Lightweight Interactive Proving inside an Automatic Program Verifier*

Sylvain Dailler[1,2,3], Claude Marché[1,2], and Yannick Moy[3]

[1] Inria, Université Paris-Saclay, F-91120 Palaiseau
[2] LRI, CNRS & Univ. Paris-Sud, F-91405 Orsay
[3] AdaCore, F-75009 Paris

Abstract. Among formal methods, the deductive verification approach allows establishing the strongest possible formal guarantees on critical software. The downside is the cost in terms of human effort required to design adequate formal specifications and to successfully discharge the required proof obligations. To popularize deductive verification in an industrial software development environment, it is essential to provide means to progressively transition from simple and automated approaches to deductive verification. The SPARK environment, for development of critical software written in Ada, goes towards this goal by providing automated tools for formally proving that some code fulfills the requirements expressed in Ada contracts.

In a program verifier that makes use of automatic provers to discharge the proof obligations, a need for some additional user interaction with proof tasks shows up: either to help analyzing the reason of a proof failure or, ultimately, to discharge the verification conditions that are out-of-reach of state-of-the-art automatic provers. Adding interactive proof features in SPARK appears to be complicated by the fact that the proof toolchain makes use of the independent, intermediate verification tool Why3, which is generic enough to accept multiple front-ends for different input languages. This paper reports on our approach to extend Why3 with interactive proof features and also with a generic client-server infrastructure allowing integration of proof interaction into an external, front-end graphical user interface such as the one of SPARK.

1 Introduction

For the development of software with high safety and security requirements, *deductive program verification* is an approach that provides access to the highest levels of guarantees. The functional requirements are expressed using formal specification languages. The conformance of the code with such specifications can be established in a fairly automated setting using automatic program verifiers available nowadays (such as Dafny, F*, KeY, KIV, OpenJML, Verifast, Viper, Why3, etc.). Such verification tools typically proceed by generating *verification conditions* (VC for short): mathematical formulas that need to be proven valid. Such VCs are typically discharged using automated solvers, such as the SMT solvers reasoning on *Satisfiability Modulo Theories*.

* Work partly supported by the Joint Laboratory ProofInUse (ANR-13-LAB3-0007, https://www.adacore.com/proofinuse) of the French national research organization

A major issue preventing the diffusion of deductive verification in industrial applications is the cost in terms of human effort required to design adequate formal specifications and to successfully discharge the VCs. To leverage this issue, it is important, when no SMT solver is able to solve a given VC, to provide the user with means to investigate the proof failure: is it because the code needs to be fixed, is it because the program is not sufficiently annotated (e.g. a missing loop invariant), or is it because the VC is too complex to be discharged by automatic provers (e.g. an induction is needed). Among the possible means is the generation of counterexamples [9]. Such a feature appears to be useful in particular to fix trivial mistakes, but for complex cases, its limitations quickly show up: because of the intrinsic incompleteness of back-end solvers, or more pragmatically because solvers proof search is typically interrupted after a given time limit is reached, the counterexamples may be spurious or absent [9]. Moreover, there is no easy mean to distinguish a true bug from insufficiently detailed specifications (although there is on-going research in that direction [15]).

This paper presents an approach that we designed in the context of the SPARK verifier for industrial development of safety-critical Ada code [7,12]. The goal is to provide simplified interactions between the user and the failing VC, so as to investigate a proof task without the need to rely on an external interactive prover. A specificity of SPARK is that the underlying toolchain from the given input Ada program to the VCs makes use of the external intermediate language Why3 [6] that itself provides access to many different automated provers (mainly Alt-Ergo, CVC4 and Z3) but also general purpose interactive theorem provers (Coq, Isabelle/HOL, PVS). Indeed, an extreme mean to investigate a proof failure is to launch an interactive theorem prover on the failing VC and to start writing a manual proof. Such a process shows up useful, mainly because writing the detailed steps, in which the user believes the proof should work, typically helps to discover missing elements in the specifications, and in such cases fixing the annotations could finally help the SMT solvers to automatically discharge the VC. Also, an ultimate situation is to finish the proof completely using the underlying interactive prover [3]. The main drawback in this process is that users should be able to use the general-purpose back-end interactive prover, forcing them to learn a completely different environment, using its own syntax for formulas, and its specific proof tactics to discharge proof tasks. Another drawback is that once the user has switched to an external proof assistant to proceed with a proof, then there is no easy mean to get back to the common environment offered by Why3 to call other automatic provers on the sub-goals generated by interactive proof tactics.

1.1 Related Work

The need for user interaction in the context of industrial use of deductive verification is not new, this issue was identified and taken into account early in the design of industrial tools. The KIV environment, used in large realistic case studies in academia and industry for more than 20 years, considered very early the importance of combining automated and interactive proving [2]. Other early tools that provide some form of interactive proving are the industrial tools, largely used in railway industry, Atelier B [1] and Rodin [13]. The KeY environment, somehow a successor of KIV, is designed to build proofs interactively, with the possibility to call efficient automated provers for solving

some leaves of the proof [10,11]. In the context of general purpose proof assistants, the need for adding automation to their general interactive theorem proving process is evident, for example in the environments ACL2, HOL Light, and more recently in Isabelle/HOL where the so-called Sledgehammer subtool is able to finish proofs using external SMT solvers [4].

