A Proof Repository for Formal Verification of Software

Michael Franssen

Eindhoven University of Technology, Dept. of Mathematics and Computer Science, P.O. Box 513/HG5.36, 5600 MB Eindhoven, The Netherlands

Abstract

We present a proof repository that provides a uniform theorem proving interface to virtually any first-order theorem prover. Instead of taking the greatest common divisor of features supported by the first-order theorem provers, the design allows us to support any extension of the logic that can be expressed in first-order logic. If a theorem prover has native support for such a logic, this is exploited. If the prover has no such support, the repository automatically uses the first-order encoding of the extension. A built-in proof assistant is provided that allows the user to manually guide the proving process when all provers fail to prove a theorem. To prove sub-theorems, the proof assistant is able to use the repository's full capabilities. The repository also maintains a database of proven theorems. When a requested theorem has been proved before, the result from the database is re-used instead of reconstructing the proof all over again. To test the repository, we constructed a tool for static verification of a basic programming language. This language is also described in this paper.

Keywords: Theorem proving, Program verification, Software tools

1. Introduction

Many tools for program verification use a layered model [1, 2]. The bottom layer exists of a logic, usually supported by some (semi)automatic tool. On top of this there is a layer of an intermediate language in which programs to be verified will be expressed. Typically, this intermediate language is simple, since it will not be used to directly write programs, but only as a stepping stone to verify programs in a more complex language. The programs in the intermediate language are explicitly annotated with assertions that express the program properties to be verified. For annotated programs written in the intermediate language a verification condition generator will calculate a set of logical conditions that must hold in order for the program to be correct. The upper layer consists of the actual programming language. Programs written in this language will first be translated into the intermediate language. During this translation, many complex features of the input language are expressed in the much simpler features of the intermediate language. Also, annotations are inserted into the intermediate program that claim

Email address: m.franssen@tue.nl (Michael Franssen)

absence of null-dereferences, array-index-out-of-bounds, etc. If the actual program is annotated with additional specifications, these specifications are translated into assertions too.

In order to obtain a high degree of automation, most tools select a single theorem prover to construct the required proofs and then tweak their proof obligations towards this theorem prover (ESCJava and SpecSharp use Simplify [1] and Perfect developer [3] has its own built-in theorem prover). When a proof fails, one can only add assumptions or tweak the program to get different verification conditions, hoping that these can be proved. Also, none of the aforementioned tools keep track of the constructed proof. So even if only a small part of the code changed, all the verification conditions are sent to the prover again. Moreover, during the development of a program, the context of definitions and assumptions grows larger. Proofs of theorems that were already proved when this context was small will take much more time to reconstruct in a larger context.

In this paper, we present a proof repository that provides the following services: (1) A rich logic to denote program specifications, definitions and abstract datatypes. (2) A single interface to connect to a wide range of automated theorem provers. This way, the programming tool is not restricted to using just one. (3) An interactive proof assistant that allows the user to guide the proof by manually constructing it himself as much as necessary. Instead of guessing assertions that might help the prover, this allows the user to gain insight in *why* a proof fails and what lemmas are needed to complete the proof. (4) A database of completed proofs that avoids reconstructing proofs for completed theorems all over again. This also implies that the allotted time to prove the verification conditions can be spent entirely on new or altered verification conditions.

To test the repository, we also introduce a basic programming language designed for modular verification. This programming language is a small extension of the guarded command language which is used, among others, by ESC/Java as an intermediate language to express programs to be verified. Nevertheless, a number of tree and list algorithms can be conveniently expressed in this language, since it exploits the repository's inductive type definitions.

2. Tool Structure

The structure of the entire toolkit described in this paper is depicted in Figure 1. The user enters a program in the text editor. This program is read by a JavaCC generated parser. The result is type-checked and then verification conditions are computed (see Section 4). Proofs for the verification conditions are obtained from the proof repository.

The repository first checks for every request whether or not the theorem exists in the database. If so, the stored result is returned. If not, one or more external theorem provers are launched (either remotely or on the same machine). For each different prover, the theorem is first translated into the native input language of that prover by a connector object. The order and time limits for the theorem provers can be configured by the user. If a proof is found, it is stored in the database for future re-use and sent back to the programming tool. If no proof is found, the proof assistant is launched to let the user manually construct a proof. The proof assistant lets the user select tactics that invoke small proof-steps. At any point the user may request a proof for a sub-theorem from the repository. In practice the proof-assistance is mainly used to perform a few initial steps and indicate where induction should be used to complete a proof. Automated theorem provers typically cannot handle proofs that require induction.

