An Interaction Concept for Program Verification Systems with Explicit Proof Object

Bernhard Beckert, Sarah Grebing, and Mattias Ulbrich

Karlsruhe Institute of Technology
{beckert,sarah.grebing,ulbrich}@kit.edu

Abstract. Deductive program verification is a difficult task: in general, user guidance is required to control the proof search and construction. Providing the right guiding information is challenging for users and usually requires several reiterations. Supporting the user in this process can considerably reduce the effort of program verification.
In this paper, we present an interaction concept for deductive program verification systems that combines point-and-click interaction with the use of a proof scripting language. Our contribution is twofold: Firstly, we present a concept for a flexible and concise proof scripting language tailored to the needs of program verification. Secondly, we explore the correspondences between program debugging and proof debugging and introduce a concept for analysing failed proof attempts which leverages well-established concepts from software debugging. We illustrate our concepts on examples – including small Java programs with non-trivial specifications – using an early prototype implementation of our interaction concepts that is built on top of the program verification system KeY.

1 Introduction

Research in automatic program verification has made a huge progress in recent years. Nevertheless, in the foreseeable future, there will always be programs and properties that are of importance in practice but for which verification systems cannot find correctness proofs automatically without user guidance [1]. Finding the right guiding information that allows a verification system to find a proof is, in general, an iterative process of repeated failed attempts.

Program verification proofs have characteristics 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.

We present an interaction concept based on using a proof scripting language together with a proof development and debugging approach, tailored to the needs of program verification. Our first contribution is 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 the possibility to define selection criteria that choose several goals at a time that are then treated uniformly. These selection criteria are resilient to change in the sense that small changes in the proof require small changes in the proof script describing that proof.

Two interaction paradigms have emerged in state-of-the-art interactive verification systems: text-based interaction (proof scripts and source code annotations) and point-and-click interaction. Compared to scripting languages where single proof statements apply to only one goal, and to a textual recording of pure point-and-click interactions, a scripting language with multi-matching allows creating more compact proof scripts.

However, powerful concepts like multi-matching, which allow proof scripts whose structure is different from the proofs they describe, have to be complemented with a suitable method to debug failed proof attempts. Thus, as a second contribution of this paper, we introduce a concept for interactive proof development. The focus of this concept is to aid the user in comprehending failed proof attempts and identifying the next step to successfully continue the proof. Proofs can be constructed using a proof scripting language as well as direct manipulation of the proof object using point-and-click interaction.

We showcase our concept, which is particularly well suited for verification systems with explicit proof objects using a sequent calculus, by applying the concept for the interactive program verifier KeY [2].

The remainder of this paper is structured as follows: In Sections 2 and 3, we discuss the proof characteristics of interactive program verification and related work. Then, we introduce the concepts for a proof scripting language tailored to the peculiarities of proofs in this domain in Section 4; and we present a concept for debugging proofs performed using a scripting language in Section 5, making use of functionalities that are adapted from program debugging. We conclude and discuss future work in Section 6.

2 Interactive Program Verification

Program verification proofs differ from mechanised proofs of mathematical theorems, particularly in the size and complexity of the occurring formulas and in the number of different cases to investigate. Program verification proofs often have a large number of individual subgoals reflecting the control-flow possibilities in the program.

Each subgoal represents 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. During proof construction, the user typically switches between focusing on one particular proof goal and looking at a number of proof branches 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.

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 is 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? 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 characterised by trial-and-error phases. 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. However, 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).

The KeY system. The design of our concept is based on the results of two focus group experiments [3,4] and is targeted towards rule-based program verification systems operating on program logics. Our primary target, in which we exemplarily realize the concept, is the interactive Java verification tool KeY.

The typical workflow of KeY is depicted in Fig. 1: Initially, the user provides a Java program, together with a specification formulated in the Java Modelling Language [5] (step 1a). Proof obligations in KeY are formalised in a program logic called Java DL, and proofs are conducted using a sequent calculus [2]. 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.

