Seamless Interactive Program Verification

Sarah Grebing[1], Jonas Klamroth[2], and Mattias Ulbrich[1]

[1] Karlsruhe Institute of Technology
[2] FZI Forschungszentrum Informatik

Abstract. Deductive program verification has made considerable progress in recent years. Automation is the goal, but it is apparent that there will always be challenges that cannot be verified fully automatically, but require some form of user input. We present a novel user interaction concept that allows the user to interact with the verification system on different abstraction levels and on different verification/proof artifacts. The elements of the concept are based on the findings of qualitative user studies we conducted amongst users of interactive deductive program verification systems. Moreover, the concept implements state-of-the-art user interaction principles. We prototypically implemented our concept as an interactive verification tool for Dafny programs.

Deductive program verification tasks lead to challenging logical reasoning tasks. Recent and ongoing progress of first-order satisfiability solvers (SMT) allow these tasks to be more and more automated. As program verification is an undecidable problem one can always find verification tasks which cannot be verified automatically – in practice, many real-world verification tasks require user guidance such that automatic reasoning engines can find a proof. With rising success of automatic verification engines, programs that require inspection are more and more sophisticated. They are likely to operate on complex data structures or make use of advanced features like concurrency.

This complexity contributes to the fact that guiding the verification tool is a non-trivial, iterative process which, in addition, requires knowledge about internals of the verification tool. First proof attempts for a verification task are likely to result in an unfinished proof either because the code does not satisfy its specification or because the given guidance is not sufficient to allow automation to close the proof. To proceed in the verification process, it is crucial for the user to be able to understand the reason for a failed proof attempt to either remedy the flaw or to provide the right guidance.

The main contribution of this paper is an interaction concept for interactive verification that supports users in understanding unfinished program verification situations and allows them to provide the right guidance to the underlying automatic reasoning engines. It allows users (a) to choose from different representations of the proof state and the kind of interaction style for proof construction according to the user's preferences and the current proof situation, (b) to seamlessly switch between proof state representations when another one seems more informative, (c) to easily recognize relations between information artifacts shown

to the user in the different representations, and (d) to focus on challenging proof subtasks, leaving trivial subtasks to the (automatic) prover.

In state-of-the-art verification systems, the user can interact on different *scopes* of the verification problem: the formal specification, the program code, and the logical representation of the proof state. Depending on which scope a verification system focuses on, different advantages arise: Systems that allow conducting proofs by annotating the source code have the advantage that the user can operate on the same abstraction level as the original source code and needs not understand the logical encoding. Comprehending why a proof attempt failed can be difficult in these verification systems because of the high degree of abstraction. In contrast, systems that allow users to directly manipulate on the logical encoding level may lack the possibility to understand how the encoding relates to the source code. Either way the user is faced with the problem that most state-of-the-art verification systems either offer only one possible representation or do no support the user in understanding the relations between different representations.

Our hypothesis is that one major bottleneck for finding proofs with verification tools is the difficulty of comprehending and exploring unfinished proofs. Without providing methods to support these tasks, advancing proofs gets extremely challenging. Users need means to be able to understand and explore the proof state in order to make an informed decision for the next goal-oriented action.

The remainder of this paper is structured as follows. In Sect. 1, we present preliminaries, followed by a brief summary of the results of our user studies in Sect. 2. Based on these results we present our interaction concept in Sect. 3 and its realization in a prototype in Sect. 4. We present related work in Sect. 5 and conclude the paper with future work in Sect. 6.

1 Setting the Stage: Interactive Program Verification

1.1 Verification Task

To prove properties of programs with a verification system, different proof artifacts interact during the verification process. First the user has to express the desired properties of a software system, the *requirement specification*, using a specification formalism understood by the verification system. Moreover, the user may provide additional specification elements for prover guidance, called *auxiliary specification* [4].

We refer to the program to be verified, together with its specification as *proof input artifacts*. In our case the specification is given in form of annotations of the program. We consider a software system to be composed of different modules and each module may in turn contain different methods and functions. The most basic components of a software system we consider here for verification are thus single methods. We will call a pair of a requirement specification and a software system currently under verification a *concern* [3]. Depending on the context, a requirement specification of a concern may in turn be an auxiliary specification in the larger picture.

Listing 1.1. Running Example Linked List (see the Appendix 1.2 for the full version)

```
1  class List {
2    ghost var seqq: seq<int>; ghost var nodeseqq: seq<Node>;
3    var head: Node;
4    method getAt(pos: int) returns (v: int)
5      requires 0 ≤ pos < |seqq| ∧ Valid()
6      ensures v = seqq[pos]
7    {
8      var idx := 0;
9      var node := head;
10     while(idx < pos)
11       decreases |seqq| - idx;
12       invariant idx ≥ 0 ∧ idx ≤ pos;
13       invariant node = nodeseqq[idx];
14     {
15       node := node.next;
16       idx := idx + 1;
17     }
18     v := node.value;
19   } ...
20 }
```

As running example we use a Dafny implementation of a singly-linked list in Ex. 1.1 (see App. 1.2 for the full version). The program contains two classes `Node` and `List`. A node has a value field and a pointer to the next node. A list object points to the first node (the head) of the list (line 3). The class `List` contains the method `getAt()` (in lines 4–19) and a function `Valid()` (not shown in the Fig. 1.2). For verification purposes we added two *ghost fields* `seqq` and `nodeseqq` (line 2) which have no effect on the program execution. They shadow the list's content. The sequence `seqq` is the sequence of values, and `nodeseqq` is the sequence of the list nodes. The function `Valid()` is a predicate which is true if these sequences correspond to the list's content. Thus, `Valid()` serves as the *object invariant* of the list. Here, we will focus on the method `getAt()` which returns the value at a given index of the list. In the method's implementation we iterate over the list until the given index is reached and return the value of that node. The requirement specification of the method `getAt()` is its pre- and postcondition pair in lines 5–6. To come up with the right set of annotations, the user has to be aware of the different dependencies that exist between and within the system and its specification, e.g., between the requirement specification in one concern and the auxiliary specification in another concern or between the requirement and auxiliary specifications within the same concern.