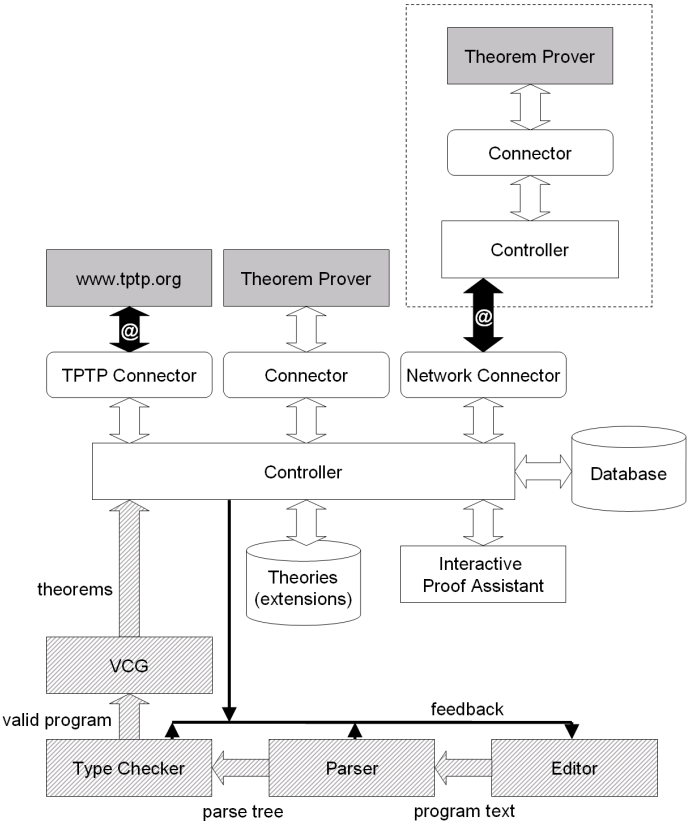

Figure 1: Architecture of the toolkit: Grey boxes represent external systems. White boxes are part of the repository. Textured boxes are part of the programming toolkit. Arrows with a @ sign represent internet connections.

Although the main contribution of our tool is the proof repository with support for inductive types and recursive definitions, we start in Section 3 by describing the programming language used to test the repository. This also introduces the notation for formulas and inductive definitions used by the repository. In Section 4, we explain how to compute the verification conditions. This demonstrates the kind of theorems for which the repository was constructed. We then describe the modules that make up the proof repository in Section 5. In Section 6 we explain how we handle inductive types and recursive function definitions. We conclude in Section 7.

3. The Programming Language

The programming language employed by our tool is an example language to show the usefulness of the repository. It is designed to be able to express some basic programming problems and to support simple generation of verification conditions.

The language is defined by the grammar in Figure 2. It is based on the guarded command language of Dijkstra, since he first proposed the weakest precondition calculus to establish formal correctness of programs. As a result, we support non-deterministic guarded **if** and **do** statements and multiple assignments. In order to make the language a bit more powerful and a more realistic test case for the proof repository, we added procedures, inductive datatypes and recursive specification functions. We will first discuss the main features of this language informally. In Section 4 we discuss how the tool computes the verification conditions.

Prog	::=	**program** *Id*; *Decl*∗ **var** *Idlt* {**post:** *Pred*} *Stat*
Decl	::=	**procedure** *Id*(*Idlt*, **var** *Idlt*) {**pre:** *Exp* **post:** *Exp* [⇓]} *Stat*
		| **type** *Id* **is** *Id*(*Idlt*) [□ *Id*(*Idlt*)] ∗ **end**
		| {**define** *Id*(*Idlt*) : *Id* **as** (*Match* | *Exp*) **end**}
Stat	::=	*Stat*; *Stat* | **skip** | *Idl*[, *Idl*]∗ := *Exp*[, *Exp*]∗ | |[**var** *Idlt*; *Stat*]|
		| **if** *GStats* **fi** | {**inv:** *Exp* [**dec:** *Exp*]}**do** *GStats* **od**
		| **match** *Exp* **with** *GStats* **end**
		| *Id*(*Expl*)
GStats	::=	*GStats* □ *GStats* | *Exp* → *Stat*
Match	::=	**match** *Id* **with** *Id*(*Idl*) → *Match*[□*Id*(*Idl*) → *Match*] ∗ **end**
Exp	::=	*Exp* = *Exp* | (∀*Id* : *Id*.*Exp*) | (∃*Id* : *Id*.*Exp*) | *Id*[(*Expl*)]
		| *Exp* ∧ *Exp* | *Exp* ∨ *Exp* | ¬*Exp*
		| *Exp* + *Exp* | *Exp* − *Exp* | *Exp* ∗ *Exp*
Expl	::=	[*Exp*[, *Exp*]∗]
Id	::=	⟨*identifier*⟩
Idl	::=	[*Id*[, *Id*]∗]
Idlt	::=	[*Id* : *Id*[, *Id* : *Id*]∗]

Figure 2: The grammar for the example programming language

3.1. Scope rules

Basically, there are two contexts: The program context and the specification context. The program context contains all the variables and definitions that are available to program statements. These are written outside the curly brackets. The specification context contains additional definitions that are available to program specifications (pre and post conditions, invariants, function definitions, etc.). In specifications, all definitions and variables of the program context and the specification context can be used.

