

KeY-ABS

<http://abs-models.org/keyabs-tutorial/>

CRYSTAL CHANG DIN



<http://www.envisage-project.eu/>

KeY-ABS is an interactive (semi-automatic) deductive verification tool that enables one to verify functional properties for concurrent and distributed ABS models with unbounded size. In this tutorial we demonstrate in detail how to specify and verify ABS applications. We strongly encourage reading this tutorial next to a computer with a running KeY-ABS system.

Contents

1	Introduction	3
2	Tool Installation	3
3	System Workflow	3
4	ABS Dynamic Logic	4
5	Example	4
5.1	Verifying the AccountImpl class against the nonNegativeBalance invariant	6
5.2	Open goals of the proof tree	6
5.3	Step-by-step interactive proof	11
6	Conclusions and Further Reading	19
7	References	22

1 Introduction

KeY-ABS is a variant of the verification system KeY for sequential Java/JavaCard. It is based on the KeY 2.0 platform - a verification system for Java. We generalize various subsystems of KeY and abstract away the Java specifics. After refactoring the KeY system provides core subsystems (rule engine, proof construction, search strategies, specification language, proof management etc.) that are independent of the specific program logic or target language. These are then extended and adapted by the ABS and Java backends.

KeY-ABS is designed for the verification of concurrent and distributed ABS models. By following this tutorial, readers should be able to execute KeY-ABS on their own machine, reproduce the proof of the example, and understand what KeY-ABS can do. Sufficient background knowledge and references are provided so the readers can understand the technicality of how the tool works.

2 Tool Installation

For a local installation of the KeY-ABS theorem prover, install Java 8. Then, download KeY-ABS from <http://heim.ifi.uio.no/~crystalld/key-abs.zip>. Unzipping the downloaded file and double-clicking on the `key.jar` file should start KeY-ABS. To start it from the command line, enter the directory `key-abs` and use:

```
java -jar key.jar
```

3 System Workflow

The input files to KeY-ABS comprise (i) an `.abs` file containing ABS program and (ii) a `.key` file containing class invariants, functions, predicates and problem specific rules required for this particular verification case. Given these input files, KeY-ABS opens a proof obligation selection dialogue that lets one choose a target method implementation. From the selection the proof obligation generator creates an ABSDL formula, which will be explained in Section 4. By clicking on the Start icon the verifier will try to automatically prove the generated formula. A positive outcome shows that the target method preserves the specified class invariants. In the case that a subgoal cannot be proved automatically, the user is able to interact with the verifier to choose proof strategies and proof rules manually. The reason for a formula to be unprovable might be that the target method implementation does not preserve one of the class invariants, that the specified invariants are too weak/too strong or that additional proof rules are required. The workflow of KeY-ABS is illustrated in Fig. 1.

4 ABS Dynamic Logic

Specification and verification of ABS models is done in ABS dynamic logic (ABSDL). ABSDL is a typed first-order logic plus a box modality: For a sequence of executable ABS statements S and ABSDL formulae P and Q , the formula $P \rightarrow [S]Q$ expresses: If the execution of S starts in a state where the assertion P holds and the program terminates, then the assertion Q holds in the final state. Given an ABS method m with body mb and a class invariant I , the ABSDL formula $I \rightarrow [mb]I$ expresses that the method m preserves the class invariant. In sequent notation $P \rightarrow [S]Q$ is written

$$\Gamma, P \vdash [S]Q, \Delta$$

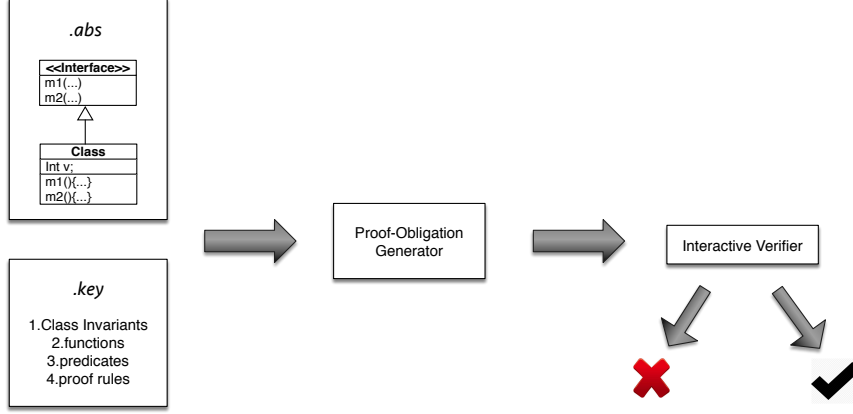


Figure 1: Verification workflow of KeY-ABS

where Γ and Δ stand for (possibly empty) sets of side formulae. A sequent calculus as realized in ABSDL essentially constitutes a symbolic interpreter for ABS. For example, the assignment rule for local program variables is

$$\text{assign} \frac{\Gamma \vdash \{v := e\}[\text{rest}]\phi, \Delta}{\Gamma \vdash [v = e; \text{rest}]\phi, \Delta}$$

where v is a local program variable and e is a pure (side effect-free) expression. This rule rewrites the formula by moving the assignment from the program into a so-called *update* [1], as $\{v := e\}$ shown above, which captures state changes. The symbolic execution continues with the remaining program rest . Updates can be viewed as explicit substitutions that accumulate in front of the modality during symbolic program execution. Updates can only be applied to formulae or terms. Once the program to be verified has been completely executed and the modality is empty, the accumulated updates are applied to the formula after the modality, resulting in a pure first-order formula. This is shown by the following rule *emptybox*.

$$\text{emptyBox} \frac{\Gamma \vdash U\phi, \Delta}{\Gamma \vdash U[]\phi, \Delta}$$

The rule for the conditional splits the proof into two branches; one for the case where its guard b evaluates to true (and the conditional's then-block is executed), the other for the case where b evaluates to false.

$$\text{ifElse} \frac{\Gamma \vdash U(b \rightarrow [s_1; \text{rest}])\phi, \Delta \quad \Gamma \vdash U(\neg b \rightarrow [s_2; \text{rest}])\phi, \Delta}{\Gamma \vdash U[\text{if } b \text{ then } s_1 \text{ else } s_2; \text{rest}]\phi, \Delta}$$

More reasoning rules will be introduced along the tutorial.

5 Example

In this tutorial, we use a banking example written in ABS to illustrate how specifications can be written and how programs are verified by KeY-ABS. The source code of the example are available at: http://folk.uio.no/crystalD/BankingAccount_tutorial_code.zip. The zip file contains the ABS banking module and two corresponding specification files.

The file `Account.abs` is the ABS implementation of the model, in which the `deposit` method increases the balance of the bank account by an input amount `x`, and the `withdraw` method decreases the balance by an input amount `x` if balance is at least `x`.

Listing 1 Banking example in ABS (`Account.abs`)

```
module Account;

interface Account {
  Int deposit(Int x);
  Int withdraw(Int x);
}

class AccountImpl(Int balance) implements Account {

  Int deposit(Int x) { balance = balance + x; return balance;}

  Int withdraw(Int x) {
    if (balance - x >= 0) {
      balance = balance - x;
    }
    return balance;
  }
}

{ new AccountImpl(100); }
```

We specify an invariant for the `AccountImpl` class that “*the balance is always positive*”. Class invariants are specified in the `.key` file. Listing 2 gives a template of `.key` files. It contains the directory of the `.abs` file, the ABS module name, the ABS class, and a list of class invariants. In addition, user-defined functions, predicates, proof rules and proof-obligation formulae may also be included in the `.key` file.

Listing 2 `.key` file template

```
\absSource “abs_file_directory”;

\module “module_name”;

\class “module_name.class_name”;

\invariants(Seq historySV, Heap heapSV, ABSAnyInterface self) {
  invariant_name1 : module_name.class_name {...};
  invariant_name2 : module_name.class_name {...};
  ...
}

\functions { ... }
\predicates { ... }
\rules { ... }
```

The parameters of the `\invariants` section include a variable, `historySV`, of type `Seq` for recording the history of communication; a variable, `heapSV`, of type `Heap` for storing the state of fields; and a variable, `self`, of type `ABSAnyInterface` as the identity of the current class instance. Each class invariant has a name and is formulated in the following format:

invariant_name : module_name.class_name { first_order_logic_formulae };

The invariant, “*the balance is always positive*”, of the `AccountImpl` class is specified in the `nonNegativeBalance.key` file shown in Listing 3.

Listing 3 Specification of the bank account example (nonNegativeBalance.key)

```
\absSource ".";

\module "Account";


\class "Account.AccountImpl";

\invariants(Seq historySV, Heap heapSV, ABSAnyInterface self) {

    nonNegativeBalance : Account.AccountImpl {
        int::select(heapSV, self, Account.AccountImpl::balance) >= 0
    };
}
```

Where the select function returns the value of the object field, i.e. `balance`, from the heap, i.e. `heapSV`, of the current class instance, i.e. `self`.