1.2 Common Issues in Automated Program Verification

Interestingly, our analysis of the common situations where fully automatic provers fail, and switching to proof interaction is needed, is very similar to the analysis made in previous work mentioned above. Here are the main identified cases:

- quantifier instantiation: a proof should be done by an adequate instantiation of a universally quantified hypothesis, that the automatic provers cannot discover. Providing the instantiation by hand helps. Similarly, for proving an existential goal, automated provers typically cannot guess the witness. This witness should be given by hand.
- reasoning by cases: an explicit case reasoning can help the automatic provers.
- controlling the context and the strategy of proof search: a prover would sometimes use most of its available time to try to solve the problem using a direction of thinking while a simple solution exists. Reducing the context manually can help.
- inductive reasoning: some properties require reasoning by induction over an integer, an algebraic datatype, or on an inductively defined predicate ; such reasoning steps are out-of-reach of common automated solvers, whereas applying an induction rule by hand usually results in sub-goals that can be automatically discharged.
- non-linear integer arithmetic: it is typically hard for automatic provers, but a few manual proof steps can make such a proof feasible. Similarly, floating-point arithmetic is also very hard for automatic provers.

1.3 Contributions and Overview of the Paper

Our goals are shared with the above-mentioned related work. However, in the context of SPARK, there is an additional issue that does not show up in previous work. Indeed, in the previous work mentioned above, the language of formulas and proof tasks (e.g. proof sequents) is directly the language in which the user writes her input problem: for example in the context of the B method, the logic language is B set theory in which the code of the B machines is written; in KeY, the underlying Dynamic Logic incorporates pieces of Java code to verify. In the context of SPARK, we have the additional issue that a proof task generated by the Why3 VC generator is written in a language that is very different from the Ada input language.

Our approach proceeds in the following steps. In Section 2, we present what we added to the Why3 intermediate tool to provide interactive proving features. Section 3 presents the use of interactive proof from SPARK, inside GNAT Programming Studio, the Ada interface development environment. Section 4 concludes and discusses remaining future work.

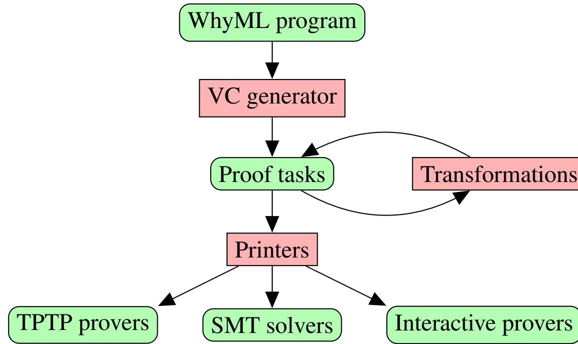

Fig. 1. Why3 general architecture

2 Adding Interactive Proving in Why3

Figure 1 presents a general overview of Why3's core architecture. The input files contain code with formal specifications, written in the dedicated language WhyML [8], which essentially consists in a set of functions or procedures annotated with contracts (pre- and post-conditions, loop invariants, etc.). The VC generator produces, from such a file, a set of *proof tasks*. A proof task, that we can denote as $\Gamma \vdash G$, consists in a set Γ of logical declarations of types, function symbols, predicates, and hypotheses, and finally the logical formula G representing the goal to prove. The soundness property of the VC generator expresses that if all generated proof tasks are valid logical statements, then the input program is safe: no runtime error can arise and formal contracts are satisfied.

Consider the following toy example of a function written in WhyML.

```
let f (a:array int) (x:int) : int
  requires { a.length ≥ 1000 }
  requires { 0 ≤ x ≤ 10 }
  requires { forall i. 0 ≤ 4*i+1 < a.length → a[4*i+1] ≥ 0 }
  ensures { a[result] ≥ 0 }
= let y = 2*x+1 in y*y
```

The function `f` takes as parameters an array `a` and an integer `x`. The first two pre-conditions express two simple requirements on the size of array `a` and an interval of possible values for `x`. The third pre-condition is a bit more complex, it expresses that for indexes that are a multiple of 4 plus 1, the values stored in `a` is non-negative. The function `f` returns an integer, denoted as the keyword `result` in the post-condition, with a post-condition expressing that the value at the index returned is also non-negative. The code simply returns the square of $2x+1$. For such a code, Why3 generates as the VC the formula

```
forall a:array int, x:int.
 length a ≥ 1000 ∧ (0 ≤ x ∧ x ≤ 10) ∧
 (forall i:int.
   0 ≤ ((4 * i) + 1) ∧ ((4 * i) + 1) < length a → a[(4 * i) + 1] ≥ 0) →
```

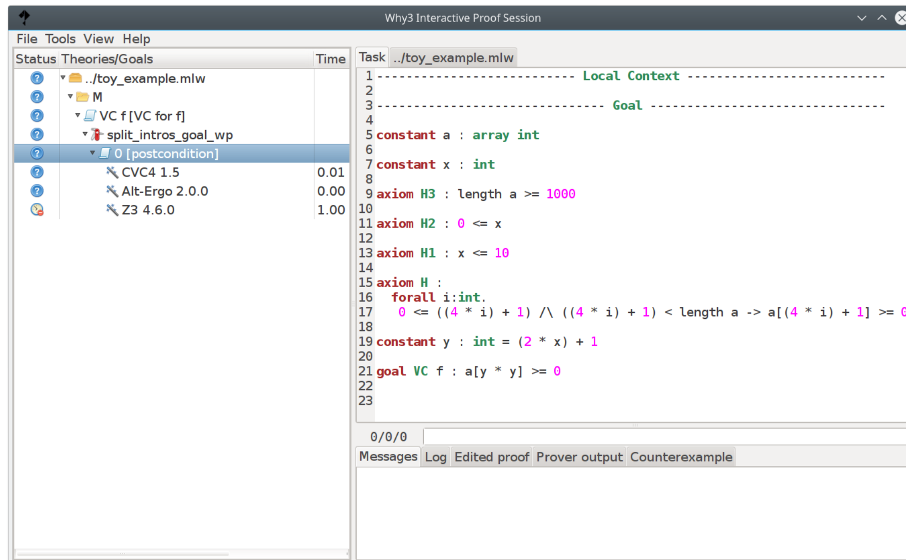

Fig. 2. Failed proof attempts shown in Why3IDE

```
(let y = (2 * x) + 1 in a[y * y] ≥ 0)
```

As one may guess such a formula can quickly become unreadable when code size grows, and is hardly suitable for human inspection.