Parameters of definitions and procedures only exist within the body of the definition or procedure. Local variables can be declared between |[and]|. These variables only exist within this range. Local variable blocks can be nested. Within procedures, only variables in the parameter list and locally declared variables exist. Global variables are not supported in order to avoid aliasing problems.

The main program can only alter the variables introduced in its own **var** part and locally declared variables.

3.2. Inductive Types

Our language supports inductive type definitions called strictly positive types (See [4]). Given the syntactic restrictions of our language, all types that can be denoted are strictly positive and hence, well defined. Hence, there is no need to discuss the theory of strictly positive types in this paper. Variables of inductive types have value semantics (sometimes called copy semantics).

There is no need for built-in types, but the language assumes fixed definitions of *bool* and *int*. These types are additionally supported by basic operators and universal and existential quantifiers $\forall$ and $\exists$. Moreover, guards of **if** and **while** statements must be of type *bool* and may not contain any quantifiers. Also, we have the polymorphic boolean relation = between elements of any datatype.

All datatypes other than *bool* and *int* must be defined within the program itself. For example, lists of integers can be defined by:

type *list* **is** *empty*() □ *cons*(*x* : *int*, *tail* : *list*) **end**

3.3. Specification Functions

Functions can be defined only within the specification context, since the type of definitions allowed does not guarantee computability of the function. Once a function is defined, a procedure can be written that actually computes it. When this procedure is proved correct, the function apparently was computable. For instance, the function to compute the length of a list of integers is written as:

```
{define length(L : list) : int as
   match L with
        empty()        →  0
   □    cons(x, tail)  →  length(tail) + 1
   end
 end}
```

Since function definitions can only occur in the declaration part of a program, they can never refer to any program variables. Therefore, the match-pattern (following the keyword **match**) must be a variable referring to one of the function parameters. Following the **with** keyword, all constructors of the recursive type of this parameter are listed (including dummies representing the constructor's parameters) followed by → and a result-expression. This result-expression can be a direct expression or a match-expression. Note that by definition of the grammar, match-expressions are no ordinary expressions, since they cannot occur as sub-expressions at arbitrary places. They are only allowed when providing specification function definitions.

3.4. Match-Statements

To write a procedure implementing a specification function, we support match-statements. Although they are similar to match-expressions there are important differences: (1) The match-pattern can be any expression instead of just a variable. (2) Result-expressions are replaced by statements (3) Match-statements are ordinary statements. Semantically, a match-statement is like an if-statement where all the guards have a specific form and introduce a set of local variables to the corresponding branch.

3.5. Procedures

Once a procedure $p(a_1, \ldots, a_n, \textbf{var}\ x_1, \ldots x_m)\{\textbf{pre:}\ P\ \textbf{post:}\ Q\}\ S$ is defined, it can be called from all succeeding procedures and the main program. S can only refer to its parameters and locally declared variables. In order to avoid aliasing problems, all arguments of variable parameters in a procedure call must be different. Also, expressions used for value parameters in a procedure call may not depend on any of the variables parameter arguments. We denote p's precondition by $p.\textbf{pre}$ and p's postcondition by $p.\textbf{post}$.

3.6. Optional termination proofs

It is up to the programmer whether or not termination of (part) of the program has to be proved. If a bound expression is provided for a loop (after the keyword **dec**), the required termination verification conditions are generated. Since procedures cannot be recursive, it is sufficient to add the optional $\Downarrow$ to the postcondition. Terminating procedures may only call other terminating procedures and all loops within its body must terminate. Loops within terminating loops must also terminate. The tool checks whether the required bound expressions and $\Downarrow$ are present in these cases.

4. Verification Conditions

The tool parses a string according to the given grammar using a JavaCC generated parser. The resulting tree is type-checked to find out if the program is valid. If so, verification conditions are generated based on a weakest precondition approach. That is, for any statement and postcondition, we compute a weakest precondition that must hold in the initial state, to guarantee that the postcondition holds in any final state. Apart from the weakest precondition, a set of side-conditions are computed that also must hold in order for the program to be correct. Based on this weakest precondition, the verification conditions for procedures and the main program are computed.

Every verification condition generated is labeled with a name that clarifies the property it expresses. In the remainder of this paper, an overlined term represents a comma-separated list of terms of the appropriate length.