5.1 Verifying the AccountImpl class against the nonNegativeBalance invariant

Start the KeY-ABS system (shown in Fig. 2). Click the  icon and enter the directory of the unzipped example folder where the `.abs` and `.key` files are located. Select `nonNegativeBalance.key` file (see Fig. 3). A Proof-Obligation Browser window, shown in Fig. 4, will then pop up, by which we select the `withdraw` method of `AccountImpl`. Then the KeY-ABS proof obligation generator will generate an ABS Dynamic Logic (ABSDL) formula, shown in Fig. 5.

Listing 4 Proof obligation of the withdraw method

```
==>
\forall ABSAnyInterface caller;
( !caller = null
->\forall int x_0;
{ x:=x_0
{ history:=seqConcat(history,
seqSingleton(invocREv(caller,this,future,Account.Account::withdraw#ABS.StdLib.Int,
seqConcat(seqEmpty, seqSingleton(x))))))
( wellformed(heap) & wfHist(history) & !this = null & Precondition & CInv(history, heap, this)
-> \{ methodframe(source <- Account.Account::withdraw#ABS.StdLib.Int,
return <- (var:result,fut:future): {
if(this.balance + x >= 0) { this.balance = this.balance + x; }
return this.balance;
}
} \} CInv(history, heap, this)))
```

Listing 4 presents the generated ABSDL formula for the `withdraw` method. It expresses that *For any calling object caller and any input data `x_0` of type `int`, if the caller is not null, the initial history is wellformed, the class invariant is satisfied before executing `withdraw` and the execution of `withdraw` terminates, the class invariant must be proven upon the termination of `withdraw`.*

Now click the  icon. The automatic proof system of KeY-ABS will then be triggered. The proof obligation of the `withdraw` method can be automatically proved by KeY-ABS. This is shown in Fig. 6, where a popped up window lists the number of nodes and branches in the closed proof tree recorded on the left side of the window.

5.2 Open goals of the proof tree

Now follow the same procedure and try to prove the class invariant `nonNegativeBalance` for the `deposit` method. You may notice that KeY-ABS cannot close the proof automatically.

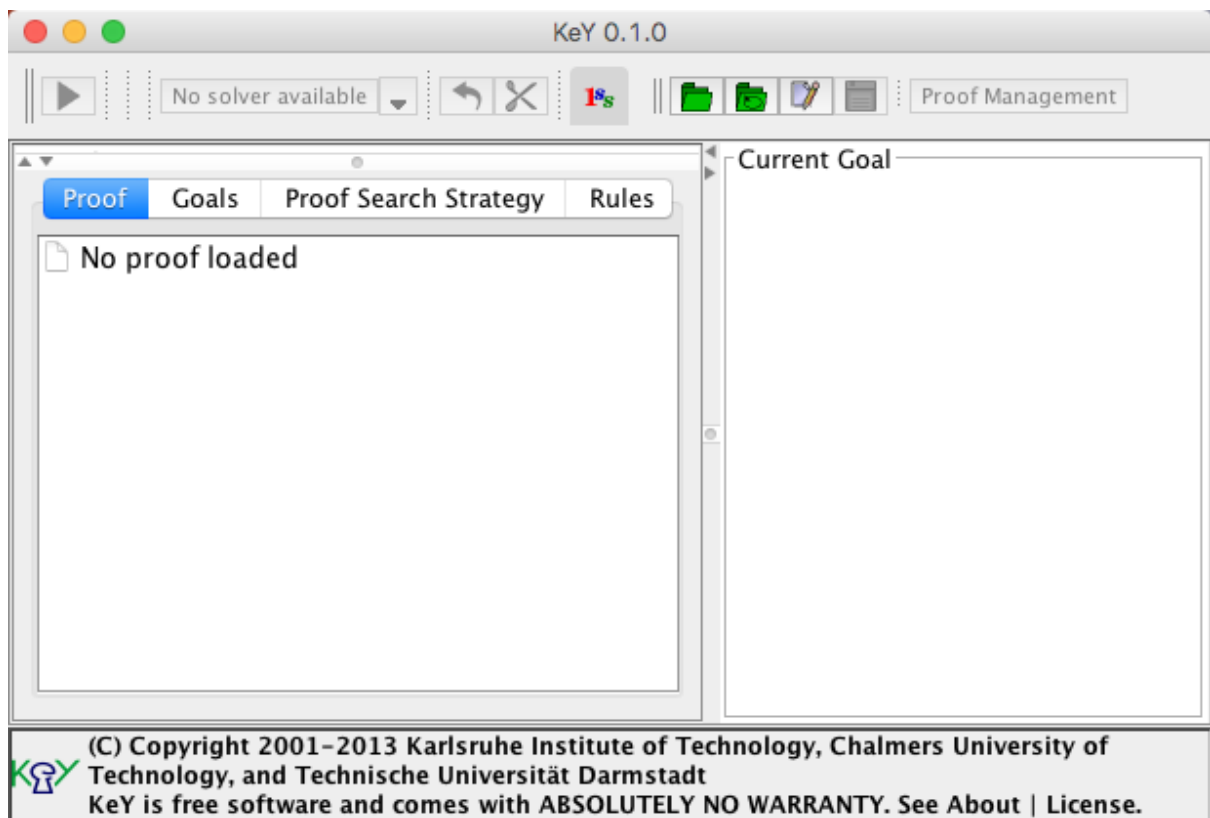


Figure 2: Start of KeY-ABS.

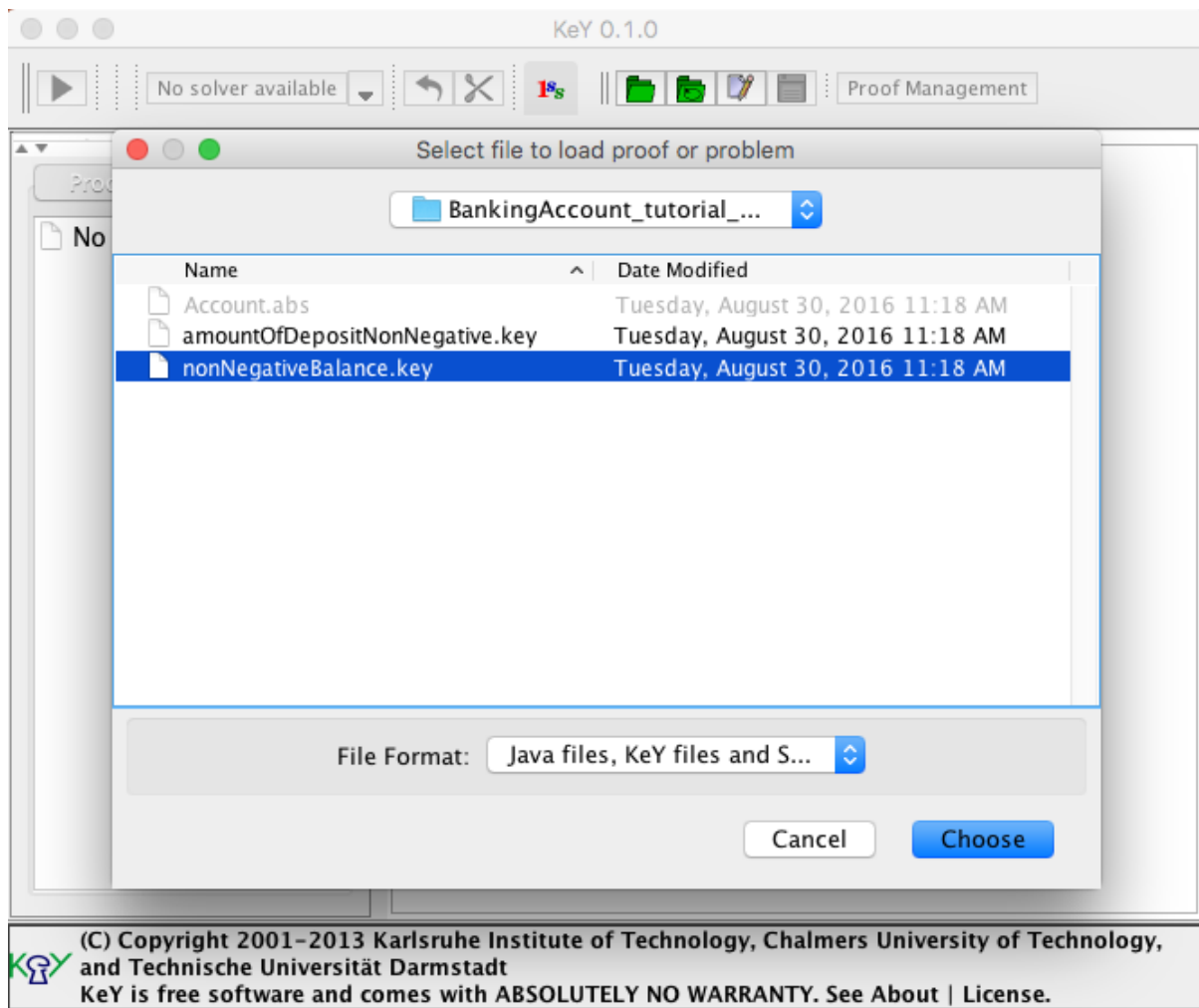


Figure 3: Invariant selection.