Proofs in KeY are organised in directed, labelled trees whose vertices are called proof nodes. Each node is labelled with a sequent, the root is labelled with the original proof obligation. Inner nodes are additionally labelled with the

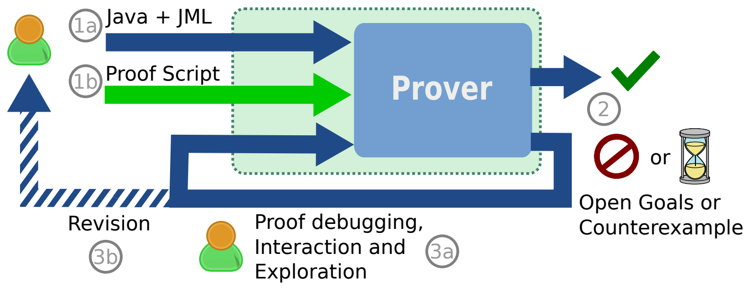


Fig. 1. Interactive Program Verification using scripts

calculus rule that was used to construct the node. When interactively applying rules, KeY allows the user to inspect the whole proof tree with all applied rules. Proof search and guidance is done by using point-and-click interaction, where the user points to a formula and mouseclicks on it to apply a rule. Besides the application of single calculus rules, it is also possible to apply sets of rules in so called *macro* steps, which we also call *prover strategies*. Two important strategies in this paper are `auto` and `symbex`. While `auto` applies all admissible rules, `symbex` only applies rules performing symbolic execution of the Java program. In this work, we introduce proof scripts (step 1b) to provide an additional way of interacting with the program verification system.

3 Related Work

Many general purpose proof assistants using higher-order logic feature text-based interaction (e.g., Isabelle/HOL [6] and Coq [7]). 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 [8] 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 [9] for Isabelle or MTac [10] for Coq) to enable users to program their own tactics tailored to the proof problem. *ProofScript* [11] 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 *ProofPeer* 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 [9] for an overview of proof languages.

There also exist approaches to debugging proof tactics and gain more insight. For example, Tinker2 [12] is a graphical tool for inspecting the flow of goals in proof tactics. And Hentschel [13] 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.

4 Concept for a Proof Scripting Language

It is part of our interaction concept to support the combination of point-and-click with scripting. The control-structures of our proof scripting language can be used to control the application of strategies of the underlying verification system. The basic principles of the language are introduced in the following.

Important Features. The characteristics of proofs for program verification (Sect. 2) 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
   - to identify and select individual proof branches,
   - to easily switch between proof branches,
   - to select multiple branches for uniform treatment (*multi-matching*);

   that is resilient to small changes in the proof;
3. a repetition construct which allows repeated application of proof strategies;
4. support for proof exploration within the language.

The objects manipulated during proof construction are called *proof goals*. We assume that each proof goal is unique and identifiable by its contents (e.g., its sequent, when using a sequent calculus).

Applying calculus rules or proof strategies to a proof goal results in the creation of new proof goals that are added to the proof.

Performing proof construction is characterised 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 such cases, the user may try different proof strategies or different lightweight techniques (such as, bounded model-checkers to find counterexamples). 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.

4.1 Preliminaries for the Proof Scripting Language

In the following, we introduce a concept for a proof scripting language taking the aforementioned principles into account. We present it using an abstract syntax and demonstrate the language constructs on smaller examples within the KeY system.

The script language supports local variables of types boolean and integer, and of domain-specific types such as goal, formula and term. Expressions can be constructed using arithmetic operators, boolean connectives, subterm selection, and substitution expressions for concrete and schematic terms and formulas. Evaluations of expressions and assignments to variables are defined as usual.

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 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.

Running Example. Our example (see Listing 1) uses a Java class `Simple` with a method `transitive(int[] a)`, which creates a copy of the argument array, sorts it, and copies the result. The goal is to prove (using KeY) that, after the execution of `transitive()`, the output array is a permutation of the input array. After applying KeY's symbolic execution strategy and a simplification strategy, the user is left with eleven open goals of which four cases correspond to the post states of the two conditional statements (in lines 11 and 12). These cases are similar as they share the same postcondition and differ only a little in their path conditions. For each of these cases, it has to be shown that the output array is a permutation of the input array, i.e., that the permutation property is preserved across the method calls in the body of `transitive()`.