The proof problem for a concern (e.g., the correctness of a method w.r.t. its specification) can be divided into smaller individual and located proofs units which we will call *proof verification conditions (PVC)*. We distinguish two levels of the proof process: the *global level*, i.e., finding the right formalization for a concern, and the *local level*, i.e., proving single PVCs. One example for PVCs is to generate a proof obligation for each conjunct in the postcondition when considering a method contract. If all PVCs can be proven valid individually, the system is correct w.r.t. its requirement specification. In our running example, one PVC would be to prove that the second part of the loop invariant (line 13) is preserved in each loop iteration. The proof obligation then encodes this PVC logically, as shown in the screenshot in Fig. 4.

1.2 Interaction Styles

Program verification systems can be categorized by their type of user interaction from purely automatic systems, over auto-active systems up to interactive program verification systems (see Fig. 1).

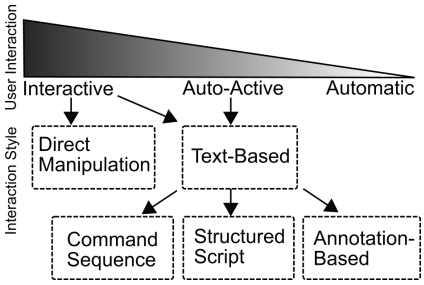

Fig. 1. Categorization of Program Verification Systems and their User Interaction

While user interaction in automatic systems is limited to starting the proof system, in auto-active systems user interaction is limited to adding guiding annotations to the program to be verified. Interactive systems allow users to interact and guide the proof search on the logical representation of the proof problem.

Mainly two interaction styles are supported for proof construction: direct manipulation and text-based interaction. Text-based interaction can furthermore be divided into script-based and command-language-based styles. In the following, we will briefly introduce these different styles.

Direct Manipulation According to Schneiderman [36], the central ideas for *direct manipulation* are that the objects the users are interacting with are visible and the actions applied to these objects are "rapid, reversible and incremental". Users use a pointing device to select objects on the screen which they then can perform actions on. It is crucial that the representation of the elements of the task domain is chosen thoughtfully – otherwise, the representation may be more confusing than helpful. In interactive program verification the objects are the proof obligations, the program with its specification and the proof. Actions on these objects include changing parts of the program or its specification, applying rules to parts of a proof obligation and modifying parts of the proof.

Text-based Interaction The *command-language* or *command line interaction* forms the basic principle for text-based interaction. Users formulate actions as commands followed by a textual representation of the objects that are being manipulated. The action is only executed when users complete their commands and explicitly execute them. For more complex actions *command sequences* can be provided, where the commands are executed in a sequential order.

The *structured script-based interaction* is a more sophisticated form of the textual interaction style. In addition to the commands in the CLI interaction the user may use control flow structures to combine commands to more complex actions. Script-based interaction can be found in different theorem provers, e.g., Isabelle/HOL [33] or the Coq [10] verification system. The kind of proof language differs between the systems. While Isabelle's proof language Isar [38] follows a more textbook style of proof construction, Coq's LTac [34] language is closer to a programming language. Feedback is given by presenting the goal states after executing a proof script. If the proof attempt was not successful, the user needs to inspect the proof to find the cause for any remaining open goal. Advantages of this

interaction style is that experienced users are able to formulate proofs efficiently as long as the interface supports the user in text editing and programming (e.g., with auto-completion or syntax highlighting). A disadvantage of this interaction style is that the principle *recognition rather than recall* [32, 30, 31] for applying the appropriate tactic can be missing, even if auto completion exists. Also, insight into the application of tactics is often missing in these systems.

Auto-active systems, such as OpenJML [14] or Dafny [28] also allow for a kind of text-based interaction purely on *annotation-level*. In this textual interaction style, users interact with the proof system by providing the program and its specification together with further assertions in the source code. These assertions serve as hints for the proof system for the proof search. Feedback is only given in terms of the program and its specification by using visual highlights on program statements alongside with textual messages. Compared to programming the auto-active style is similar to the idea of literate programming [24], where the documentation and the source code are interweaved. One idea behind this interaction style is to hide the proof object and the verification system from the user. Hence, the user does not need to change between different proof artifacts (and thus contexts) when constructing the program and its specification. However, when trying to find the cause for a failed proof attempt detailed insight into the logical representation of the proof problem may be needed [13].

2 User Activities revealed in User Studies

To gain insights into which actions users perform in the program verification process and to identify factors for time-consuming interactions, we conducted qualitative, explorative user studies with intermediate and expert users of interactive verification systems. In particular, we were interested in the proof step granularities, the time-consuming actions and the feedback mechanisms the verification systems offer. We conducted two focus group discussions [6, 5, 20]: one for Isabelle/HOL as a representative for a verification system with script-based interaction and one for the interactive program verification system for Java programs KeY [1], as representative for the direct manipulation interaction style.

Subsequently, we conducted semi-structured interviews together with practical tasks where the participants were asked to perform a proof for a small program with the help of KeY. In this study we have also shown different proof states to the participants and asked for a description of the states [20].[3]

The actions of the participants in the practical tasks have been recorded, and in the analysis phase we have extracted consolidated sequence models [11, 9]. The answers to the interview questions and the voices of the focus group discussions have been evaluated using qualitative content analysis methods [25].

The following are the key results of our user studies influencing our concept.

Different domain elements have to be combined. The evaluation of the user interactions in the practical tasks revealed that one common interaction was

[3] Further details can also be found `https://formal.iti.kit.edu/~grebing/SWC`

that users switched between the different representations of the proof state and the annotated program to relate parts of the specification, respectively program to the formulas in the proof state. One participant in the KeY user study even placed the text editor containing the annotated program next to the proof state with its open goals to find the relation between the artifacts. Switching between contexts can be costly for users: firstly, users needs to (re)gain the orientation in each representation, and, secondly, users need to relate artifacts to each other that refer to the same state but are represented in a different formalism.