2.1 Proof Tasks and Transformations

The core of Why3 comes as a software library, written in the OCaml language and with a documented API, that proposes in particular data-types for terms, formulas and proof tasks, and a large collection of operations to operate on them. A central notion is the notion of *transformation*: an OCaml function that takes a proof task as input and returns a set of proof tasks. All implemented transformations are expected to be sound, in the sense that if all the resulting proof tasks are valid, then the original task is valid too. A simple example of such a transformation is *splitting*, which basically transforms a task of the form $\Gamma \vdash G_1 \wedge G_2 \wedge \cdots \wedge G_k$ into the set of tasks $\Gamma \vdash G_i$ for $1 \leq i \leq k$. The VC generator is designed so as to produce a single proof task for each procedure or function of the input code, as in the example above. To ease the visibility and understanding of the resulting formula, a generalized splitting transformation is typically applied, so as to decompose such a VC into a set of simpler VCs for specific properties to check, *e.g.* checking if an array index is in bounds, checking the preservation of a loop invariant, checking the pre-condition of a sub-procedure called, etc.

Figure 2 presents a screenshot of Why3's graphical interface (Why3IDE) when given our toy program as input. The left part of that window is the *proof task tree*. The current selected line is for the proof task after applying the transformation named

`split_intros_goal_wp` which implements the generalized splitting transformation mentioned above.

Indeed the role of transformations is two-fold. The first role is to somehow simplify the given task (such as the splitting above). In such a case a transformation is applied on user's request inside the graphical interface. The second role is to preprocess a task before sending it to an external prover: typically an external prover does not support all features of Why3's logic, such as polymorphic types, algebraic data types and the corresponding pattern-matching, recursive or inductive definition of predicates, etc. Hence, when the user wants to invoke an external prover on a given task, Why3 transparently applies some transformations to make the proof task fit into the logic of the target prover, before using an appropriate *printer* suitable for the back-end prover input syntax (e.g. SMT-LIB). On Figure 2 it can be seen that the provers Alt-Ergo, CVC4 and Z3 were invoked but none of them were able to discharge the expected post-condition. It is likely that the combination of non-linear arithmetic and the need for finding an appropriate instantiation for the hypothesis `H` is the reason why they fail. Mixing quantifiers and arithmetic makes very difficult goals to prove for all categories of automatic provers: SMT solvers quantifier handling is based on triggers, that do not interact well with arithmetic, while TPTP provers have a more powerful handling of quantifiers but do not support arithmetic. This corresponds to the limitation of automatic provers that we called *quantifier instantiation* in Section 1.2.

On top of the core architecture of Figure 1, Why3 features two major components: the *proof session* manager and the graphical interface. Adding support for interactive proving in this global architecture requires extensions that we detail in the subsections below: extensions of the GUI, extension of the transformation setting of the core architecture, and extensions in the proof session manager. A feature, that has important consequences on our approach presented below for adding interactive proving in Why3, is the genericity of transformations handling. The Why3 library is designed so that an additional user-written transformation can be dynamically loaded at run-time, using a mechanism of registration with a name.

2.2 Extending User Interface

As shown in Figure 2, the graphical interface is naturally where proofs tasks are displayed and where the user can decide which transformations to apply and which prover to call. Below the top-right part of the window, where the current proof task is displayed, and above the bottom-right part where different kinds of messages are displayed, we added a kind of command-line input field where the user can input arbitrary character text to form a command. If we consider our toy example, one possibility to progress towards proving the goal is to replace `y*y` by the term `4*(x*x+x)+1`. To achieve that, the user can directly input the text

```
replace y*y 4*(x*x+x)+1
```

in the input field and hit the return key. An alternative possible transformation would be to instantiate the hypothesis `H` with `x*x+x`, which can be done with the input

```
instantiate H x*x+x
```