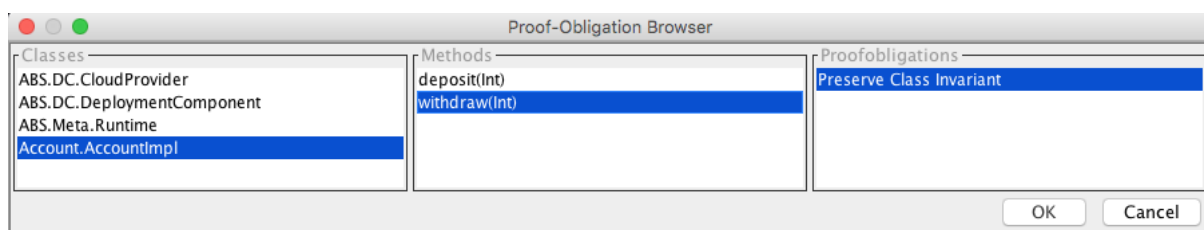


Figure 4: Proof obligation browser.

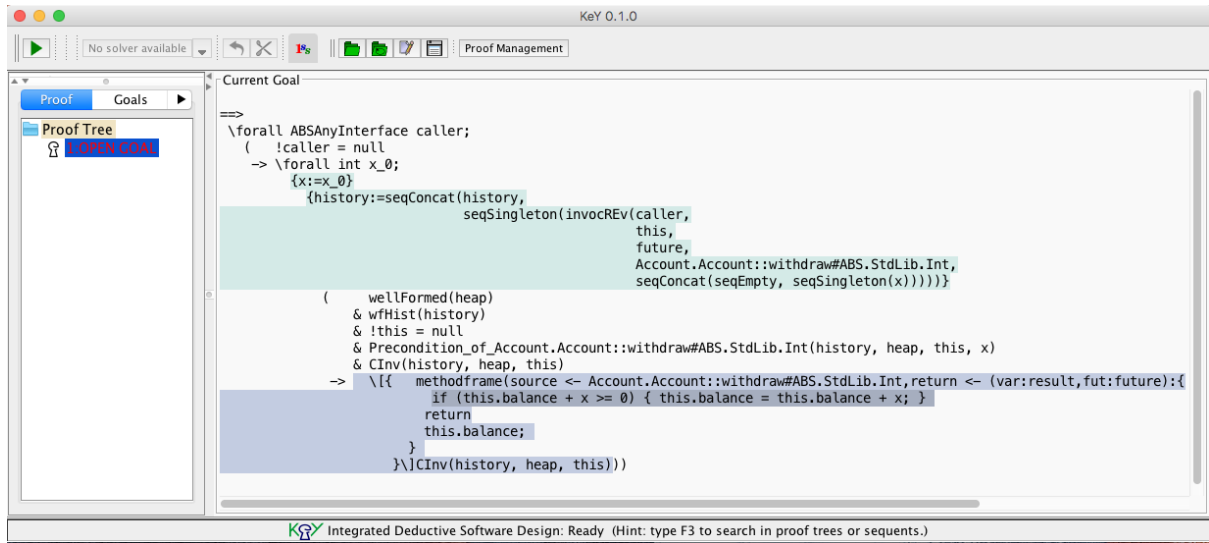


Figure 5: Automated generated proof obligation.

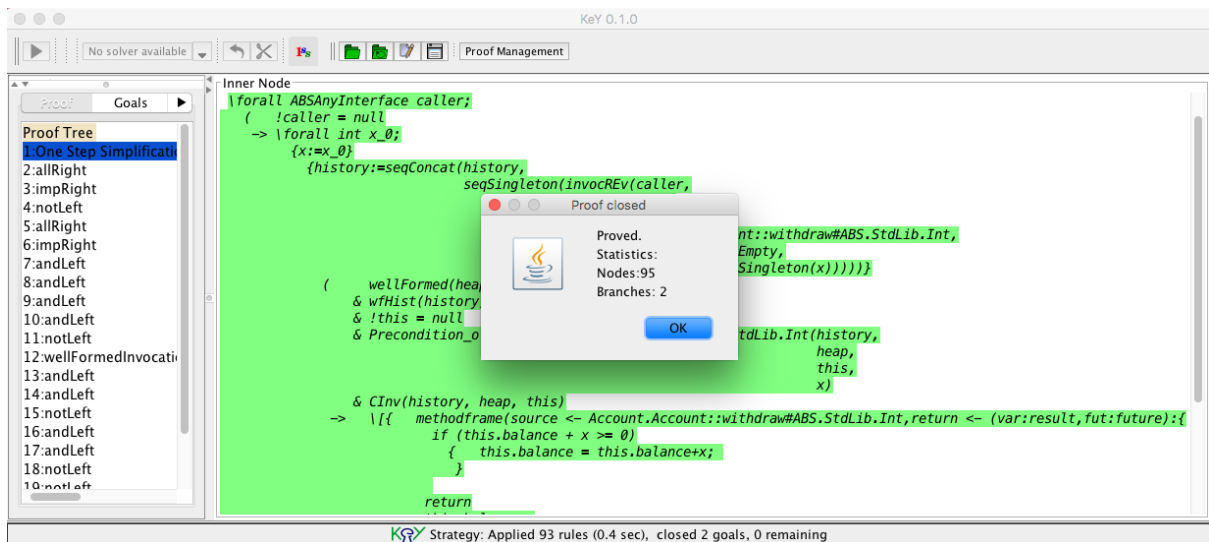


Figure 6: Closed proof and statistics for method withdraw.

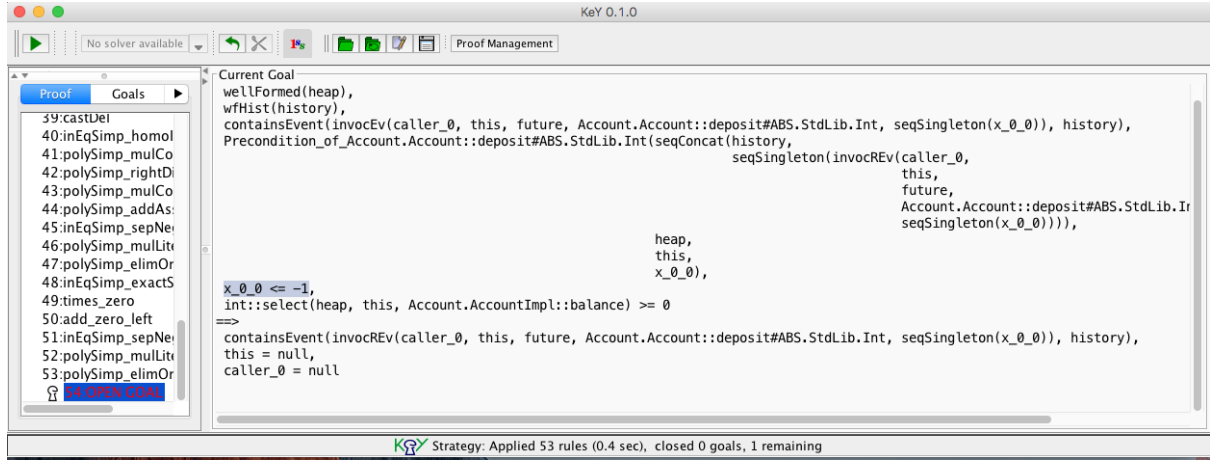


Figure 7: Open goal for proving deposit.

The open goal is shown in Fig. 7. To find out the reason why the proof cannot be closed, we may check if the class invariant is too weak/too strong, or if additional proof rules are required, or simply the method implementation is not correct. If we look closely to the formula at the right side of the window, shown in Fig. 7, while encountering the open goal, we notice that there is an assumption: $x_0_0 \leq -1$. It expresses that the parameter x_0_0 of `deposit` method is not positive. The proof cannot be closed because the balance might not stay positive upon the termination of `deposit` when the input parameter data is negative. KeY-ABS symbolically generates two execution branches in the proof tree for `deposit`: one for positive method parameter and one for negative method parameter. In order to prove that the balance is still positive after executing `deposit`, we need to strengthen the class invariant by specifying an additional property, i.e. *the method parameter for `deposit` is always non-negative*. This is captured by the history-based class invariant, `amountOfDepositNonNegative`, shown in Listing 5. History is a sequence of events recording the communication between objects. History based verification of ABS can be found in [4].

