Bachelor's Thesis at Karlsruhe Institue of Technology.

[16] Ben Schneiderman (1983): *Direct Manipulation. A Step Beyond Programming Languages*. *IEEE Transactions on Computers* 16(8), p. pp. 5769.

[17] Norbert Völker (2003): *Thoughts on Requirements and Design Issues of User Interfaces for Proof Assistants.* In: *User Interfaces for Theorem Provers (UITP 2003)*.

A Screenshots



```
script t1(){
    symbex;
    foreach{
      tryclose;
    }
}
```

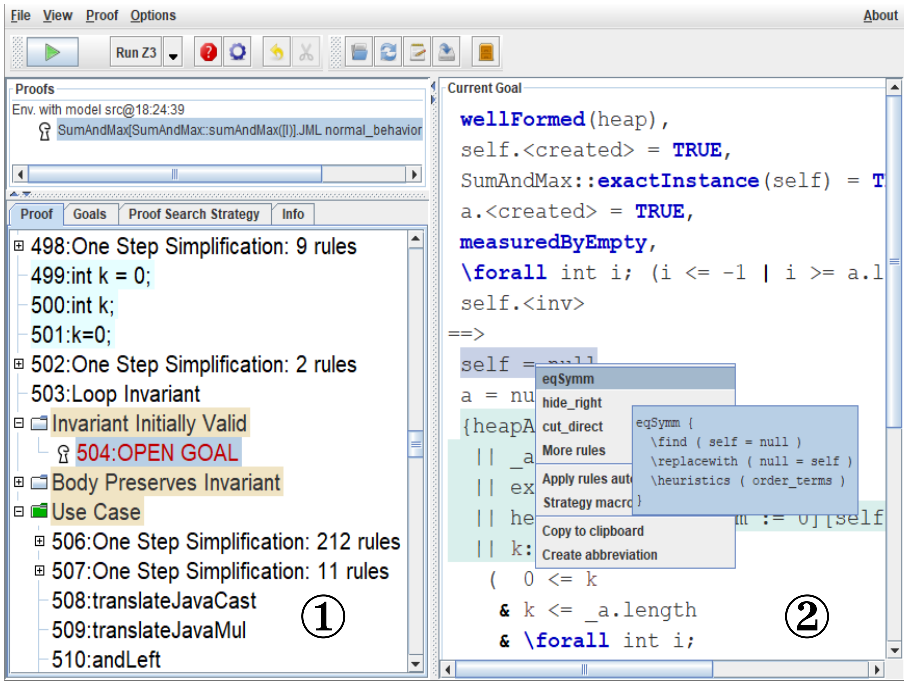
```
▾if x_1 true
    62: One Step Simplification
    63: postdecrement
```

```
result=x_3; // return x_3; // x_3=_x; // _x=x_4; // _x=(in
result=x_3; // return x_3; // x_3=_x; // _x=x_4; // _x=(in
```

```
    /*@ public normal_behavior
      @ requires x >= 0;
      @ ensures \result >0;
      @*/
    public int very_simple(int x){
        if(x > 0){
            x--;
        }else{
            x++;
        }
        return ++x;
    }
}
```

```
 x >  0,
 wellFormed(heap),
 self.<created> = TRUE,
 Simple::exactInstance(self) = TRUE,
 measuredByEmpty,
 x >= 0
==>
 self = null,
 {exc:=null || result:=x}(result >  0 & se
```

Figure 5: The user interface of PSDBG



```
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 6: The user interface of KeY with interactive rule application