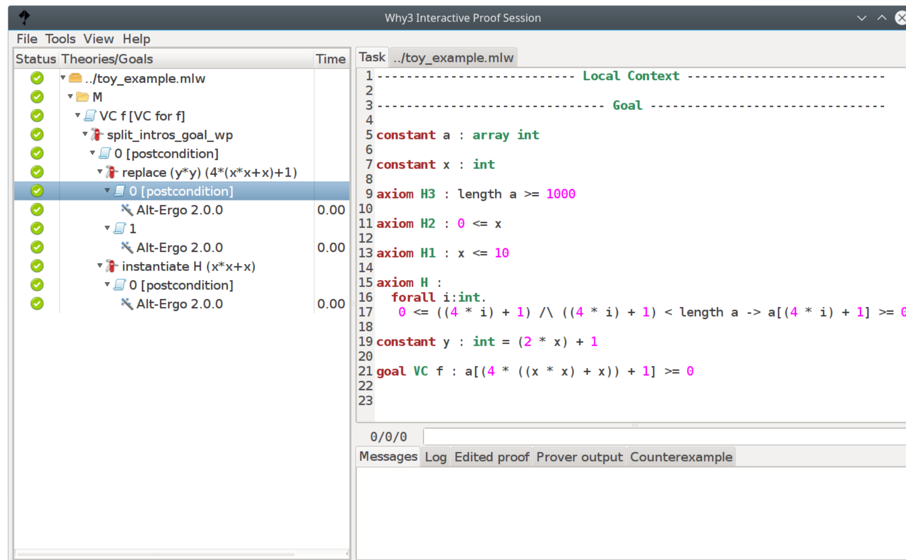

Fig. 3. Completed proof using two possible transformations with arguments

Figure 3 displays the GUI after trying both transformations. As seen on left, the transformation `replace y*y 4*(x*x+x)+1` generated two subgoals: first proving the formula after the replacement, second proving that `y*y=4*(x*x+x)+1`. Both subgoals are proved by Alt-Ergo. Similarly, the transformation "`instantiate H x*x+x`" generates one subgoal where an additional hypothesis is present (the instance of `H` for the given particular value for `i`) and is also proved by Alt-Ergo. As can be seen, applying any of these two transformations is enough to finish the proof automatically.

Even if this mechanism using a textual interface may seem old-fashioned, it permits a lot of genericity. We'll see below how it simplifies the communication with a front-end such as SPARK. It also permits a lot of extra features that show themselves important in practice, such as searching in the proof context.

2.3 Adding Parameters to Proof Transformations

A central design choice towards the introduction of interactive proofs is to reuse the existing infrastructure of transformations. Basically, since that infrastructure already allows the user to select among a given set of transformations to apply on a given proof task, we just have to extend this set, after extending the concept of transformations so that they can take parameters, like the two transformations used above: `replace` is a transformation that takes two terms as input. `instantiate` takes an hypothesis name and a term. We faced two main issues for this extension. First, the transformations can take various objects as parameters: a term, a formula, a name, a string, etc. It means that at the level of the API, we need some typing mechanism in order to declare what are the right types of objects to pass as parameters. Second, at the level of the interface, the

data submitted by the user are just textual, so we need a generic infrastructure to parse them and turn them into objects of the right kind.

At the level of the API, in order to handle the large variability of the kinds of transformation parameters, we were able to use the advanced concept of GADT (Generalized Abstract Data Types). An excerpt of the new declaration of transformation type in the API is as follows (the real one has 20 constructors):

```
type _ trans_typ =
  | Ttrans_l : (task -> task list) trans_typ
  (** transformation with no argument, and many resulting tasks *)
  | Tstring : 'a trans_typ -> (string -> 'a) trans_typ
  (** transformation with a string as argument *)
  | Tprsymbol : 'a trans_typ -> (Decl.prsymbol -> 'a) trans_typ
  (** transformation with a Why3 proposition symbol as argument *)
  | Tterm : 'a trans_typ -> (Term.term -> 'a) trans_typ
  (** transformation with a Why3 term as argument *)
  | Topt : string * ('a -> 'c) trans_typ -> ('a option -> 'c) trans_typ
  (** transformation with an optional argument. The first string is
     the keyword introducing that optional argument*)
```

To implement a transformation like `instantiate` above, we first have to program it with an OCaml function, say `inst`, of type `prsymbol` $\rightarrow$ `term` $\rightarrow$ `task list`, and then register it under the proper name as follows

```
wrap_and_register "instantiate" (Tprsymbol (Tterm Ttrans_l)) inst
```

Not only will it make the transformation available for use in the interface, but it will automatically proceed with the parsing, name resolution and typing of the textual arguments, as given by the type `(Tprsymbol (Tterm Ttrans_l))`. This mechanism based on GADTs is powerful enough to handle optional parameters. For example, the `replace` transformation is declared with type

```
(Tterm (Tterm (Topt ("in", Tprsymbol Ttrans_l))))
```

which means that a third optional argument is allowed, of type `prsymbol` and introduced by the keyword `in` to say that the replacement should be done in the hypothesis of the given name instead of the goal, e.g. "`replace (length a) 1000 in H`". Notice the large genericity of this mechanism, in particular the keyword used for introducing the optional argument. The genericity also comes from the `wrap_and_register` function which is defined once and for all and does all the hard job. In particular, the resolution of variable names given as arguments had to be carefully made consistent with the printing of the task, which can rename variables.

Here is a quick summary of the major transformations with parameters that we added in Why3. They are supposed to cover the major needs for interaction as already listed in Section 1.2.

- case analysis on a formula (`case` P), on an algebraic data (`destruct_alg` t), decomposition on propositional structure of an hypothesis (`destruct` H). For example, the transformation "`case`P" where P is an arbitrary formula would be transforming a task $\Gamma \vdash G$ into the two tasks $\Gamma, P \vdash G$ and $\Gamma, \neg P \vdash G$.

- introduction of an auxiliary hypothesis (`cut` P, `assert` P) or term (`pose` x t)
- induction on integers, on inductive predicates
- instantiation as seen above (`instantiate` H $t_1, \ldots, t_k$), including existential case (`exists` t), or via direct application of an hypothesis to a goal (`apply` H)
- various rewriting and computation transformations: `rewrite` H (`in` H'), `replace` t_1 t_2 (`in` H), `subst` x, `subst_all`, etc.
- context handling: `remove` $H_1, \ldots, H_k$, `clear_but ...`
- unfolding a definition: `unfold` f
- import an extra theory: `use_th` T

2.4 Extending The Proof Session Mechanism

A proof session is essentially a record of all the proof tasks generated from a given input file, and also a record of all transformations applied to these tasks. It is indeed an internal representation of the proof task tree displayed on the left part of Figure 2. Such a session can be stored on disk, and can be reloaded to a former state by the user. A crucial feature of the session manager is to manage the changes if the input file is modified (e.g. more annotations are added): the manager implements a clever and sound *merging* operation to discover which parts of the proof session can be reused, which tasks are modified, which external proofs should be replayed [5].