Listing 5 Additional class invariant for the banking example

```
amountOfDepositNonNegative : Account.AccountImpl {
  forall HistoryLabel ev; (
    forall int i; ( i >= 0 & i < seqLen(historySV) ->
      (ev = HistoryLabel::seqGet(historySV, i) &
        (isInvocationEv(ev) | isInvocationREv(ev)) &
        getMethod(ev) = Account.AccountImpl::deposit#ABS.StdLib.Int
        -> int::seqGet(getArguments(ev), 0) >= 0 )
    ))
};
```

In Listing 5 the function `seqLen(a)` returns the length of the input sequence `a`. The function `seqGet(a,b)` returns the elements of the input sequence `a` at index `b`. The predicates `isInvocationEv(ev)` and `isInvocationREv(ev)` return true if the input event `ev` is an invocation event and invocation reaction event, respectively. These two events records the method invocation from the caller side and the starting of method execution at the callee side. The function `getMethod(ev)` returns the method name contained in the event `ev`. The function `getArguments(ev)` returns the sequence of method parameters contained in the event `ev`. Method names are expressed in the following format:

module_name.class_name::method_Name#type_of_parameter1, type_of_parameter2,...

where a list of method parameters and the names of interface and class that the method belong to should be given. Class invariants are specified based on communication histories. This additional class invariants literally expresses that for all the invocation events and invocation reaction events for the `deposit` method, the method parameters contained in the events are always larger and equal to zero.

The updated specification is written in `amountOfDepositNonNegative.key`, shown in Listing 6, which contains two class invariants, `nonNegativeBalance` and `amountOfDepositNonNegative`. By selecting `amountOfDepositNonNegative.key` and the `deposit` method, KeY-ABS generates a proof obligation for `deposit` such that the conjunction of these two invariants should be proven upon method termination.

It has been proven that both `withdraw` and `deposit` methods preserve this strengthened class invariant. However, the proof is semi-automatic for both methods due to the need of quantifier instantiation. In the following section we prepare a step-by-step guidance for interacting with the KeY-ABS prover and accomplishing the proof.

Listing 6 Extended specification of the bank example (`amountOfDepositNonNegative.key`)

```
\absSource ".";

\module "Account";

\class "Account.AccountImpl";

\invariants(Seq historySV, Heap heapSV, ABSAnyInterface self) {

    nonNegativeBalance : Account.AccountImpl {
        int::select(heapSV, self, Account.AccountImpl::balance) >= 0
    };

    amountOfDepositNonNegative : Account.AccountImpl {
        \forall HistoryLabel ev; (
            \forall int i; ( i >= 0 & i < seqLen(historySV) ->
                (ev = HistoryLabel::seqGet(historySV, i) &
                    (isInvocationEv(ev) | isInvocationREv(ev)) &
                    getMethod(ev) = Account.Account::deposit#ABS.StdLib.Int
                    -> int::seqGet(getArguments(ev), 0) >= 0 )
            ))
    };
}
```

5.3 Step-by-step interactive proof

Interactive proof can be non-trivial and requires certain user experience. Depending on the proof strategy, there are different ways to interact with the prover to close a proof. This section will show some basic tips to interact with KeY-ABS through an example. A screencast of how to prove the example is provided in the end of the section. Users can easily exploit the tool further after fully understand this tutorial.

To begin with we need to find *where* to prune the proof and how to continue the proof from there. To prune the proof means to select a node in the middle of the proof tree and then cut off the whole subtrees below this node. In KeY-ABS this is done by clicking a node in the proof tree and press the ✂ icon in the top bar of the window. A straight forward place to prune the proof is the place in the proof tree right after KeY-ABS symbolically execute the whole method body. This state is captured by the node of the form $\langle n \rangle: \{ \}$ in the proof tree. The

symbol $\{ \}$ expresses that the method body is now empty and this reasoning rule removes the empty modality box in the ABSDL formula. Note that there are several nodes of the same form $\langle n \rangle: \{ \}$. We here choose the one closest to the root of the tree. We prune the proof right after the node 28: $\{ \}$, because the current node will be pruned as well.

In Fig. 8 we show the proof obligation formula generated after the execution of `deposit` method. Now we can clearly see that our goal is to prove the satisfaction of class invariant $CInv(history, heap, this)$ based on the *updates* updated by the method execution. This updates captures the updated state of the field in the heap, where `balance` is increased by `x_0_0`. In addition, the initial `history` sequence has been extended with an *invocation reaction event* and a *completion event* capturing the starting and the termination of the `deposit` method.

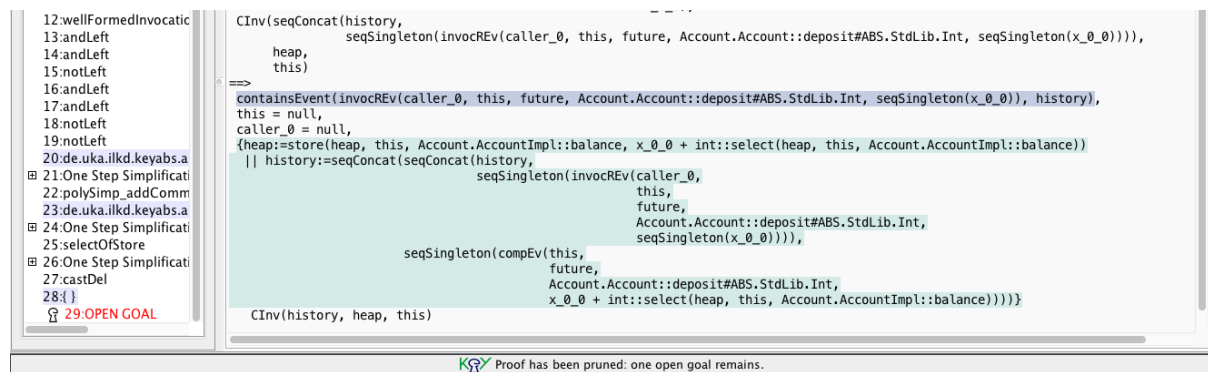


Figure 8: Prune the proof.

The automated proof search implemented in KeY-ABS can be interleaved with *interactive rule application*. The KeY-ABS prover has a graphical user interface that is built upon the idea of direct manipulation. To apply a rule, the user first selects a focus of application by highlighting a (sub-)formula or a (sub-)term in the goal sequent. The prover then offers a choice of rules applicable at this focus. Rule schema variable instantiations are mostly inferred by matching. Accordingly, now we move the mouse on the formula until both the *updates* and the class invariant are highlighted. This step is shown in Fig. 9. Then we **left-click** the mouse and select a reasoning rule called *One Step Simplification*. This reasoning rule substitutes the parameters of the $CInv$ predicate with the symbolic values provided by the *updates*. Then we highlight the whole $CInv$ predicate and apply the reasoning rule `insertClassInvariantFor<Account.AccountImpl>` over the predicate. This step is shown in Fig. 10. This rule substitutes the `history` and the object field `balance` in the class invariants with the symbolic values contained in the $CInv$ predicate. The substitution result is shown in Fig. 11.

The class invariant is a conjunction of `nonNegativeBalance` and `amountOfDepositNonNegative`. Let's prove them one by one. As the standard first-order-logic rule for conjunction at the right side of an implication will cause a split of proof tree into two branches, one for each conjunct,

$$\text{andRight} \frac{\Gamma \vdash \phi, \Delta \quad \Gamma \vdash \psi, \Delta}{\Gamma \vdash \phi \wedge \psi, \Delta}$$

the result of applying the rule `andRight` (Fig. 12) on the class invariant can be found in Fig. 13.

The first proof branch The first case is to prove that the balance is non-negative upon method termination (shown in Fig. 13). Since we have an assumption that the class invariant holds at the beginning of the method execution, we know that the method parameter of `deposit` is

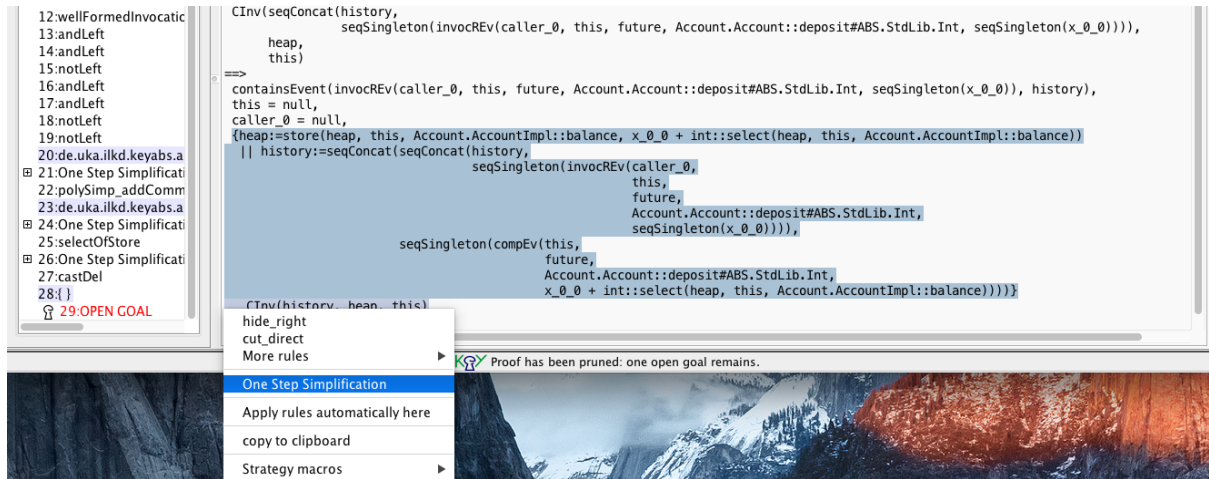


Figure 9: One Step Simplification.