The informal argument for why this holds is that the invoked methods `copyArray()` and `sort()` each preserve the permutation property (as specified in lines 17 and 25), and that the method `log()` does not change the heap (line 30). These methods are called in the body of `transitive()` on the array `a` (lines 8–13).

In the following, we first demonstrate script language features on smaller examples but will finally return to our running example at the end of this section and show a full script for the proof.

4.2 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 and use the KeY system as a showcase for our examples. The abstract syntax of our language concept is summarised in Fig. 2.

Mutators. Mutators (M in Fig. 2) are the most basic building blocks that when executed change the script state and the proof state by adding nodes to

```java
public final class Simple {
   boolean b1, b2;

   /*@ public normal_behavior
     @ ensures seqPerm(array2seq(\result), \old(array2seq(a)));
     @ assignable \everything;  */
   public int[] transitive(int[] a){
       a = Simple.copyArray(a);
       sort(a);
       int[] b = Simple.copyArray(a);
       if(b1) { b = Simple.copyArray(a); }
       if(b2) { log(b); }
       return b;
   }

   /*@ public normal_behavior
     @ ensures seqPerm(array2seq(a), \old(array2seq(a)));
     @ assignable a[*];  */
   public void sort(int[] a) { /* in-place sorting */ }

   /*@ public normal_behavior
     @ ensures (∀ int i; 0 <= i < input.length; input[i]==\result[i])
     @ && \result.length == input.length;
     @ ensures \fresh(\result);
     @ ensures seqPerm(array2seq(\result), array2seq(input));
     @ assignable \nothing;    */
   public /*@ helper @*/ static int[] copyArray(int[] input) { /* deep-copy */ }

   /*@ public normal_behavior
     @ assignable \strictly_nothing;    */
   public void log(int[] a) { /* ... */}
}
```

Listing 1. Java program with JML annotations (running example).

the proof tree. 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. Additionally, the language allows calling other scripts as mutators.

The semantics for both mutator types is similar: they change the set of open proof goals of the proof state. However, native proof commands are only applicable to a single goal in our concept. If the goal set of a proof state consists of more than one goal, it is ambiguous to which of these the command should be applied. To avoid confusing results, we define this to result in an error state.

The termination of native proof commands depends on the underlying proof system. Native commands that may run indefinitely long thus allow the specification of a timeout or a maximal number of rules application as arguments.

Example 1. The mutator

$$\texttt{applyEq on=}\overbrace{\texttt{'==> x==y'}}^{\textit{mutation target}}\texttt{ with=}\overbrace{\texttt{'y==1 ==>'}}^{\textit{side condition}}$$

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

$$
\begin{aligned}
M &::= (\mathit{script\_name} \mid \mathit{native\_command})\ \mathit{args}\\
C &::= C_1; C_2 \mid \mathit{var}\ := \ \mathit{expression} \mid \texttt{repeat}\ \{C\} \mid \texttt{foreach}\ \{C\} \mid \texttt{theonly}\ \{C\}\\
&\quad \mid \texttt{cases}\ \{\texttt{case}\ S_1 : \{C_1\}\ \ldots\ \texttt{case}\ S_n : \{C_n\}\ \} \mid S\\
S &::= \mathit{expression} \mid \texttt{matchSeq}\ \mathit{schemaSeq} \mid \texttt{closes}\ \{C\} \mid\\
&\quad \mid \texttt{matchLabel}\ \mathit{regexp} \mid \texttt{matchRule}\ \mathit{rulename}
\end{aligned}
$$

Fig. 2. Abstract syntax of the proof scripting language.

succedent (the part right of `==>`), replacing `x==y` with the formula `x==1`.

$$\overbrace{\texttt{x==1, y==1 ==> x==y}}^{\text{state before applyEq}} \quad\rightsquigarrow\quad \overbrace{\texttt{x==1, y==1 ==> x==1}}^{\text{state after applyEq}}$$

If either of the formulas `y==1` and `x==y` is not present in the goal, this mutator is not applicable.