The Why3 session files do not store any internal representation to avoid any problem when the Why3 tools themselves evolve. Accordingly, we decided that the arguments of transformations should be stored under their textual form too. This definitely avoids potential problems with changes in internal representations, but still some problems with renaming could occur. For example, an automatically introduced name for any hypothesis, say `H1`, could perfectly be renamed into `H2`, e.g. if an extra annotation is added in the source code. It is thus perfectly normal that from time to time, while reloading a proof session, a transformation with argument does not apply anymore. In order to avoid the loss of any sub-proof tree, we implemented the new notion of *detached nodes* in the proof task tree. These nodes are a record of the state of the previous session, but simply without any corresponding proof task anymore. We then implemented a mechanism to copy and paste fragments of proof trees from one node to another. This copy-paste mechanism showed itself very useful in practice for maintaining interactive proofs.

2.5 Examples

A first way we evaluated the new interactive proof features of Why3 was to restudy past examples where some VCs could not be discharged except using the Coq proof assistant. Bobot et al. paper [6] illustrated the use of Why3 on the three challenges of the VerifyThis competition in 2012. On each of these case studies, at least one Coq proof was required. We have reconsidered the first challenge (Longest Repeated Substring) and were able to replace all three Coq proofs with interactive proofs. Interestingly, only very few transformations were needed, because we quickly arrived to simpler subgoals that are discharged by automatic solvers.

Another illustrative example is the proof of Dijkstra's shortest path algorithm on graphs: again, we were able to replace Coq proofs with interactive transformations and

automatic solvers. We noticed that not only does it simplify the proofs, they are now easier to maintain in case of a change in Why3 implementation or standard library.

We still have to evaluate to what extent the interaction using transformations with argument is easy to use for regular users. Moreover, we need more practice in order to see if this mechanism is of effective help for debugging proofs, as explained in the introduction.

3 Adding Interactive Proving in SPARK

Although the SPARK verifier, called GNATprove, is based on Why3 for generating VCs and proving them, there is a large gap between the SPARK and WhyML programming languages. Therefore, the interactive proof features of Why3 cannot be used directly by SPARK users. The same issue arose in the past with the counterexample generation features of Why3, which required translation back to SPARK for use inside GNATprove [9]. That issue also involved interactions with users inside different IDEs, Why3IDE for Why3 users and GNAT Programming Studio (GPS) for SPARK users, but that interaction was one-way only: the counterexamples output by provers were translated back to either Why3 or SPARK syntax and displayed in their respective IDE. Here, we need a two-way interaction where users can input commands (possibly with elements of the code as parameters) and Why3 returns a modified set of proof tasks.

We start by presenting a simple SPARK program that cannot be proved with automatic provers in Section 3.1. Then we describe in Section 3.2 the client-server architecture that we have adopted for two-way communication between the IDE for SPARK programs and Why3. We detail in Section 3.3 how we translate back and forth between user-level names in SPARK and internal names in Why3. Finally, we explain in Section 3.4 how to complete the proof of our example, which resisted provers.

3.1 Unprovable Code Example

The code in Figure 4 is a simplified version of an excerpt from a bounded string library. This code contains a post-condition that cannot be proved today by any prover available with SPARK (Alt-Ergo, CVC4, Z3). The reason for this unproved property is characteristic of the kind of problems faced by users of a technology like SPARK. It is in the class of problems called *quantifier instantiation* already presented in Section 2.

The code in Figure 4 computes the location of a sub-list of integers within a list of integers, when the sub-list is contained in the list. Lists are implemented here as SPARK arrays starting at index 1 and ranging over positive indexes. The post-condition of function `Location` introduced by `Post` states the following properties: the result of the function ranges from 0 to the length of the list; a positive result is used when the sub-list is contained in the list, and value 0 is used as result otherwise. For the sake of simplicity, we do not state the complete post-condition of `Location`, but this could be done easily. This post-condition relies on the definition in function `Contains` of what it means for a sub-list `Fragment` to be contained in a list `Within`. This function is only used in specifications, which is enforced by marking it as `Ghost`. It is defined as an expression function, quantifying with `for some` (Ada existential quantification) over a

```
type List is array (Positive range <>) of Integer
  with Predicate => List'First = 1;
subtype Natural_Index is Integer range 0 .. Positive'Last;

function Contains (Within : List; Fragment : List) return Boolean is
  (Fragment'Length in 1 .. Within'Length and then
    (for some K in 1 .. (Within'Length - Fragment'Length + 1) =>
      Within (K .. (K - 1 + Fragment'Length)) = Fragment))
with Ghost;

function Location (Fragment : List; Within : List) return Natural_Index
with
  Post => Location'Result in 0 .. Within'Length and then
    (if Contains (Within, Fragment) then
       Location'Result > 0
     else
       Location'Result = 0)
is begin
   if Fragment'Length in 1 .. Within'Length then
      for K in 1 .. (Within'Length - Fragment'Length + 1) loop
         if Within (K .. (K - 1 + Fragment'Length)) = Fragment then
            return K;
         end if;
         pragma Loop_Invariant
           (for all J in 1 .. K =>
              Within (J .. (J - 1 + Fragment'Length)) /= Fragment);
      end loop;
   end if;
   return 0;
end Location;
```

Fig. 4. Example of SPARK code with an unprovable post-condition

range of scalar values the property that the sub-list `Fragment` is equal to a moving slice of the list `Within (K .. (K - 1 + Fragment'Length))`. This last expression is the slice of array `Within` from index `K` to index `(K - 1 + Fragment'Length)`. For the sake of simplicity, function `Location` naively iterates over each possible location for a match and tests for equality of the corresponding slice with the argument sub-list. The loop invariant repeats the post-condition and specializes it for the Kth iteration of the loop, so that the loop invariant is itself provable and can be used to prove the post-condition.