Many degrees of freedom. Our evaluation also revealed that there is not *one single* proof process that is followed, rather users take advantage of the many degrees of freedom in user interaction for proof construction and proof comprehension provided by the systems. One example is that some participants used KeY in a first attempt to gain feedback on an initial specification they provided to be able to step-wise adjust it, while others started by using a lot of time to come up with the right specification before using the KeY system for proof construction. For proof construction participants used the full range from performing single proof steps, combinations with sequences of proof steps that were performed automatically by the prover up to fully automatic proof search. For creative tasks like proof construction the degrees of freedom in the interaction are advantageous, however, when developing an interaction concept these degrees of freedom need to be taken into account to not limit expert users.

Alternation of Abstraction and Focusing. As users tried to gain orientation in the proof, we were able to observe an alternation between abstracting from and focusing on specific details of the proof state: users tried to gain an overview over the proof by adjusting the view onto the proof tree, e.g., with hiding features that remove intermediate proof steps from the displayed tree such that only branching nodes are visible. After gaining an overview, users then focused on specific open goals by navigating to them to see in which cases the proof stays open and then focused on sequents and single formulas in the goal. In a subsequent step, users related the inspected elements to the input artifacts.

From our observations in the user studies, we derived the hypothesis that users of interactive program verification systems need both an overview over the system and the bigger picture of the proof task and a way to focus on specific parts of the proof problem. At the same time users need different ways to interact with the proof system for proof comprehension and proof construction.

3 Seamless Interaction Concept

Our interaction concept presented in the following is based on observations from our user studies, on established design principles, on existing interaction functionalities of state-of-the-art verification systems, and on the usability principles for theorem provers by Easthaughffe [17]. Our concept follows the assumption that users must have the ability to interact on a high level of abstraction, the programming language level like in the auto-active approach, but at the same

time must be able to learn about and work on the level of the logical proof encoding, like in the interactive approach.

In our concept, we support users (a) in accessing and combining information from different domains during proof state inspection and proof construction by structuring the proof state into views, (b) in gaining both an overview over the proof state and focusing on details, by providing a view containing proof and system information, together with a mechanism to seamlessly switch between the views. At the same time, our concept retains the idea of many degrees of freedom, e.g., by allowing proof construction in each view and by providing mechanisms to inspect dependencies between the views.

We have seen in our user studies that users need different context information both to comprehend the proof situation and to advance the proof. Which information is needed depends both on the concern the user is focused on and also on whether the user is working on the global or local level.

One challenge of providing the user with the right amount of information are existing dependencies between the different constituents of the system and between the requirement specification and auxiliary specification currently in focus. To support users in proof construction and proof comprehension, we provide multiple *projections* of the proof problem and proof state (shown in Fig. 2), each with their own set of available interactions. The complexity of the proof problem is reduced by concentrating on a subset of dependencies at a time and by providing the required information (resp. hiding unnecessary information) whenever possible during the verification process.

Our concept also supports verification by means for *abstraction* of, as well as *focusing* on details of the proof problem and by breaking down the proof task into smaller *subtasks*. This step-wise focusing should help the user in keeping the overview and dependency information gained from one representation and transferring this knowledge between representations.

One key feature of our concept is that users can choose their preferred way of interacting with the proof system and use the different interaction styles interchangeably. For this we integrate direct manipulation with structured scripts. Actions performed using the direct manipulation style are textually encoded and added to the proof script, adhering to the usability principle of *substituitivity* [16]. The script also serves as a mean to advance the proof by allowing users to textually add proof commands to the script and executing it.

3.1 Projections: Multiple Views onto the Proof Problem

To support users in the verification task we propose multiple projections on the proof problem to be presented to users in different views. These views support users on the level they are currently working on but also take into account that users shift their focus during the proof process. Presenting multiple views of the proof state and allowing meaningful operations on those views are considered as two usability principles for theorem provers. The different views should support users in forming models of the proof task to be able to choose the most appropriate representation for the next goal-directed action [17].

To allow for a seamless change from the global to the local level, users have to be able to inspect dependencies between the components of the verification target, as well as the relations between levels. As a prerequisite, these dependencies need to be made visible to the user.

We assume that the user has a different *focus* on the components on each level and thus different context information may have to be shown to the user. This idea adheres to the usability principle of *anticipation*, which makes the claim that "all information and tools needed for each step in the process" [37] should be provided to the user.

We consider the following views (see Fig. 2) as essential for program verification: ① a view showing the system and proof structure as well as the proof progress overview, ② a view showing the proof input artifacts, i.e, the source code and its annotations, ③ a view that focuses on the logical representation of a single PVC with the possibility to construct a deductive proof for this PVC(④).

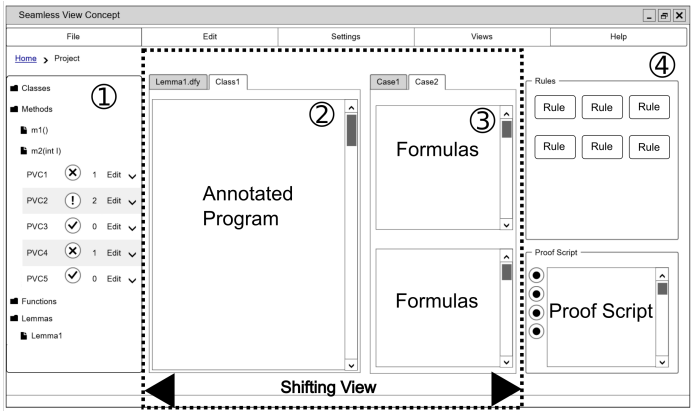

Fig. 2. Abstract concept of our user interface. From left to right: the *System and Proof Overview* (①), the *Source Code View* (②), the *Logical View* (③) and the *Proof Manipulation View* (④). The user can access two adjacent views at the same time (indicated by the dotted rectangle).

Source Code View

In our user study, participants accessed the annotated program in different phases of the process and with different intents and actions: to formulate the requirement and auxiliary annotations at the beginning of the process, as well as for inspection and modification of the annotated program during the proof.