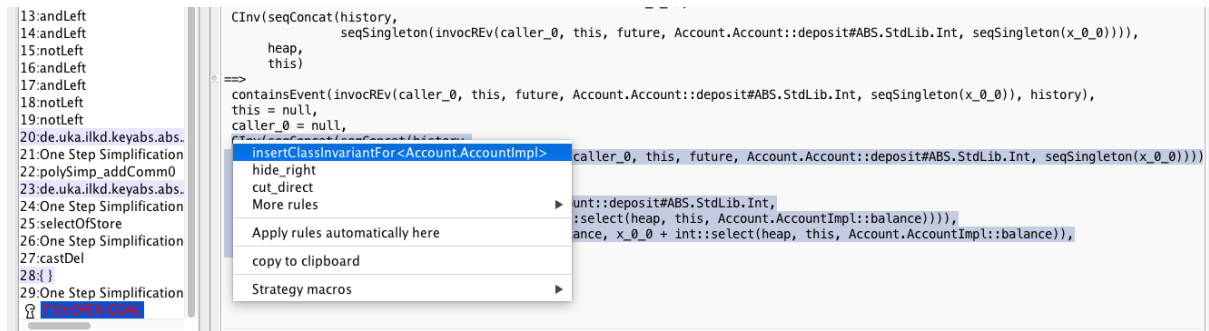


Figure 10: Insert class invariant.

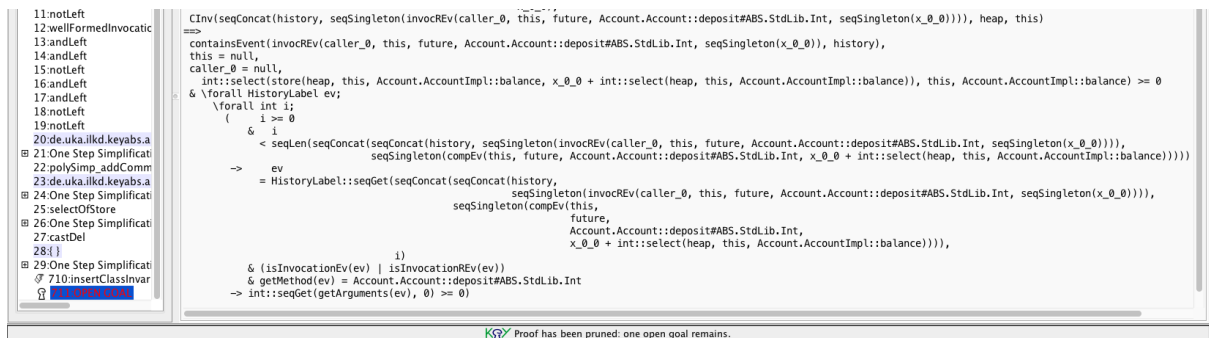


Figure 11: Instantiated class invariant.

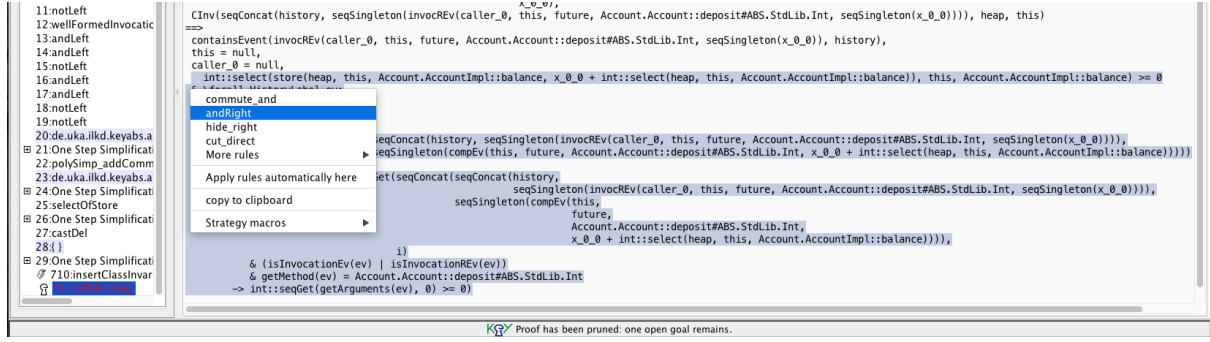


Figure 12: Rule andRight.

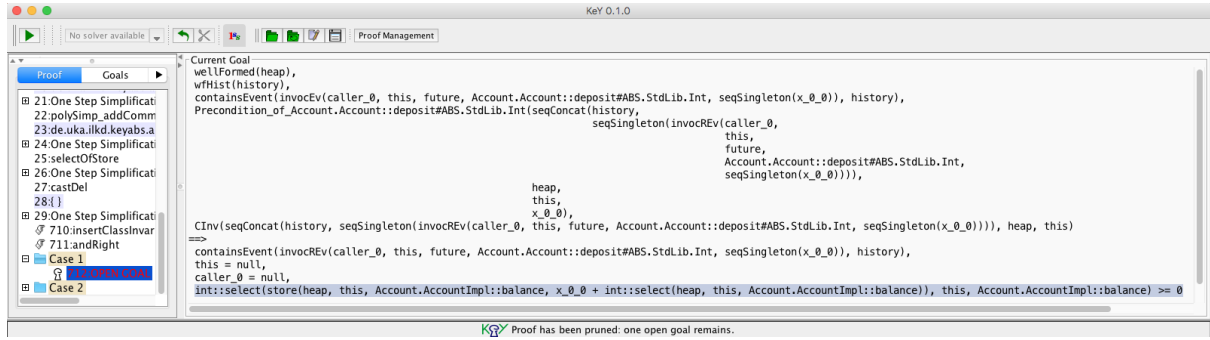


Figure 13: Branching in the proof tree.

non-negative and the balance is non-negative before increasing the value. Based on these two assumptions, we should be able to close this proof branch. Detailed proof steps are described below.

First we apply the rule `insertClassInvariantFor<Account.AccountImpl>` on the class invariant predicate `CInv` at the left side of the implication. The class invariant predicate `CInv` is written as the following:

$$CInv(seqConcat(history, seqSingleton(invocREv(caller_0, this, future, Account.Account :: deposit\#ABS.StdLib.Int, seqSingleton(x_0_0)))), heap, this)$$

This step is presented in Fig. 14. Next we apply the standard first-order-logic rule, i.e., `andLeft` in KeY-ABS, on the class invariant.

$$\text{andLeft} \frac{\Gamma, \phi, \psi \vdash \Delta}{\Gamma, \phi \wedge \psi \vdash \Delta}$$

This rule rewrites the conjunction of two formulae on the left side of an implication into a list of two formulae. This step is shown in Fig. 15.

Now we would like to use the assumption of `amountOfDepositNonNegative`. Since there are quantifiers in `amountOfDepositNonNegative` invariant, we need to instantiate the quantified variables. There are two reasoning rules which can be applied for instantiating quantified variables. One is `allLeft` and another is `allLeftHide`.

$$\text{allLeft} \frac{\Gamma, \forall x. \phi, [x/c] \phi \vdash \Delta}{\Gamma, \forall x. \phi \vdash \Delta} \quad \text{allLeftHide} \frac{\Gamma, [x/c] \phi \vdash \Delta}{\Gamma, \forall x. \phi \vdash \Delta}$$

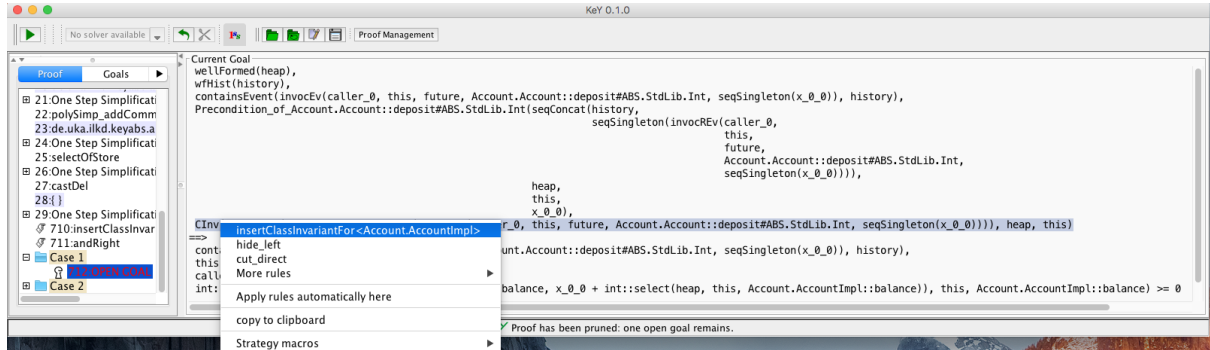


Figure 14: Insert class invariant at the left side of the implication.