Control Flow. Besides sequential composition and variable assignment, the language supports control structures (C in Fig. 2) targeting command application to one or more proof goals. To be able to apply proof commands to a single goal node 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 state anymore.

Example 2. Consider the following example script for KeY containing a repeat command: `repeat { andLeft }`. 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 formulas 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, the rule `andLeft` is only applied to the conjunctions in the antecedent.

Selectors. As the application of calculus rules can cause a proof goal to split into different cases, it would be ambiguous to apply a proof command after a split. Therefore, one must be able to indicate to which proof goals a proof command is to be applied. Selectors (S in Fig. 2) can be used to select one or more proof goals. Our language concept includes the *cases*-command for this purpose. It is tailored to the needs of proving in the domain of program verification, allowing the formulation of proof goal sets using *matching conditions*. These are expressions evaluated for each proof goal; all goals which satisfy a matching condition S_i are then subject to the corresponding proof command C_i. Thus uniform treatment for several goals can be realised. If a proof goal satisfies more than one matching condition, the first one wins. 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 have been executed.

In our language concept, we support three fundamentally different types of matching conditions: *State conditions* consist of an expression over the script variables. 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* (notated 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 evaluation 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` can be used in situations where the user expects that there is exactly one goal in the proof state. If there is more than one when the command is evaluated, a warning is passed to the user.

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 3. Consider the following simple example for the use of a matching condition within a cases selector, where the template matches sequents containing an implication in the succedent:

```
case matchSeq '==> ?A -> ?B' : { impRight; andLeft on='?A' }
```

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 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 in the sequent, 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 statement "`closes` { C }". It examines whether applying the proof command C would close the current goal (without actually effecting the current state). Besides its use for exploration, `closes` can be used in the cases statement as matching condition.

Example 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 can be expressed using `closes` as follows: `closes (assume '==> F'; auto)`, where `assume '==> F'` is a proof command adding `F` to the succedent. Adding arbitrary formulas to the proof obligation during proof construction is unsound. Thus, the `assume` command is only allowed in `closes` statements. The proof command `auto` is then used to try to prove the newly created proof obligation.

Explorations that check whether a certain formula is derivable (as shown in the above example), 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 matchSeq 'x > 0'` would miss the goal node, an expression checking for derivability of `x > 0` would match the sequent.

Running Example. In Fig. 2, a proof script for proving the correctness of the method `transitive` (see Fig. 1) is shown, which uses the building blocks described above. After symbolic execution and some simplification steps (lines 2–3), the KeY system stops in a state with 11 open goals. The tricky cases are those where the postcondition of the method `transitive` has to be shown to be consequences of the postconditions of the called methods `copyArray, sort` and `log`. Corresponding schematic sequent templates (lines 6–10 and 22–25) are then used in a cases statement to select the relevant goal nodes which need user interaction. The cases statements select goal nodes that contain predicates `seqPerm(seq1, seq2)` formalising that sequence `seq1` is a permutation of sequence `seq2`. Rules deriving relations about different heaps using the symmetry and transitivity properties of the permutation predicate are applied (lines 11–18 and 26–31). Each condition matches two goals, the commands close them. To all other goal nodes not selected by the two matching conditions, the proof command `auto` is applied with at most 10000 rule applications (line 34).

Without the script and the matching feature it uses, the rule applications in the two cases statements would have to be applied separately to each of the four open branches. Additionally, the two cases are similar, so the user is able to copy-paste the first case and adjust it to the situation of the second case. Note that the scripting language is especially useful when used together with the point-and-click features of the system, to ease the selection process for applying rules/strategies onto terms. This allows one to make use of the mechanism for suggesting applicable rules of the underlying system.