When running GNATprove on this program, it proves automatically the absence of runtime errors (no integer overflows, no array access out of bounds, no other runtime check failures), as well as the loop invariant in function `Location`, but it does not prove the post-condition of that function. The user interaction is quite simple here. The user requests that this program is verified by GNATprove inside GPS, and a few seconds later receives the output of the tool as messages attached to program lines. Between

these two instants, GPS called GNATprove; GNATprove translated the SPARK program into an equivalent WhyML program w.r.t. axiomatic semantics for generation of VCs; an internal program using Why3 API successively generated VCs by calling Why3 VC generator and dispatched VCs to provers; this internal program collected the output of provers and returned the overall results to GNATprove; GNATprove generated and adapted results for GPS; and GPS displayed these results to the user.

All this work occurred transparently for the user, who never had to see the generated WhyML code in Why3IDE, or to launch Why3 commands in a terminal. A less-than-ideal process for completing the proof of the post-condition would consist in asking the user to open the session file generated by Why3 in Why3IDE to complete the proof using the interactive proof feature described in Section 2. While this would work, this is not a suitable solution in an industrial context. Indeed, asking users to interact directly with a generated artifact in a different language (the Why3 file) is akin to asking them to debug their programs at assembly level. While this is possible and sometimes useful, it is best left to rare occasions when this is really needed, and instead interaction should be done as much as possible at source code level.

3.2 Client-Server Architecture

Most modern IDEs like GPS provide client-server interfaces with various tools such as debuggers or, in the context of formal verification, with proof assistants such as Emacs ProofGeneral support for Coq, Isabelle, Lego, HOL, etc. We have adopted a client-server architecture to allow two-way communications between the interactive proof module (acting as a server) and the client IDE, which can be Why3IDE or GPS here. The server handles requests from the user (through the IDE), such as proof transformations (see Section 2.3) and direct calls to provers. After the requested action terminates, the server informs the IDE of changes to the proof task tree.

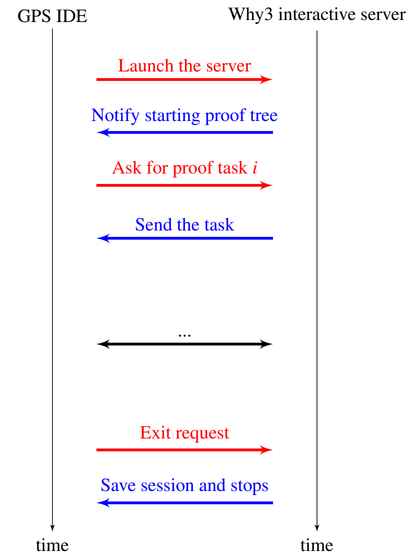

Fig. 5. Schematic of the interactions between IDE and server

For the integration in GPS, we developed both a wrapper in OCaml for the underlying service to act as the server, and a plugin in Python to communicate with the server from GPS. The server takes the session file as initial argument, gets its input in JSON format on standard input, calls Why3 core services, updates the session file accordingly, and returns its output in JSON format on the standard output. The plugin modifies GPS interface to add a console window for command-line interaction, a window to display VCs and a window to display the proof task tree. The plugin translates requests made by the user on the command-line interface into JSON requests that are sent to the server, and translates back the server notifications into updates of the graphical user interface (adding nodes in the proof task tree, changing the VC, etc.). For example, as seen in Figure 5, GPS first starts the server, then the server