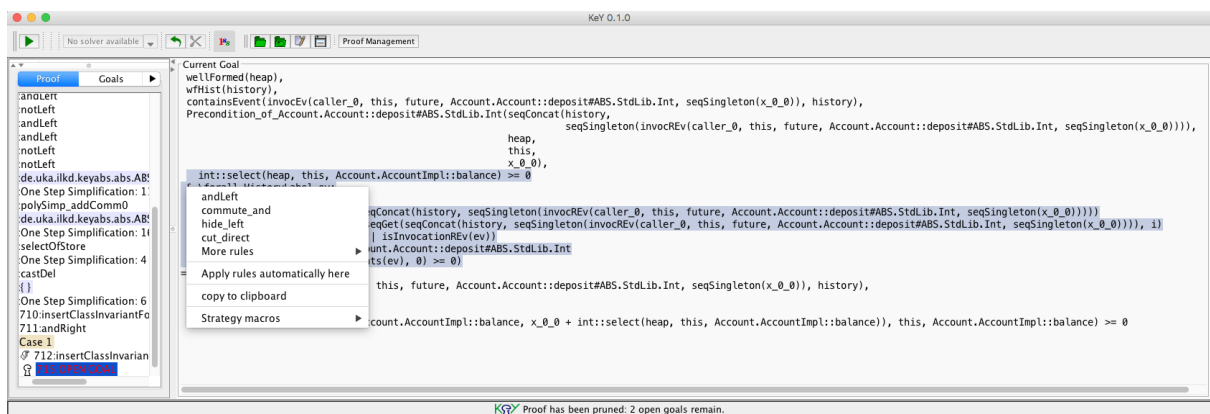


Figure 15: Rule addLeft.

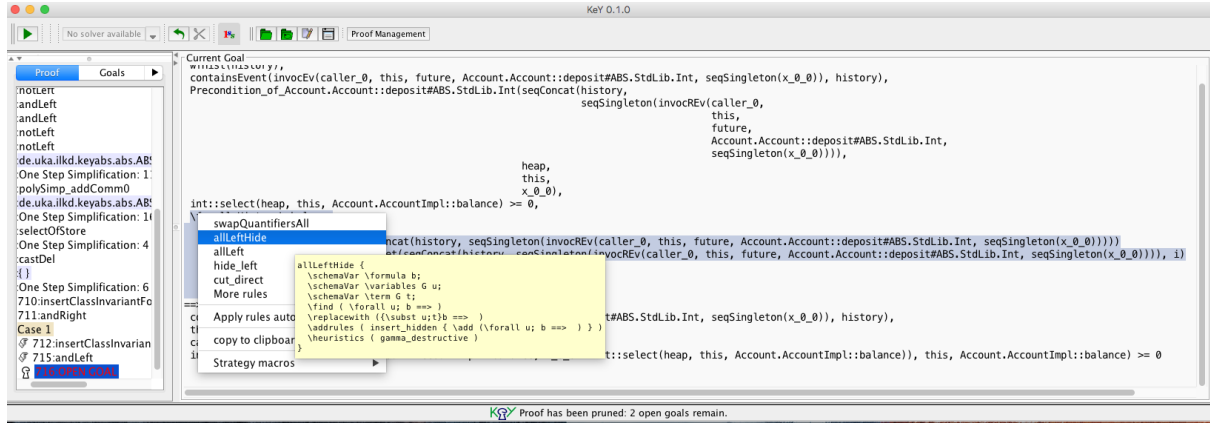


Figure 16: Rule `allLeftHide`.

The difference is that the former one keeps a copy of the highlighted universal quantified formula in addition to an instantiated version. This is the standard reasoning rule for first-order-logic while universal quantifier is at the left side of the implication. To keep the proof formula compact, we choose to apply the reasoning rule `allLeftHide`, which hides the copy of the original universal quantified formula. Note that by applying this rule a popup window will show up. This step is shown in Fig. 16.

KeY-ABS pops up a window for users to type in a symbolic value that the quantified variable should be instantiated to. Users can also select an existing term from the formula, drag and drop it to the blank box in the window to instantiate the variable. But first we need to have an idea about what we can instantiate the variables to. The `amountOfDepositNonNegative` invariant contains two quantifiers: one for *invocation events* or *invocation reaction events* of the `deposit` method, and one for the index of the history sequence locating the event. That means we need to apply `allLeftHide` rule twice. As we know the history at the beginning of the method execution is the concatenation of initial history with an invocation reaction event of the `deposit` method shown below

$$\text{seqConcat}(\text{history}, \text{seqSingleton}(\text{invocREv}(\text{caller_0}, \text{this}, \text{future}, \text{Account.Account} :: \text{deposit}\#\text{ABS.StdLib.Int}, \text{seqSingleton}(x_0_0))))$$

we can instantiate variable `ev` to

$$\text{invocREv}(\text{caller_0}, \text{this}, \text{future}, \text{Account.Account} :: \text{deposit}\#\text{ABS.StdLib.Int}, \text{seqSingleton}(x_0_0))$$

and variable `i` to `seqLen(history)`, which is the length of the history sequence. This index points out where the selected `invocREv` event locates. Note that the index of the first element in the history sequence is 0. These instantiation steps are shown in Fig. 17 and Fig. 18.

Now we can close this proof branch by right-clicking the OPEN GOAL node and choosing Apply Strategy.

The second proof branch The second branch shown in Fig. 19 is to prove that the method parameter of `deposit` is non-negative upon method termination. This can easily be proved because we have an assumption that the method parameter is non-negative at the beginning of method execution. To achieve this, we first apply the reasoning rule `allRight` on the class

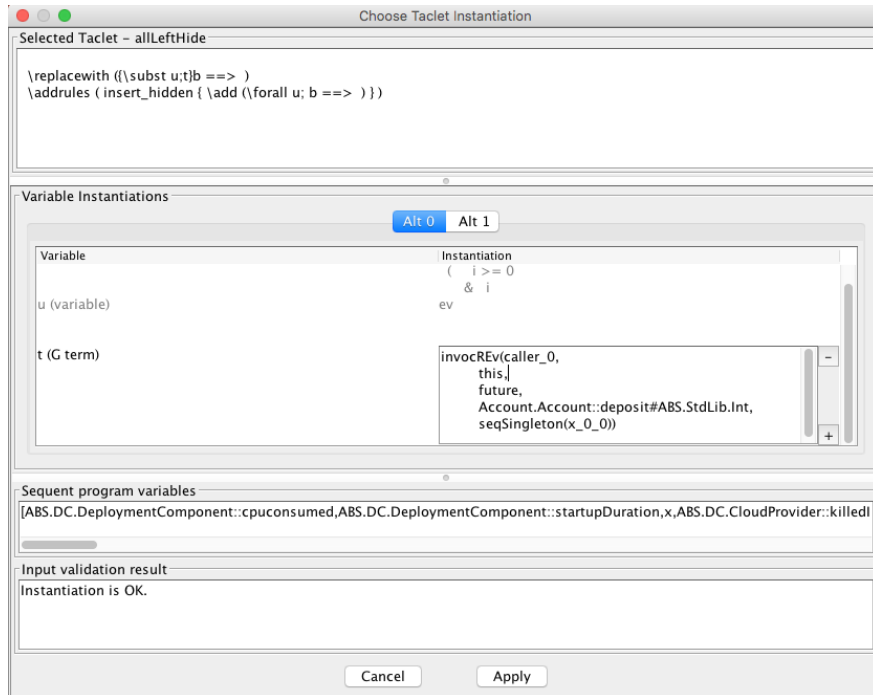


Figure 17: Instantiation of event variable ev.

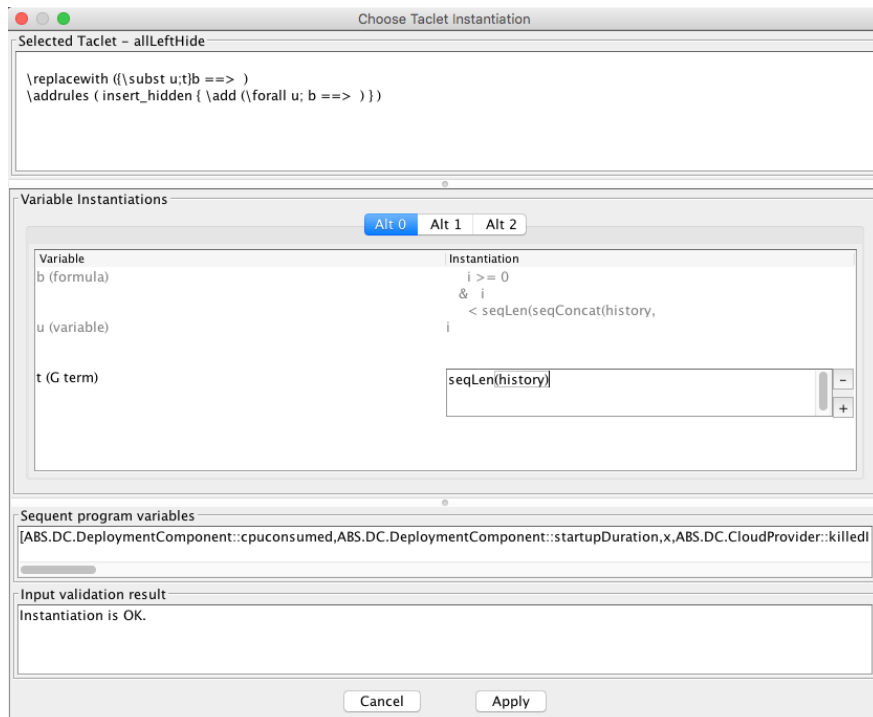


Figure 18: Instantiation of index variable i.