5 Concept for Debugging Proof Attempts

5.1 Analogy between Programs and Proof Scripts

Scripts formulated in a scripting language like the one presented in the previous section can be considered to be "programs" that construct (partial) proofs for

```
script prove_transitive() {
  symbex;                          // perform symbolic execution of the program
  foreach { heapSimplification; }  // simplify heap terms
  cases {
    case matchSeq
      'seqPerm(?Res0Copy, ?Arr),
       seqPerm(?Res0Sort, ?Res0Copy),
       seqPerm(?Res1Copy0, ?Res0Sort),
       seqPerm(?Res2Copy1, ?Res0Sort) ==>
       seqPerm(?Res2Copy1, ?Arr)':
    { SeqPermSym on='seqPerm(?Res0Copy, ?Arr) ==>';        // symmetry rule
      SeqPermSym on='seqPerm(?Res0Sort, ?Res0Copy) ==>';   // symmetry rule
      SeqPermSym on='seqPerm(?Res1Copy0, ?Res0Sort)==>';   // symmetry rule
      SeqPermSym on='seqPerm(?Res2Copy1, ?Res0Sort) ==>';  // symmetry rule
      SeqPermTrans on='seqPerm(?Res0Copy, ?Arr) ==>';      // transitivity rule
      SeqPermTrans on='seqPerm(?Arr, ?Res0Sort) ==>'       // transitivity rule
                   with='seqPerm(?Arr,?Res2Copy1)';        // with specific term
      SeqPermSym on='seqPerm(?Arr,?Res2Copy1)';
      auto maxSteps=10000               // automatic strategy with 10000 rule applications
    }
    case matchSeq
      'seqPerm(?Res0Copy, ?Arr),
       seqPerm(?Res0Sort, ?Res0Copy),
       seqPerm(?Res1Copy0, ?Res0Sort) ==>
       seqPerm(?Res1Copy0, ?Arr)':
    { SeqPermSym on='seqPerm(?Res0Copy, ?Arr)';
      SeqPermSym on='seqPerm(?Res0Sort, ?Res0Copy)';
      SeqPermSym on='seqPerm(?Res1Copy0, ?Res0Sort)';
      SeqPermTrans on='seqPerm(?Res0Copy, ?Arr)';
      SeqPermTrans on='seqPerm(?Arr, ?Res0Sort)'
      SeqPermSym On='==> seqPerm(?Res1Copy0, ?Arr)';
      auto maxSteps=10000
    }
    case true: { auto maxSteps=10000 }
  }
}
```

Listing 2. Example proof script for method `transitive()`.

a 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 analysing failed proof attempts on the other. The main analogies between the two processes are summarised in Table 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 Sect. 4) 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 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

Table 1. Analogies between program debugging and debugging failed proof attempts.

Proof Debugging	$\leftrightarrow$	Program Debugging
proof script	$\leftrightarrow$	program source code
script state (incl. proof state)	$\leftrightarrow$	program state
sources and open proof goal(s)	$\leftrightarrow$	program input
proof tree	$\leftrightarrow$	traces of all threads
proof branch	$\leftrightarrow$	trace of an individual thread
partial proof	$\leftrightarrow$	trace of an incomplete program run
completed proof	$\leftrightarrow$	trace of a successfully terminating program run

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 if these differ in their expected outputs.

5.2 Analogy between Debugging and Failed Proof Analysis

Software debugging is the analysis process of understanding unexpected program behaviour, localising the responsible piece of code, and mending it. Typically, a concrete run of the program exposing the bug is analysed using specialised software (a debugger) which supports the user in the process by various means of visualisation 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 mechanising 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 behaviour. This process involves *localising* the responsible part of the proof and identifying the type of failure: Does the underlying 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? 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.

5.3 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) proofs. We focus on user support for the activities of localisation, comprehension, and exploration. Additionally, we adapt the presentation of program states for script states, allowing a detailed inspection.

A screenshot of our early prototype[1] (based on the KeY system) realising these concepts is shown in Fig. 3.