There are different possibilities to display the program and its annotations to the user: both artifacts can be either shown in a combined view or shown separately, but also the amount of annotations that are shown can vary (e.g., showing all annotations as provided by the user, or only certain parts).

Showing both artifacts in a combined view allows users to access information that is closely related and allows for an easy navigation of dependencies between the annotations and the source code locations for inspection. A further advantage is that the maintenance of annotations is simplified such that when changing the source code, the annotations can be directly adjusted as well.

Using two separate views for the program and the annotations would provide more *visual clarity* compared to the sometimes cluttered source code shown in a combined view. Furthermore, a multi-formalism document is avoided, allowing users to stay in one formalism for each view, reducing the user's cognitive load.

The source code view needs to provide actions to capture the following user intents: the comprehension of the proof problem or the proof state, the search for mistakes in the proof input artifacts, the construction of the program and adding (auxiliary) annotations.

As support for inspection, mechanisms have to be provided to inspect the relation between elements of the proof state and the proof input artifacts to support the comprehension tasks. Support to proceed in the verification process should include functionalities similar to common text editing features found in IDEs (e.g., program refactoring) and features for navigation through the source code. Furthermore, to provide immediate feedback to the user, support for writing the annotated program in the form of auto-completion and checking for syntax errors should be provided.

System and Proof Overview

Similarly to the systems to be verified, also concerns have an internal (hierarchical) structure. This structure helps to divide the large task into smaller sub-tasks. It is crucial for the user to gain an overview over the system and its dependencies to build up a mental model about the problem and the proof progress to facilitate decisions about the next actions to take. Providing dependency information also supports the user in keeping track of parts of specifications that influence each other during the proof process.

We devise a view to support the user in gaining a *global* overview over the concern and the proof task. This view allows for activities to progress in the proof process on a more global level, as well as provide means for navigation to the individual proofs for a concern. Typical activities we consider for this view are the *selection* of proof artifacts for inspection, *selection* of proof tasks to work on and *browsing activities* to build a mental model about the problem structure.

The system and verification task structure in this view is shown as a collapsible tree. A further possibility to show call dependencies and dependencies between different specifications needs to be available in this view to support the user in comprehending impacts of modifications to one of involved elements.

In addition to the browsing activities, the user is able to apply general proof search strategies to all or a part of the PVCs in this view without the need to know the internal details of the proof. If the proof search is able to prove the concern there is no need for the user to look into the proof, allowing the user to focus on PVCs that need further attention. We observed such activities during the

user study, where some participants used the automatic proof search strategies of KeY to determine which parts need detailed inspection or interaction.

To gain an overview over the proof progress and its complexity, one of the simplest information that can be displayed is a number of PVCs already closed amongst all PVCs. Furthermore, as the proof problem is divided into smaller tasks, the progress information of the smaller tasks should be presented. To give an overview over the nature of the proof of a PVC we also consider presenting the number of open branches of a proof for a PVC in this overview.

Logical and Proof Construction View

In auto-active verification systems, users need to come up with auxiliary annotations for which often insight into the logical encoding is needed [4, 13]. We propose a view that enables users to gain insight into the logical encoding (PVC) and the deduction steps performed, if they are not able to solve the problem on the program level. Users should be able to focus on one verification task in isolation.

The notation of the logical encoding should be as close as possible to the input artifacts, e.g., names of identifiers such as fields in the program should not be renamed by deduction steps. This would adhere to the principle of *consistency* [8, 16]. In cases where renaming cannot be avoided, a possibility to trace back to the original version of an entity should be provided. In the user study the renaming issue and the retracing of the origin of symbols was criticized by participants.

On the logical view, proof construction takes place on the most detailed level. Here, the user focuses on the individual propositions that are either assumed or need to be proven for the corresponding program state. We argue that formulas with similar origins should be displayed close to each other, e.g., by grouping formulas resulting from the precondition of a program together.

To deal with large logical representations, the user should have possibilities to freely customize the logical view. It should be possible to abstract from formula sets by hiding them or by abbreviating them by names. Furthermore, formulas should be arbitrarily arrangeable (by grouping and sorting).

A natural choice to allow proof construction in this view is via direct manipulation: by pointing and clicking onto terms and formulas the user retrieves possible rule applications for the selected position and is able to apply them. As users should be prevented from performing actions by mistake, it is essential to be able to observe the result of a rule application before actually applying it. This information should be given both on a more abstract level, to give a rough estimate about the rule's effect (e.g., by showing how many branches result from the rule application) and on a more detailed level, to provide an insight into how the proof would evolve if the rule would be applied.

Additionally, to be able to persist the interaction we devise that also a sub view of the logical view should contain the possibility to perform the proof using textual interaction. We will call this sub view *script-view*.

Both interaction styles should be usable interchangeably and in alternation, to allow for different user preferences. Interactions performed using direct manip-

ulation have to automatically extend the proof script. This also allows that the actions performed using direct manipulation are reversible, as users just have to delete the corresponding statement in the script.

Besides the single PVCs, also the current overall proof state needs to be presented to the user to allow for navigation through the proof performed so far.

3.2 Relations between Proof Artifacts

Up to now, we have addressed the difficulties that arise with the system's and problem's structure and dependencies on each view individually. However, dependency relations also exist between the different proof artifacts across the different proposed views. These dependencies are often implicit and users need to keep them in mind while proving or invest resources to search for these dependencies, as was also observable in our user study. One building block of our concept is thus to make these *hidden dependencies* [12] visible to the user.

Concerning the verification target, each implementation in the *source code view* has a representation in the *system overview* that shows its location in the system hierarchy. This information about call contexts of subsystems can be used by the user when formulating requirement specifications of the subsystems.

For an overview over the proof task, not only the proof progress is important but also the information about dependencies between the proof obligations, the proof input artifacts, the PVCs, as well as lemmas. This dependency information is necessary to get an overview about what can be (re)used during the proofs.

The PVCs in the *system and proof overview* have a relation to the properties of a subsystem, i.e., a PVC has a relation to a path through the program and the annotations along this path. A PVC also has a direct relation to the *logical view* where the logical representation is a formalization of the PVC. The two relations together induce the relation between the individual formulas in the logical representation and the annotated program. Formulas in the *logical view* can have their origin from statements in the program or the annotations or from specific rule applications, such as the cut-rule.