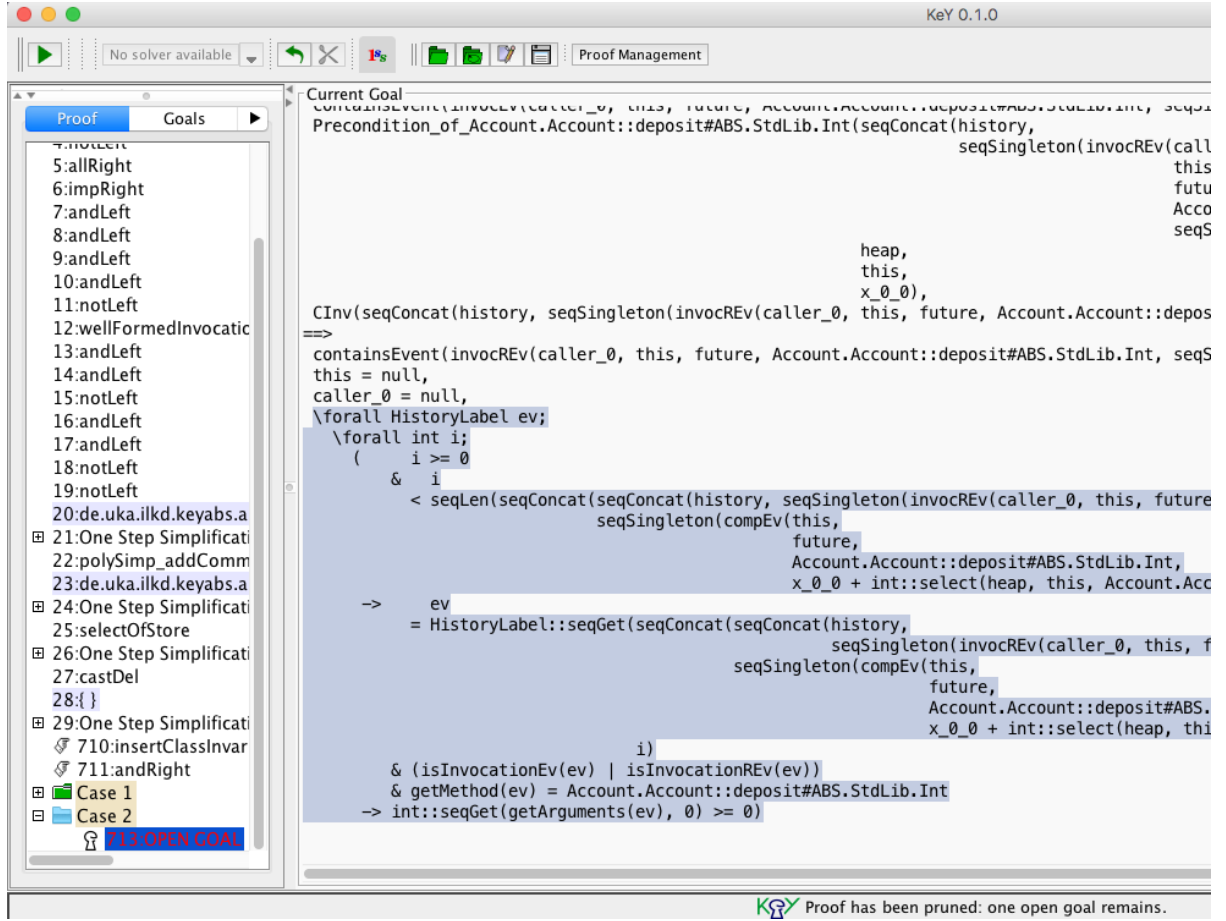


Figure 19: Proof of Case 2.

invariant at the right side of the implication.

$$\text{allRight} \frac{\Gamma \vdash [x/c]\phi, \Delta}{\Gamma \vdash \forall x. \phi, \Delta}$$

This is a standard first-order-logic rule which instantiates universal quantified variables at the right side of an implication to fresh variables. Since there are two universal quantifiers, we need to apply allRight rule twice. This step is shown in Fig. 20. The prover then automatically instantiates the quantified variable ev to a fresh variable ev_0 and variable i to a fresh variable i_0 . The instantiated formula is shown in Fig. 21. Note that the fresh variables created by the rule allRight are not necessarily ev_0 and i_0 . It depends on how many proofs were done before. The subscripts could be different.

Class invariant $CInv$ at the left side of the implication should be expanded, and the rules andLeft and allLeftHide are applied on the expanded class invariant. This procedure is the same as the one for the first proof branch and is shown in Fig. 14 ~ 16. Then we manually instantiate the universal quantified variable ev to the fresh variable ev_0 and variable i to the fresh variable i_0 at the left side of the implication. This step is shown in Fig. 22 and Fig. 23. Note that the subscripts of the automatically generated fresh variables could be different. The manual instantiation of the universal quantified variables at the left side of the implication here should always be consistent with the generated ones at the right side of the implication.

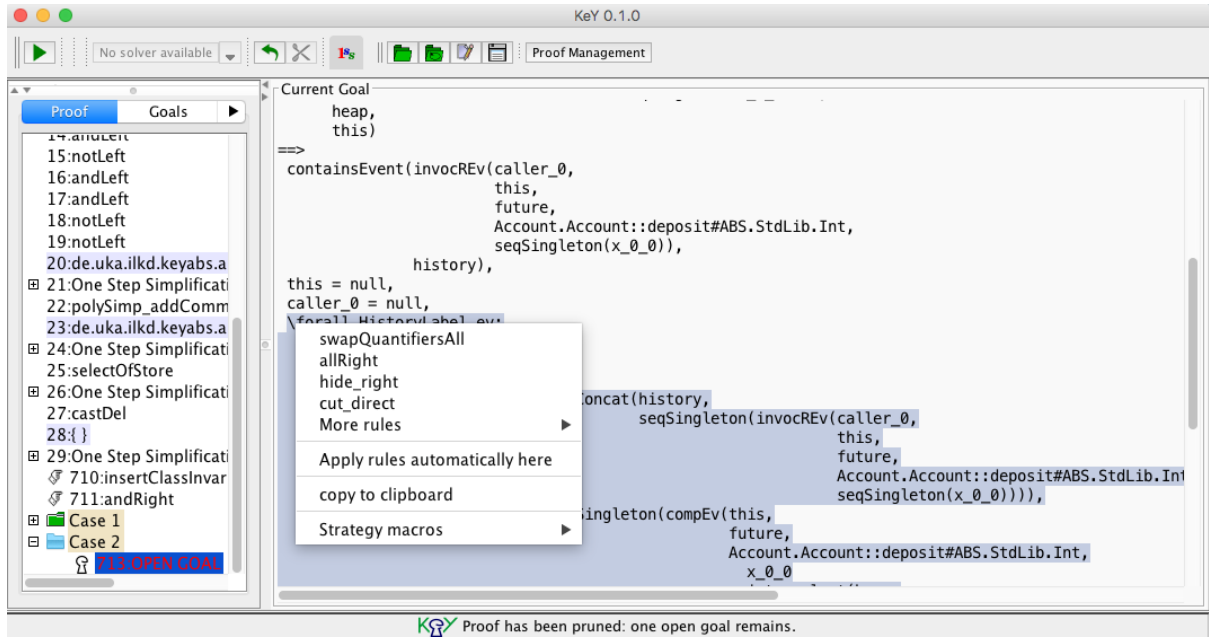


Figure 20: Rule `allRight`.

Finally, we close the whole proof for the `deposit` method by clicking the green triangle button at the top-left corner of the window. The proof result is shown in Fig. 24.

Screencast of the proof At <http://folk.uio.no/crystald/deposit.mov> a screencast showing how to prove the `deposit` method can be downloaded.

Now the readers can try to prove that `withdraw` method also preserves `nonNegativeBalance` and `amountOfDepositNonNegative` such that the `AccountImpl` class preserves all class invariants.

6 Conclusions and Further Reading

This tutorial presents the system workflow for the KeY-ABS theorem prover. It provides the links for readers to download the tool and example. This tutorial is self-contained such that the readers are able to reproduce the proof and obtain the basic knowledge of KeY-ABS. A simple sequential program, a banking account example, is used in this tutorial to teach the basic functionalities of KeY-ABS. For further reading, we recommend the following reading list. The KeY book [1] lays the foundation of KeY-ABS, specifically for the logic system, the symbolic execution engine and the *taclets* language for formulating reasoning rules. History-based reasoning for ABS is presented in [2]. A tool paper of KeY-ABS is in [3]. For more complex verification applications on unbounded concurrent ABS programs, one can read [4].