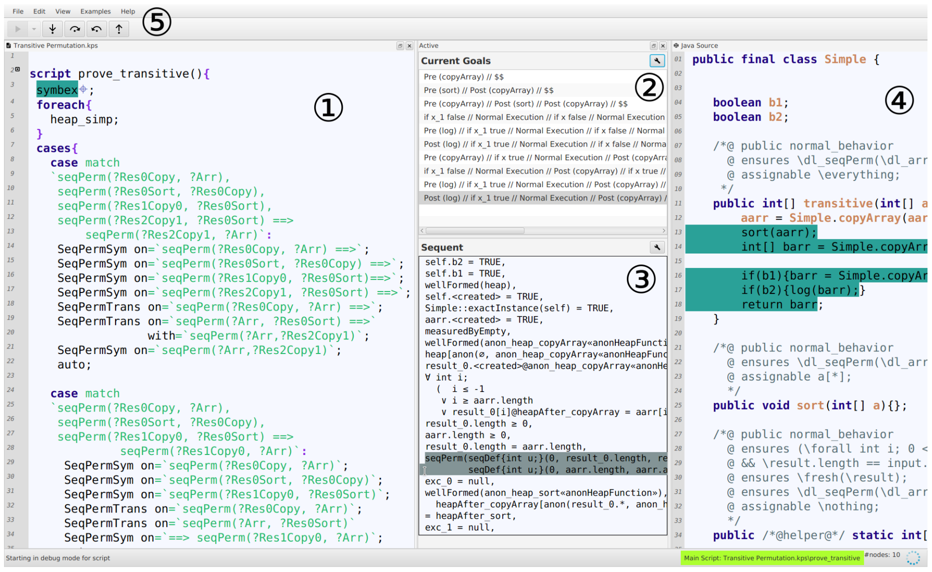
State Presentation. Program states in software debugging may be very complex. To support the user in inspecting and understanding a state, debugging systems present the state's information in a structured manner.

Our concept for proof states includes a structured presentation and functionalities for inspecting the state similar to program debugging systems. For this, we have identified the following parts of a state that should be visualised in isolation: (a) the proof tree with a visual highlight of the current node (i.e., the node containing the open goal to which the currently active proof command is being applied), (b) sequent of the current node (i.e., the current open goal), (c) the currently active proof command 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.

Localisation. To support the user in localising the cause of a defective behaviour, debugging systems provide *breakpoints*. These allow the user to inspect the program execution in detail when a program location is reached.

In the setting of program verification, defective behaviour corresponds to a proof with open goals, and the user is mostly interested in understanding these. In our concept, using point-and-click interaction with 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