Between the Script View and the Logical View relations exist as well. Each statement in the script corresponds to a proof state of a PVC, which evolves by applying rules. Altogether, there exists a relation between a rule application, a statement in the proof script and the logical representation of the proof state, e.g., a node in a proof tree. It is crucial for user support to make these relations visible. As the number of relations may become large, it is advisable to display this information on user request.

Furthermore, we devise to support "zooming-in" from the abstract overview to the detailed representation, which was also observable in the user study. This especially means to support the user in switching between the different views and keeping track of the dependencies that exist in one view but also the dependencies between the views. This can be supported by positioning views sequentially, i.e., starting with the most abstract view (proof and project overview) down to the detailed logical view, and thus restricting the work-flow to the "zooming-in" process. This restriction ensures that important relations are always visible and

can be inspected. The proposed arrangement of views supports users in carrying over relevant information from one view to another in two ways: firstly by placing contextually close views next to each other and secondly by always keeping the last view present when switching to a new one. Further support could be to allow the user to trace the origin of element across the different views.

4 Realization of the Concept

The presented concept is prototypically implemented as a tool called DIVE (Dafny Interactive Verification Environment)[4] for programs written in Dafny.

The main window of DIVE is split up into 4 different views (as also shown in the concept in Fig. 2): the *System and Proof Overview*, the *Source Code View*, *Logical View* and the *Proof Manipulation View*. Each view corresponds to a different projection of the same proof state. In the course of this section we will present each view and their relations in more detail.

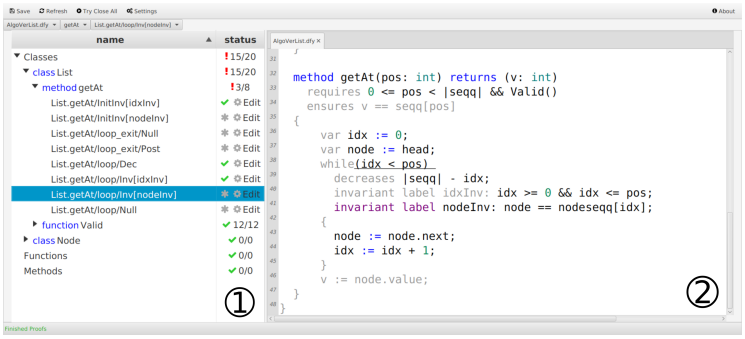

Fig. 3. The *Project and Proof Overview* (①) and the *Source Code View* (②) of DIVE adjacent to each other. This view is the first view users see when loading a project.

The Source Code View (② in Fig. 3) shows the proof input artifacts. In DIVE this view is a tabbed window with standard text editor features. We included syntax highlighting for the annotated source code as a visual user guidance, as well as syntax checking with a visual highlighting of the location containing a syntax error.

As soon as users edit any proof input artifact in the source code view, the contents of *all* other views are disabled as the information presented in these views is outdated. This decision has been made to maintain a consistent state of all proof artifacts. These views are re-enabled as soon as the user saves the changes. We chose to refresh the project state only if the user explicitly requests this refresh to avoid unnecessary reload attempts during typing.

[4] Available at `https://formal.iti.kit.edu/~grebing/dive/vstte19/`.

The System and Proof Overview (see ① in Fig. 3) enables users to quickly get an overview of the overall progress made so far. It allows users to see how many PVCs are still open/closed, as well as the structure of the overall proof and the navigation to the logical representation of a PVC.

We show the hierarchical structure of the components of the verification target in a tree view. Besides these (sub)systems of the verification target, we also include lemmas written by the user in this tree view. The tree structure is collapsible and expandable to allow the user to either gain a more abstract overview or to be able to focus on single components of the system.

To support the user in relating a PVC to the annotated program we have chosen to use identifiers for each PVC that contain information of the symbolic execution path they represent, followed by an identifier that captures the PVCs purpose. Also, users can label annotations to use this label in the PVC's name.

To allow for an overview over the proof progress, for proof inspection and proof construction on the system level, we included means for interacting with the PVCs in the tree structure. For each PVC, an indicator of its proof status is shown, the number of proof branches in a proof and a possibility to select the PVC for proof construction on the local level. Users can start proof construction in this view by selecting a (sub)system or PVC and applying a general SMT solver or script. When selecting a system, the corresponding implementation is shown in the *Source Code View*. Expanding a system results in displaying the hierarchical structure of the system down to all PVCs. Users can retrieve a presentation of the PVCs for a system and a PVC's relation to the program control structure by selecting a single PVC. Upon selection, in the *Source Code View* the relevant part of the source code is put into focus together with a highlight of those statements that correspond to the symbolic execution path for the selected PVC.

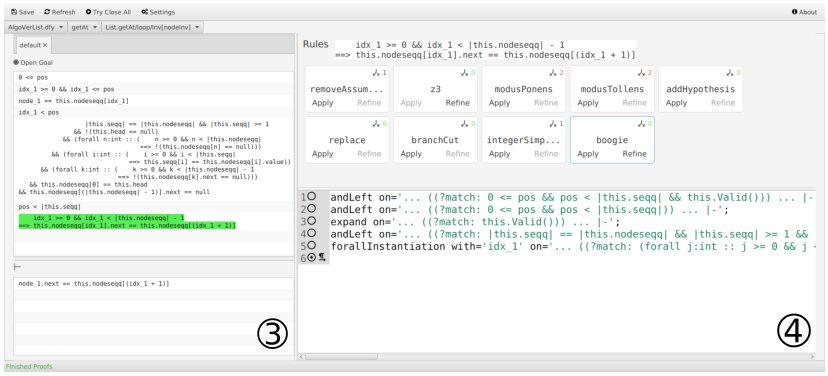

Fig. 4. The *Logical View* (③) and the *Proof Manipulation View* (④) of DIVE. These views adjoin to the right of the views in Fig. 3

The Logical View (see ③ in Fig. 4) shows the logical representation of a PVC. We decided to use a sequent calculus as underlying logic so PVCs are shown

as sequents. This view consists of two list views which represent the antecedent and succedent of a sequent. Formulas in the antecedent may be assumed for the selected PVC (e.g., preconditions) whereas to discharge the PVC at least one of the formulas of the succedent has to be shown (e.g., assertions or post-conditions).