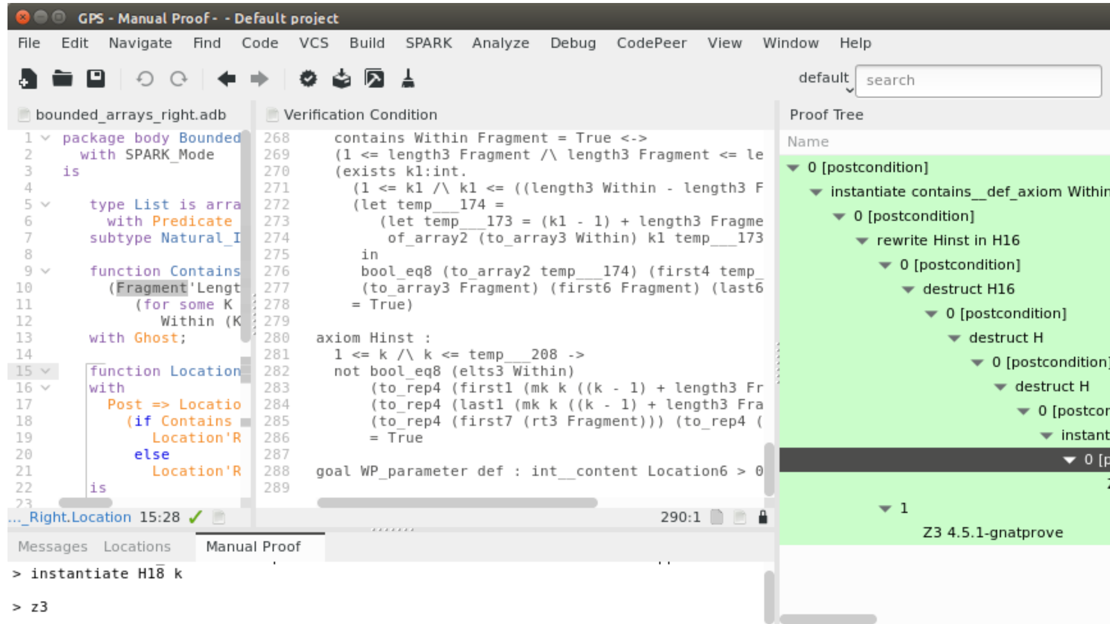

Fig. 6. Example of interactive proof in GPS

returns the initial proof task tree which is printed by GPS. When the user clicks a node, GPS asks for the corresponding proof task which the server returns and GPS then prints it for the user. This goes on until the user exits manual proof which GPS interprets by sending the exit request: Why3 interactive server ends its execution. The death of the process is detected by GPS which goes back to its normal interface. A schematic of the interactions between the IDE and the server is depicted in Figure 5. With very little effort, we transformed a generic IDE such as GPS into an elementary proof assistant.

As seen on Figure 6, the user interface in GPS is similar to the one in Why3IDE presented in Figure 2. The same windows are present but they are not located at the same place. From left to right, we can see the SPARK code window, the proof task window and the proof task tree window. The command-line console is displayed as a bottom panel sharing its window with other panels for Messages (tool output) and Locations (tool messages). The user can type commands for applying transformations and calling provers inside the command-line console, similar to what we saw for Why3IDE.

One can observe that the VC from Figure 6 is much more complex than the simple VC we generated with Why3 in Figure 2. This is mainly a consequence of the complexity of the WhyML code generated from SPARK. Simple data structures and control flow in SPARK are modeled with much more complex data structures and control flow in WhyML to encode the semantic features of SPARK. This complexity gets exposed in the VC generated by Why3 VC generator, as it combines the complexity of data structures and control flow. This inherent complexity cannot be eliminated and must be dealt with to present SPARK users with understandable VCs.

3.3 Printing of Proof Tasks and Parsing of Transformation Arguments

In order for users to be able to relate the proof task to their code, it is necessary to use the names of source SPARK entities in the proof task instead of the generated Why3 names. This is achieved by creating a mapping from Why3 names to their source SPARK name during the generation of Why3 code from SPARK. This mapping is embedded in the WhyML code using *labels*, a generic mechanism in Why3 to attach strings to terms. For example, in the following code snippet the label `"name:Not_y"` is attached to the identifier `y`:

```
let f (a:array int) (x:int) : int
  ensures {result = 20}
= let y "name:Not_y" = 2*x+1 in y*y
```

Labels on terms are preserved by VC generation and transformations. For example, the label `"name:Not_y"` remains attached to the constants derived from `y` in the final VC obtained after VC generation and transformations. Thus, when printing a proof task, the name of an identifier can be replaced by the name found in the attached label when there is one. Thus we get the following proof task where the source SPARK name `Not_y` is used instead of the generated Why3 name `y`:

```
constant x : int
constant Not_y : int = (2 * x) + 1
goal VC f : (Not_y * Not_y) = 20
```

Different Why3 names with the same name label are currently distinguished by appending a unique number to the source SPARK name. The consecutive problem of interpreting the names of SPARK entities as Why3 names occurs when the user types a command with arguments referring to the names of SPARK entities. Why3 uses here the inverse map from distinguished SPARK names to Why3 names associated to a given proof task to translate automatically the arguments of transformations.

3.4 Application to the Unprovable Code Example

We can use the interactive proof interface in GPS to complete the proof of the post-condition of `Location` from Section 3.1. For the sake of simplicity, we will only describe the proof part of the `then` branch in this subsection, shown in the screenshot in Figure 6. The root of the proof task tree is green, showing that the initial VC was fully proved.

The interactive proof proceeds by deriving a contradiction from the loop invariant property in the last iteration of the loop and the condition of the `then` branch `Contains (Within, Fragment)`. This requires unfolding the definition of `Contains` and finding suitable instances of quantified properties. In this case, the proof task tree is linear and a complete proof script is the following:

```
instantiate contains__def_axiom Within,Fragment
rewrite Hinst in H16
destruct H16
destruct H
```

```
destruct H
instantiate H18 k
```

The intuition of the proof is that we need to explain to the tool how to combine the hypotheses coming from the loop invariant and the definition of `Contains`. The first two transformations (`instantiate`, `rewrite`) are used to make the axiomatic definition of `Contains`, applied to the right arguments, appear as an hypothesis in the context. The three calls to the `destruct` transformation are used to destruct the head connective of the hypothesis that appeared. They transform $\Gamma, H : (A \wedge \exists k.P(k)) \vdash G$ into $\Gamma, H : A, k :$ `int`$, H_1 : P(k) \vdash G$. The objective is to use k as an instance for the loop invariant property (to make the contradiction appear). This is exactly what transformation "`instantiate H18 k`" does. The quantification disappears and SMT solvers are now able to solve the goal. This completes the proof of the `then` branch of the post-condition of `Location`. The `else` branch is handled similarly, which completes the proof of the program.