[1] `http://formal.iti.kit.edu/key-psdebugger`


File
Edit
View
Examples
Help
⑤
Transitive Permutation.kps
①
Active
Current Goals
②
Sequent
③
Java Source
④
Starting in debug mode for script
Main Script: Transitive Permutation.kps\prove_transitive
#nodes: 10

```
script prove_transitive(){
 symbex;
 foreach{
   heap_simp;
 }
 cases{
   case match
   `seqPerm(?Res0Copy, ?Arr),
    seqPerm(?Res0Sort, ?Res0Copy),
    seqPerm(?Res1Copy0, ?Res0Sort),
    seqPerm(?Res2Copy1, ?Res0Sort) ==>
        seqPerm(?Res2Copy1, ?Arr)`:
    SeqPermSym on=`seqPerm(?Res0Copy, ?Arr) ==>`;
    SeqPermSym on=`seqPerm(?Res0Sort, ?Res0Copy) ==>`;
    SeqPermSym on=`seqPerm(?Res1Copy0, ?Res0Sort)==>`;
    SeqPermSym on=`seqPerm(?Res2Copy1, ?Res0Sort) ==>`;
    SeqPermTrans on=`seqPerm(?Res0Copy, ?Arr) ==>`;
    SeqPermTrans on=`seqPerm(?Arr, ?Res0Sort) ==>`
                with=`seqPerm(?Arr,?Res2Copy1)`;
    SeqPermSym on=`seqPerm(?Arr,?Res2Copy1)`;
    auto;

   case match
   `seqPerm(?Res0Copy, ?Arr),
    seqPerm(?Res0Sort, ?Res0Copy),
    seqPerm(?Res1Copy0, ?Res0Sort) ==>
             seqPerm(?Res1Copy0, ?Arr)`:
     SeqPermSym on=`seqPerm(?Res0Copy, ?Arr)`;
     SeqPermSym on=`seqPerm(?Res0Sort, ?Res0Copy)`;
     SeqPermSym on=`seqPerm(?Res1Copy0, ?Res0Sort)`;
     SeqPermTrans on=`seqPerm(?Res0Copy, ?Arr)`;
     SeqPermTrans on=`seqPerm(?Arr, ?Res0Sort)`
     SeqPermSym on=`==> seqPerm(?Res1Copy0, ?Arr)`;
```

```
Pre (copyArray) // $$
Pre (sort) // Post (copyArray) // $$
Pre (copyArray) // Post (sort) // Post (copyArray) // $$
if x_1 false // Normal Execution // if x false // Normal Execution
Pre (log) // if x_1 true // Normal Execution // if x false // Normal
Post (log) // if x_1 true // Normal Execution // if x false // Norma
Pre (copyArray) // if x true // Normal Execution // Post (copyArra
if x_1 false // Normal Execution // Post (copyArray) // if x true //
Pre (log) // if x_1 true // Normal Execution // Post (copyArray) //
Post (log) // if x_1 true // Normal Execution // Post (copyArray) /
```

```
self.b2 = TRUE,
self.b1 = TRUE,
wellFormed(heap),
self.<created> = TRUE,
Simple::exactInstance(self) = TRUE,
aarr.<created> = TRUE,
measuredByEmpty,
wellFormed(anon_heap_copyArray«anonHeapFuncti
heap[anon(∅, anon_heap_copyArray«anonHeapFunc
result_0.<created>@anon_heap_copyArray«anonHe
∀ int i;
  (  i ≤ -1
   ∨ i ≥ aarr.length
   ∨ result_0[i]@heapAfter_copyArray = aarr[i
result_0.length ≥ 0,
aarr.length ≥ 0,
result_0.length = aarr.length,
seqPerm(seqDef{int u;}(0, result_0.length, re
        seqDef{int u;}(0, aarr.length, aarr.a
exc_0 = null,
wellFormed(anon_heap_sort«anonHeapFunction»),
  heapAfter_copyArray[anon(result_0.*, anon_h
= heapAfter_sort,
exc_1 = null,
```

```
public final class Simple {


   boolean b1;
   boolean b2;

   /*@ public normal_behavior
     @ ensures \dl_seqPerm(\dl_arr
     @ assignable \everything;
    */
   public int[] transitive(int[] a
       aarr = Simple.copyArray(aar
       sort(aarr);
       int[] barr = Simple.copyArr

       if(b1){barr = Simple.copyAr
       if(b2){log(barr);}
       return barr;
   }

   /*@ public normal_behavior
     @ ensures \dl_seqPerm(\dl_arr
     @ assignable a[*];
     */
   public void sort(int[] a){};

   /*@ public normal_behavior
     @ ensures (\forall int i; 0 <
     @ && \result.length == input.
     @ ensures \fresh(\result);
     @ ensures \dl_seqPerm(\dl_arr
     @ assignable \nothing;
     */
   public /*@helper@*/ static int[
```

Fig. 3. 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 from Listing 2); the currently active proof command is highlighted in blue. In the middle (2), the open goals of the current proof state are listed; here, the last 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 toolbar (5) shows UI elements for stepping through the proof script.

localise 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.*

A *regular breakpoint* is a syntactical marker that represents a location in the proof script. If, in debug mode, execution of the proof script 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 are 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, searchpoints are the starting point. 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.