During proof construction branching rules may be applied to a sequent such that the PVC has more than one branch. In this case the logic view is tabbed to allow the user to navigate between different branches. When applying rules the effect of the last rule application is also shown as graphical hints in the logic view so that users are able to see the consequences of actions.

Additionally users are able to request the origin of (sub-)formulas displayed in the logic view. These origins may stem from the source code (e.g. a pre- or postcondition) in which case the responsible part in the source code is highlighted or may have resulted from rule applications in which case the relevant part of the script in the *Proof Manipulation View* is highlighted. This allows users to relate the elements in the projections to each other and build up as well as keep a mental model consistent with the proof state.

The Proof Manipulation View (see ④ in Fig. 4) allows proof construction using the two interaction styles direct manipulation and text-based interchangeably. Each manipulation technique is modeled in DIVE with one subview which supports users in manipulating the proof in their desired way.

The *Script View* is a text field where the user can manually extend the proof script. This view also allows users to navigate through different proof states of one PVC by placing the cursor in the script on a proof command or clicking on *checkpoint markers* next to the script. The script is only executed on request to prevent repeated executions when no full command has been provided.

The *Rule View* allows users to apply rules using direct manipulation. Users select formulas (or sub-formulas) in the *Logical View* and a list of applicable rules appear in the rule view. Each rule is displayed as panel with the name of the rule, the number of branches that would be opened if this rule is applied and the possibility to apply this rule to the selected formula. Additionally if a panel is selected the effects of applying this rule are shown in the *Logical View*. All these features support users in gaining information about the effect of applying a certain rule and thus allows users to make an informed decision which rule to apply in which situation. Applying a rule automatically updates the script and the *Logical View* to represent the new proof state.

An Exemplary Walk-Through. As an example for a common interaction with DIVE, consider that the user loads the Dafny file containing the implementation of our running example. This results in the system showing the content of the file in the Source Code View, at the same time, the PVCs are generated and shown in the System and Proof Overview. The user is now able to modify the source code and the annotations. In a next step the proof process can be started by selecting the PVCs in the system overview and applying lightweight solvers to them. The result of applying the lightweight solvers to all generated PVCs of the class `List` is depicted in Fig. 3. In our example, some PVCs stay open (indicated by the red symbol). The user may now focus on those PVCs in the subsequent

steps. For example, the second invariant remains open. The proof obligation (③ in Fig. 4) states that at each iteration of the loop the local variable `node` is the `i`-th element of the ghost sequence which mirrors the list (thus stating that at each iteration `node` is the `i`-th element of the list).

To close the proof the correct instantiation of a quantifier in the antecedent is missing. The user can select this quantified formula in the logical view and apply the rule "forall-instantiation" in the proof manipulation view. Now the SMT solver is able to close this part of the proof and the user can focus on the next PVC. Alternatively, the user could have provided a script to close the proof.

Another scenario is to inspect the sequent and recognize a contradiction due to which the proof could not be closed. In this case the user would like to determine the origin of the contradiction. To do so it is possible to request the origins of the involved formulas. Thus, users are able to see whether the contradiction is due to an error in the source code or is introduced by a rule application (e.g., a cut).

5 Related Work

One of the key features of our concept is to present different projections of the proof state in adjacent views and allowing a seamless switch between these views. Systems such as KIV [22, 2], Why3 [18], KeY, KeYmaeraX [29, 35, 19], Coq [10] or Isabelle [33] provide views with different purposes to the user. These systems show the current proof state in a view that is different from the view containing the whole proof structure. Each view should support users in specific tasks, thus each view contains actions and features that are necessary for the task. For example, in KIV or KeY the current goal view allows to retrieve the applicable rules and context-sensitive support for rule application, or the script view in Isabelle or Coq allows to extend the proof script with support for text-editing.

In KeYmaeraX, the actions accessible in the view containing *deduction paths* allow for a step-wise focusing on the different steps of a proof branch, by expanding details of the path in a tab, starting at the open goal. With progressing proof, this view may become cluttered as a deduction path may contain a large number of proof steps. Structuring of proof goals is also possible in KeYmaeraX: users can change the sequent view by hiding formulas.

The System and Proof Overview and the hierarchical structuring of concerns was inspired by the Why3 platform, which splits concerns into single PVCs and presents them hierarchically. Different means to discharge the proof verification conditions can be used, like SMT solvers or interactive verification using Coq. Concerning providing information about relations between the elements of different views, Why3 highlights relevant statements in the annotated source code to relate PVCs to the proof input artifacts. Relations between the formula in Coq and the proof input artifacts get lost during transformation. Compared to our concept, the user needs to switch between different systems and user interfaces, which requires the user to gain orientation and to perform a context switch.

Also KeY offers support for inspecting relations between proof artifacts. In response to our user study, KeY was improved to show the relation between the

annotated program and the proof state in a new window, where all symbolically executed statements are highlighted. However, changing the annotated program requires an external editor and a restart of the verification process.

In our concept we integrate interactive with auto-active verification as found in tools like VCC [15] and Dafny [27]. These tools give feedback on the program level, and provide integration into an IDE. Users may retrieve information about violated assertions at specific program locations and inspect the values of variables for a program path. Concerning the direct manipulation style, mainly KeY, KeYmaeraX and KIV were role-models for our concept.

Similarly to the concept presented here, in previous work we integrated direct manipulation and script-based interaction for KeY [7, 21]. Other systems that allow for different combinations of interaction styles include KeYmaeraX which also allows for both direct manipulation and text-based interaction interchangeably.