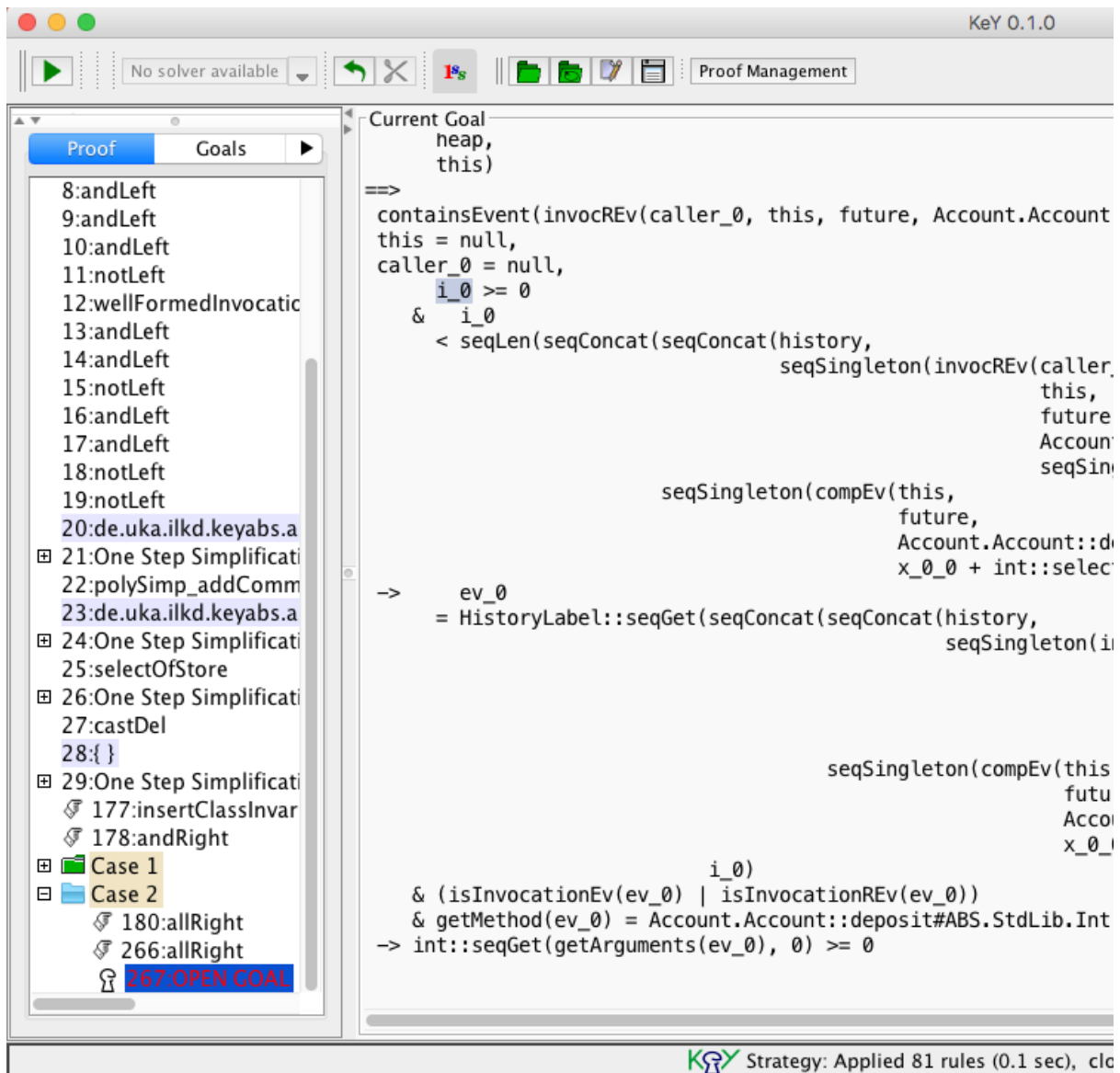


Figure 21: Automatic instantiation.

Choose Taclet Instantiation

Selected Taclet - allLeftHide

```
\replacewith ({\subst u;t}b ==> )
\addrules ( insert_hidden { \add (\forallall u; b ==> ) })
```

Variable Instantiations

Alt 0 Alt 1 Alt 2

Variable	Instantiation
b (formula)	\forallall int i; (i >= 0 & i
u (variable)	ev
t (G term)	ev_0

Sequent program variables

[ABS.DC.DeploymentComponent::cpuconsumed,ABS.DC.DeploymentComponent::startupDuration,x,ABS.DC.CloudProvider::killed]

Input validation result

Rule is not applicable.
Detail:Missing Instantiation:
Row: 2
Instantiation missing for t (G term)

Cancel Apply

Figure 22: Instantiation of ev.

Choose Taclet Instantiation

Selected Taclet - allLeftHide

```
\replacewith ({\subst u;t}b ==> )
\addrules ( insert_hidden { \add (\forallall u; b ==> ) })
```

Variable Instantiations

Alt 0 Alt 1 Alt 2 Alt 3

Variable	Instantiation
b (formula)	i >= 0 & i
u (variable)	i < seqLen(seqConcat(history,
t (G term)	i_0

Sequent program variables

[ABS.DC.DeploymentComponent::cpuconsumed,ABS.DC.DeploymentComponent::startupDuration,x,ABS.DC.CloudProvider::killed]

Input validation result

Rule is not applicable.
Detail:Missing Instantiation:
Row: 2
Instantiation missing for t (G term)

Cancel Apply

Figure 23: Instantiation of i.

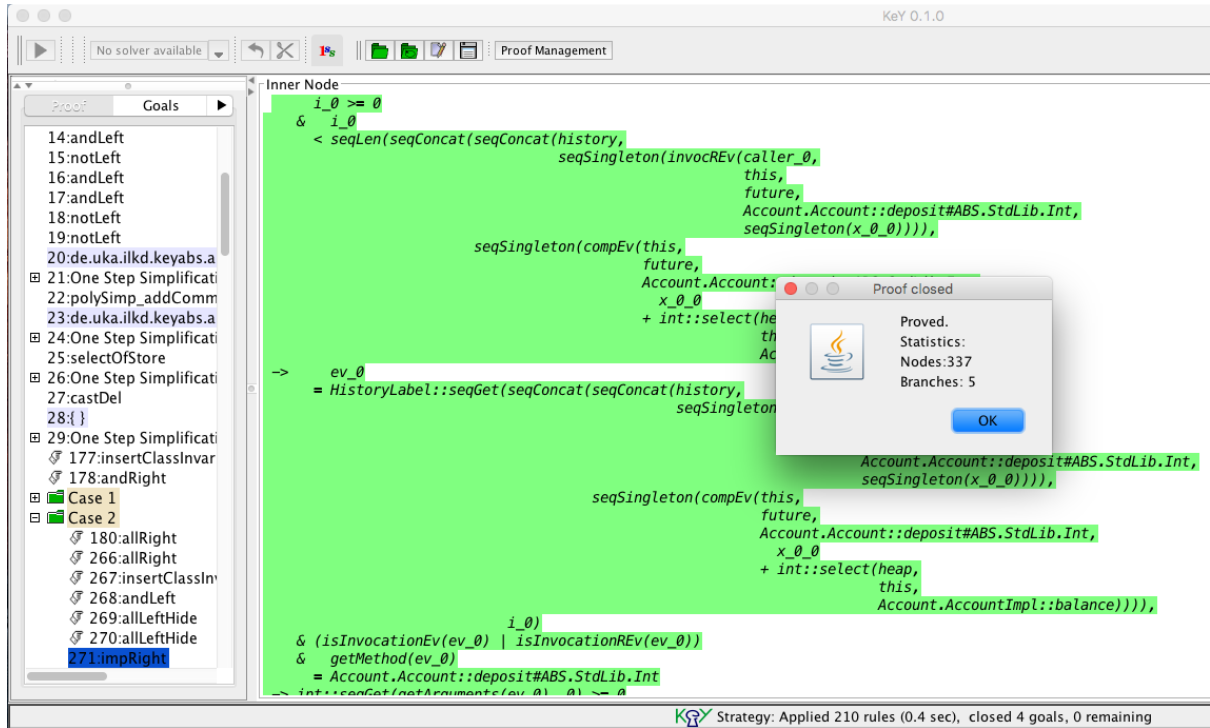


Figure 24: Close the proof for the deposit method.

7 References

1. W. Ahrendt, B. Beckert, R. Bubel, R. Hähnle, P. Schmitt and M. Ulbrich. The KeY Book: Deductive Software Verification in Practice: LNCS 10001. Springer, 2016.
2. C. C. Din and O. Owe. Compositional reasoning about active objects with shared futures. *Formal Aspects of Computing*, 27(3):551 - 572, 2015
3. C. C. Din, R. Bubel, and R. Hähnle. KeY-ABS: A deductive verification tool for the concurrent modelling language ABS. *CADE'15*, LNCS 9195. Springer, 2015.
4. C. C. Din , S. L. T. Tarifa, R. Hähnle, E. B. Johnsen. History-Based Specification and Verification of Scalable Concurrent and Distributed Systems. *ICFEM'15*, LNCS 9407. Springer, 2015.