Conditions in breakpoints and searchpoints can be boolean expression 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 detailed inspection, the next activity is to stepwise retrace what state changes are made by the proof script. To simplify this process, the proof 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). This stepwise 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 task of forming hypotheses about the cause of a defect 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 behaviour of the proof script by actively changing the proof state in debug mode. 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., analyse whether the origin of the part of state that was changed (e.g., the precondition of the program) has to be adapted.

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 in a successful way.

6 Conclusion and Future Work

We have presented an interaction concept for deductive program verification systems that combines point-and-click interaction with the use of a proof scripting language. This concept introduces a flexible and concise proof scripting language tailored to the needs of program verification. In this domain, proofs often consist of many structurally and/or semantically similar cases which are syntactically large but of small intrinsic complexity. Using matching mechanisms, the language provides means taylored to this type of proofs.

Further, we have explored the correspondences between program debugging and proof debugging and introduced a concept for analysing failed proof attempts, which leverages well-established concepts from software debugging.

A prototypical implementation using the KeY system and a case study is currently work in progress. It remains for future work to evaluate the effectiveness of the concepts by performing usability studies.

Acknowledgements. Special thanks go to Alexander Weigl who provided valuable comments concerning the proof debugging concept, the realisation of the script language, and the prototype.

References

1. Hähnle, R., Huisman, M.: Deductive software verification: From pen-and-paper proofs to industrial tools. In: LNCS 10000. (2017)
2. Ahrendt, W., Beckert, B., Bubel, R., Hähnle, R., Schmitt, P.H., Ulbrich, M., eds.: Deductive Software Verification - The KeY Book: From Theory to Practice. Volume 10001 of LNCS. Springer (2016)
3. Beckert, B., Grebing, S., Böhl, F.: How to put usability into focus: Using focus groups to evaluate the usability of interactive theorem provers. In Benzmüller, C., Woltzenlogel Paleo, B., eds.: UITP 2014. Volume 167 of EPTCS. (July 2014) 4–13
4. Beckert, B., Grebing, S., Böhl, F.: A usability evaluation of interactive theorem provers using focus groups. In Canal, C., Idani, A., eds.: 12th International Conference on Software Engineering and Formal Methods (SEFM 2014) – Collocated Workshops: Human-Oriented Formal Methods (HOFM 2014). Volume 8938 of LNCS., Springer (September 2014) 3–19
5. Leavens, G.T., Baker, A.L., Ruby, C.: Preliminary design of JML: A behavioral interface specification language for Java. SIGSOFT/SEN **31**(3) (2006) 1–38
6. Nipkow, T., Paulson, L.C., Wenzel, M.: Isabelle/HOL — A Proof Assistant for Higher-Order Logic. Volume 2283 of LNCS. Springer (2002)
7. Bertot, Y., Castran, P.: Interactive Theorem Proving and Program Development: Coq'Art The Calculus of Inductive Constructions. 1st edn. Texts in Theoretical Computer Science An EATCS Series. Springer-Verlag Berlin Heidelberg (2004)
8. Wenzel, M.: 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) 167–184
9. Matichuk, D., Murray, T., Wenzel, M.: Eisbach: A proof method language for isabelle. Journal of Automated Reasoning **56**(3) (Mar 2016) 261–282
10. Ziliani, B., Dreyer, D., Krishnaswami, N.R., Nanevski, A., Vafeiadis, V.: Mtac: A monad for typed tactic programming in coq. SIGPLAN Not. **48**(9) (September 2013) 87–100
11. Obua, S., Scott, P., Fleuriot, J.: Proofscript: Proof scripting for the masses. In Sampaio, A., Wang, F., eds.: Theoretical Aspects of Computing – ICTAC 2016: 13th International Colloquium, Taipei, Taiwan, ROC, October 24–31, 2016, Proceedings, Cham, Springer International Publishing (2016) 333–348
12. Lin, Y., Le Bras, P., Grov, G.: 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) 573–579
13. Hentschel, M.: Integrating Symbolic Execution, Debugging and Verification. PhD thesis, Technische Universität Darmstadt (January 2016)