6 Conclusion and Future Work

We have presented a concept for a user interface for a seamless interactive program verification process, which is based on results from user studies, as well as general usability principles and principles for theorem provers. One of the goals of the concept is to support users in step-wise focusing on the different parts of the proof artifacts for inspection and proof construction, as well as the possibility to seamlessly switch to more abstract presentations if necessary. At the same time users are supported in the inspection of relations between the proof artifacts. The concept integrates auto-active and interactive program verification and allows for an alternating use of text-based interaction and direct manipulation.

We also presented DIVE as a prototypical implementation of our concept, which structures parts of the proof artifacts in different views. Views are arranged in a way to support users in step-wise focusing on more detailed parts of the proof artifacts. Only two adjacent views are shown to the user at the same time. The goal of the arrangement is to support the users in carrying over information about relations and dependencies between different proof artifacts from one view to the other, by trying to keep the cognitive load low. Additionally, users may invoke mechanisms for inspecting relations and dependencies.

Future work includes integrating missing features like the a view showing the call or usage dependencies, or proof exploration techniques for the logical view. Furthermore, following the user-centered design process, case studies, as well as user studies have to be performed to evaluate the effectiveness of the interaction concepts and the user experience. The user studies should also evaluate whether our proposed work-flow may need to be adapted for expert users.

To evaluate whether the integration of the interactions styles is improving user support, we suggest performing a comparative evaluation using the prototype. Groups of participants should perform comparable verification tasks with either using only a single interaction style or with a combination of styles. Both task completion time should be measured, as well as the user experience using standardized questionnaires, such as UEQ [26] or SUMI [23].

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. Michael Balser, Wolfgang Reif, Gerhard Schellhorn, Kurt Stenzel & Andreas Thums (2000): *Formal System Development with KIV*. In: *Proceedings of the Third Internationsl Conference on Fundamental Approaches to Software Engineering: Held As Part of the European Joint Conferences on the Theory and Practice of Software, ETAPS 2000*, FASE '00, Springer-Verlag, Berlin, Heidelberg, pp. 363–366. Available at http://dl.acm.org/citation.cfm?id=645368.650817.
3. Bernhard Beckert, Thorsten Bormer & Daniel Grahl (2016): *Deductive Verification of Legacy Code*. In Tiziana Margaria & Bernhard Steffen, editors: *7th International Symposium on Leveraging Applications of Formal Methods, Verification and Validation (ISoLA 2016)*, *LNCS 9952* I: Foundational Techniques, Springer, pp. 749–765, doi:10.1007/978-3-319-47166-2_53.
4. Bernhard Beckert, Thorsten Bormer & Vladimir Klebanov (2011): *Improving the Usability of Specification Languages and Methods for Annotation-based Verification*. In Bernhard Aichernig, Frank S. de Boer & Marcello Bonsangue, editors: *9th International Symposium on Formal Methods for Components and Objects (FMCO 2010), State-of-the-Art Survey*, *LNCS* 6957, Springer.
5. Bernhard Beckert, Sarah Grebing & Florian Böhl (2014): *How to Put Usability into Focus: Using Focus Groups to Evaluate the Usability of Interactive Theorem Provers*. In Christoph Benzmüller & Bruno Woltzenlogel Paleo, editors: *Eleventh Workshop on User Interfaces for Theorem Provers (UITP 2014)*, *EPTCS* 167, pp. 4–13.
6. Bernhard Beckert, Sarah Grebing & Florian Böhl (2014): *A Usability Evaluation of Interactive Theorem Provers Using Focus Groups*. In Carlos Canal & Akram Idani, editors: *12th International Conference on Software Engineering and Formal Methods (SEFM 2014) – Collocated Workshops: Human-Oriented Formal Methods (HOFM 2014)*, *Lecture Notes in Computer Science* 8938, Springer, pp. 3–19, doi:10.1007/978-3-319-15201-1_1.
7. 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.
8. Catherine Plaisant Ben Shneiderman (2005): *Designing the User Interface: Strategies for Effective Human-Computer Interaction*. Pearson.
9. D. Benyon (2010): *Designing Interactive Systems: A Comprehensive Guide to HCI and Interaction Design*. Addison Wesley.
10. 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.
11. Hugh Beyer & Karen Holtzblatt (1998): *Contextual Design: Defining Customer-centered Systems*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.
12. Alan Blackwell & Thomas R. Green (2007): *A Cognitive Dimensions Questionnaire (V. 5.1.1)*. www.cl.cam.ac.uk/~afb21/CognitiveDimensions/CDquestionnaire.pdf.