4 Conclusions and Future Work

We brought interactive proving features to the SPARK verification environment for Ada. This was done by conservatively extending the intermediate Why3 tool by allowing to pass arguments to proof task transformations. The design is generic enough to allow simple addition of new transformations via Why3's API. The proof session mechanism and the graphical interface have been extended to allow simple user interaction and facilities for proof maintenance. A few program examples were re-proved, showing that former external interactive proofs using Coq can be substituted with light interactive proofs in our new setting. The user interface of Why3 was redesigned under the form of a generic client-server architecture, allowing to bring interactive proving features to the SPARK front-end inside the regular GPS graphical interface.

Future Work will go into several directions. First, we certainly want to enlarge the set of transformations with arguments, to cover more needs for interactive proofs. The needed transformations will be identified from practice. Notice that others are already reusing our API to implement new transformations, for example for doing proofs by reflection for complex non-linear goals [14]. Second, we plan to reuse the client-server architecture to provide an alternative interface for Why3, this time within a web browser. We also plan to bring the interactive proof feature to the Frama-C front-end for C code, by augmenting the existing Jessie plug-in of Frama-C that uses Why3 internally. A third longer-term work is to allow for more customization of the printing of tasks and the parsing of transformation arguments: from SPARK, we would like the terms in proof tasks expressed in a more Ada-like syntax, in particular for arrays. For this we will need to design in Why3's generic API a possibility to register printers and parsers. A fourth even longer-term issue is the question of trust in the implemented transformations. Since we implement more and more complex OCaml code for that purpose, a general need for verifying the soundness of the transformations shows up. It is likely that we will need to design a language of proof terms or proof certificates to achieve this long-term goal.

References

1. Abrial, J.R., Cansell, D.: Click'n Prove: Interactive proofs within set theory. In: Theorem Proving in Higher Order Logics, 16th International Conference, TPHOLs'03. Lecture Notes in Computer Science, vol. 2758, pp. 1–24. Springer (2003)
2. Ahrendt, W., Beckert, B., Menzel, W., Reif, W., Schellhorn, G.: Automated Deduction - A Basis for Applications, Applied Logic Series, vol. 9, chap. Integrating Automated and Interactive Theorem Proving. Springer, Dordrecht (1998)
3. Berghofer, S.: Verification of dependable software using SPARK and Isabelle. In: Brauer, J., Roveri, M., Tews, H. (eds.) 6th International Workshop on Systems Software Verification. OpenAccess Series in Informatics (OASIcs), vol. 24, pp. 15–31. Dagstuhl, Germany (2012)
4. Blanchette, J.C., Böhme, S., Paulson, L.C.: Extending sledgehammer with SMT solvers. J. Autom. Reasoning 51(1), 109–128 (2013)
5. Bobot, F., Filliâtre, J.C., Marché, C., Melquiond, G., Paskevich, A.: Preserving user proofs across specification changes. In: Cohen, E., Rybalchenko, A. (eds.) Verified Software: Theories, Tools, Experiments (5th International Conference VSTTE). Lecture Notes in Computer Science, vol. 8164, pp. 191–201. Springer, Atherton, USA (May 2013)
6. Bobot, F., Filliâtre, J.C., Marché, C., Paskevich, A.: Let's verify this with Why3. International Journal on Software Tools for Technology Transfer (STTT) 17(6), 709–727 (2015), see also http://toccata.lri.fr/gallery/fm2012comp.en.html
7. Chapman, R., Schanda, F.: Are we there yet? 20 years of industrial theorem proving with SPARK. In: Klein, G., Gamboa, R. (eds.) Interactive Theorem Proving - 5th International Conference, ITP 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 14-17, 2014. Proceedings. Lecture Notes in Computer Science, vol. 8558, pp. 17–26. Springer (2014)
8. Filliâtre, J.C., Paskevich, A.: Why3 — where programs meet provers. In: Felleisen, M., Gardner, P. (eds.) Proceedings of the 22nd European Symposium on Programming. Lecture Notes in Computer Science, vol. 7792, pp. 125–128. Springer (Mar 2013)
9. Hauzar, D., Marché, C., Moy, Y.: Counterexamples from proof failures in SPARK. In: De Nicola, R., Kühn, E. (eds.) Software Engineering and Formal Methods. pp. 215–233. Lecture Notes in Computer Science, Vienna, Austria (2016)
10. Hentschel, M., Hähnle, R., Bubel, R.: An empirical evaluation of two user interfaces of an interactive program verifier. In: Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering. pp. 403–413. ASE 2016 (2016)
11. Hentschel, M., Hähnle, R., Bubel, R.: The interactive verification debugger: Effective understanding of interactive proof attempts. In: Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering. pp. 846–851. ASE 2016, ACM (2016)
12. McCormick, J.W., Chapin, P.C.: Building High Integrity Applications with SPARK. Cambridge University Press (2015)
13. Mehta, F.: Supporting proof in a reactive development environment. In: Proceedings of the Fifth IEEE International Conference on Software Engineering and Formal Methods. pp. 103–112. SEFM'07, IEEE Computer Society (2007)
14. Melquiond, G., Rieu-Helft, R.: A Why3 framework for reflection proofs and its application to GMP's algorithms. In: 9th International Joint Conference on Automated Reasoning. Oxford, United Kingdom (Jul 2018), https://hal.inria.fr/hal-01699754
15. Petiot, G., Kosmatov, N., Botella, B., Giorgetti, A., Julliand, J.: Your proof fails? testing helps to find the reason. In: Tests and Proofs - 10th International Conference. Lecture Notes in Computer Science, vol. 9762, pp. 130–150. Springer (2016)