13. Thorsten Bormer (2014): *Advancing Deductive Program-Level Verification for Real-World Application: Lessons Learned from an Industrial Case Study*. Ph.D. thesis, doi:10.5445/IR/1000049792.
14. David R. Cok (2011): *OpenJML: JML for Java 7 by Extending OpenJDK*. In Mihaela Bobaru, Klaus Havelund, Gerard J. Holzmann & Rajeev Joshi, editors: *NASA Formal Methods*, Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 472–479.
15. M. Dahlweid, M. Moskal, T. Santen, S. Tobies & W. Schulte (2009): *VCC: Contract-based modular verification of concurrent C*. In: *2009 31st International Conference on Software Engineering – Companion Volume*, pp. 429–430, doi:10.1109/ICSE-COMPANION.2009.5071046.
16. Alan Dix, Janet Finlay, Gregory Abowd & Russell Beale (2004): *Human-computer Interaction*. Prentice-Hall, Inc.
17. Katherine A. Easthaughffe (1998): *Support for Interactive Theorem Proving: Some Design Principles and Their Application*. *User Interfaces for Theorem Provers (UITP 1998)*.
18. Jean-Christophe Filliâtre & Andrei Paskevich (2013): *Why3 – Where Programs Meet Provers*. In: *ESOP'13 22nd European Symposium on Programming*, *LNCS* 7792, Springer, Rome, Italy. Available at https://hal.inria.fr/hal-00789533.
19. Nathan Fulton, Stefan Mitsch, Jan-David Quesel, Marcus Völp & André Platzer (2015): *KeYmaera X: An Axiomatic Tactical Theorem Prover for Hybrid Systems*. In Amy P. Felty & Aart Middeldorp, editors: *CADE*, *LNCS* 9195, Springer, pp. 527–538, doi:10.1007/978-3-319-21401-6_36.
20. Sarah Grebing (2019): *User Interaction in Interactive Deductive Program Verification*. Ph.D. thesis. Defended Feb. 2019, KIT, To Appear.
21. Sarah Grebing, An Thuy Tien Luong & Alexander Weigl (2018): *Adding Text-Based Interaction to a Direct-Manipulation Interface for Program Verification – Lessons Learned*. In Mateja Jamnik & Christoph LÃijth, editors: *13th International Workshop on User Interfaces for Theorem Provers (UITP 2018)*. To appear.
22. Dominik Haneberg, Simon Bäumler, Michael Balser, Holger Grandy, Frank Ortmeier, Wolfgang Reif, Gerhard Schellhorn, Jonathan Schmitt & Kurt Stenzel (2005): *The user interface of the KIV verification system – a system description*. In: *Proceedings of the User Interfaces for Theorem Provers Workshop (UITP 2005)*.
23. J. Kirakowski: *The Use of Questionnaire Methods for Usability Assessment*. Available at http://sumi.uxp.ie/about/sumipapp.html.
24. D. E. Knuth (1984): *Literate Programming*. *The Computer Journal* 27(2), pp. 97–111, doi:10.1093/comjnl/27.2.97. Available at https://doi.org/10.1093/comjnl/27.2.97.
25. U. Kuckartz (2014): *Qualitative Inhaltsanalyse. Methoden, Praxis, Computerunterstützung*. Grundlagentexte Methoden, Beltz Juventa.
26. Bettina Laugwitz, Theo Held & Martin Schrepp (2008): *Construction and evaluation of a user experience questionnaire*. In: *Symposium of the Austrian HCI and Usability Engineering Group*, Springer, pp. 63–76.
27. K. Rustan M. Leino & Valentin Wüstholz (2014): *The Dafny Integrated Development Environment*. In Catherine Dubois, Dimitra Giannakopoulou & Dominique Méry, editors: *Proceedings 1st Workshop on Formal Integrated Development Environment, F-IDE 2014, Grenoble, France, April 6, 2014.*, *EPTCS* 149, pp. 3–15, doi:10.4204/EPTCS.149.2. Available at https://doi.org/10.4204/EPTCS.149.
28. Rustan Leino (2010): *Dafny: An Automatic Program Verifier For Functional Correctness*. Microsoft Research. Available at https://www.microsoft.com/en-us/research/publication/dafny-automatic-program-verifier-functional-correctness/.

29. Stefan Mitsch & André Platzer (2017): *The KeYmaera X Proof IDE - Concepts on Usability in Hybrid Systems Theorem Proving*. In Catherine Dubois, Paolo Masci & Dominique Méry, editors: *Proceedings of the Third Workshop on Formal Integrated Development Environment, Limassol, Cyprus, November 8, 2016*, *Electronic Proceedings in Theoretical Computer Science* 240, Open Publishing Association, pp. 67–81, doi:10.4204/EPTCS.240.5.
30. Rolf Molich & Jakob Nielsen (1990): *Improving a Human-computer Dialogue. Commun. ACM* 33(3), pp. 338–348, doi:10.1145/77481.77486.
31. Jacob Nielsen (1995): *10 Usability Heuristics for User Interface Design*. Available at nngroup.com/articles/ten-usability-heuristics.
32. 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.
33. Tobias Nipkow, Lawrence C. Paulson & Markus Wenzel (2002): *Isabelle/HOL: A Proof Assistant for Higher-Order Logic*. LNCS 2283, Springer.
34. C. Paulin-Mohring (2012): *Tools for Practical Software Verification, LASER 2011 summerschool, Revised Tutorial Lectures*, chapter Introduction to the Coq proof-assistant for practical software verification, pp. 45–95. *Lecture Notes in Computer Science* 7682, Springer-Verlag.
35. André Platzer (2018): *Logical Foundations of Cyber-Physical Systems*. Springer, Switzerland. Available at http://www.springer.com/978-3-319-63587-3.
36. Ben Schneiderman (1983): *Direct Manipulation. A Step Beyond Programming Languages. IEEE Transactions on Computers* 16(8), p. pp. 57âĂŞ69.
37. Bruce Tognazzini (1987-2014): *First Principles of Interaction Design (Revised and Expanded)*. Available at https://asktog.com/atc/principles-of-interaction-design/.
38. Markus Wenzel (1999): *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, Springer-Verlag, London, UK, UK, pp. 167–184.

A List Example

Listing 1.2. Running Example: Linked List

```
class Node {
  var value: int
  var next: Node

  method Init(value: int, next : Node)
    modifies this;
  {
    this.value := value;
    this.next := next;
  }
}

class List {
  ghost var seqq: seq<int>;
  ghost var nodeseqq: seq<Node>;

  var head: Node;

  function Valid() : bool
  {
    |seqq| = |nodeseqq| ∧
    (∀ i • i ≥ 0 ∧ i < |seqq| ⟹ seqq[i] = nodeseqq[i].value) ∧
    nodeseqq[0] = head ∧
    nodeseqq[(|nodeseqq| - 1)].next = null ∧
    (∀ j • j ≥ 0 ∧ j < |nodeseqq| - 1 ⟹ nodeseqq[j].next = nodeseqq[j + 1]) ∧
    (∀ k • k ≥ 0 ∧ k < |nodeseqq| - 1 ⟹ nodeseqq[k].next ≠ null) ∧
    (∀ n • n ≥ 0 ∧ n < |nodeseqq| ⟹ nodeseqq[n] ≠ null) ∧
    head ≠ null
  }

  method getAt(pos: int) returns (v: int)
    requires 0 ≤ pos < |seqq| ∧ Valid()
    ensures v = seqq[pos]
  {
    var idx := 0;
    var node := head;
    while(idx < pos)
      decreases |seqq| - idx;
      invariant idx ≥ 0 ∧ idx ≤ pos;
      invariant node = nodeseqq[idx];
    {
      node := node.next;
      idx := idx + 1;
    }
    v := node.value;
  }
}
```