Definition 1. *Weakest Preconditions*
The weakest precondition $wp(S, Q)$ for statement S and postcondition Q is defined as:

$$wp(S_1; S_2, Q) = wp(S_1, wp(S_2, Q))$$

$$wp(\textbf{skip}, Q) = Q$$

$$wp(\overline{x} := \overline{e}, Q) = Q[\overline{x} := \overline{e}]$$

$$wp(|[\textbf{var}\overline{x : T}; S\,]|, Q) = (\forall \overline{x : T}.wp(S, Q))$$

$$\begin{array}{lll} wp(\ \textbf{if} & G_1 \to S_1 & = G_1 \Rightarrow wp(S_1, Q) \\ \quad \Box & \ldots & \quad \wedge \ldots \\ \quad \Box & G_n \to S_n & \quad \wedge G_n \Rightarrow wp(S_n, Q) \\ \quad \textbf{fi} & , Q) & \quad \wedge (G_1 \vee \ldots \vee G_n) \end{array}$$

$$\begin{array}{lll} wp(\{\textbf{inv:}I\ \textbf{dec:}B\} & & = I \\ \quad \textbf{do} & G_1 \to S_1 & \\ \quad \Box & \ldots & \\ \quad \Box & G_n \to S_n & \\ \quad \textbf{od} & , Q) & \end{array}$$

Also, the following side-conditions have to be proved:

invariance:	$I \wedge G_i \Rightarrow wp(S_i, I)$ *for any* $1 \leq i \leq n$
finalisation:	$I \wedge \neg G_1 \wedge \ldots \wedge \neg G_n \Rightarrow Q$
boundness:	$I \wedge G_i \Rightarrow 0 \leq B$ *for any* $1 \leq i \leq n$
progress:	$I \wedge G_i \wedge B = C \Rightarrow wp(S_i, B < C)$ *for any* $1 \leq i \leq n$ *where* C *is a fresh constant*

boundness and progress only have to be proved if the optional B is provided

$$\begin{array}{lll} wp(\textbf{match}\ E\ \textbf{with} & & = E = C_1(\overline{v_1}) \Rightarrow wp(S_1, Q) \\ \quad C_1(\overline{v_1}) & \to S_1 & \quad \wedge \ldots \\ \quad \Box & \ldots & \quad \wedge E = C_n(\overline{v_n}) \Rightarrow wp(S_n, Q) \\ \quad \Box & C_n(\overline{v_n}) \to S_n & \\ \textbf{end}, Q) & & \end{array}$$

$$wp(p(\overline{e}, \overline{v}), Q) = p.\textbf{pre}[\overline{a}, \overline{x} := \overline{e}, \overline{v}]$$

Also, the following side-condition has to be proved:

correct use of p:	$p.\textbf{post}[\overline{a}, \overline{x} := \overline{e}, \overline{v}] \Rightarrow Q$

Note that in order to prove a side condition with free variables, we actually prove its universal closure.

Using the *wp* function from Definition 1 directly would yield a very rigid system. A procedure's postcondition would always have to match the entire postcondition of a procedure call. Usually, a procedure is used to satisfy only part of the postcondition.

To relax the verification conditions, we will split a postcondition into two parts: A part that is altered by the program and the part that is independent of the program. In order to define this split, we first provide some definitions.

$FV(P)$ denotes the free variables in P and is defined as usual.

Definition 2. *Altering Variables*
Given a program S, the set of variables that can possibly be altered during execution of S is computed by the function $AV : Stat \to \mathcal{P}(\mathcal{V})$. AV is defined as:

$$\begin{array}{lcl}
AV(S_1; S_2) & = & AV(S_1) \cup AV(S_2) \\
AV(\textbf{skip}) & = & \emptyset \\
AV(\overline{x} := \overline{e}) & = & \{\overline{x}\} \\
AV(|[\textbf{var } \overline{x} \bullet S]|) & = & AV(S) \setminus \{\overline{x}\} \\
AV(\textbf{if } G_1 \to S_1 \Box \ldots \Box G_n \to S_n \textbf{ fi}) & = & (\bigcup_{1 \le i \le n} AV(S_i)) \\
AV(\{\textbf{inv: } I\}\textbf{do } G_1 \to S_1 \Box \ldots \Box G_n \to S_n \textbf{ od}) & = & (\bigcup_{1 \le i \le n} AV(S_i)) \\
AV(p(\overline{e}, \overline{x})) & = & \{\overline{x}\}
\end{array}$$

Note that AV is computed from the syntax of the statement. Stated simply: $AV(S)$ lists all free variables that occur in S either at the left-hand side of an assignment statement or as an argument for a var-parameter of a procedure call.

Definition 3. *Keeps and Alters*
Let $Q = Q_1 \wedge \ldots \wedge Q_n$ be a predicate and let S be a statement. Without loss of generality, assume that $FV(Q_i) \cap AV(S) = \emptyset$ for $0 \le i < j$ and $FV(Q_i) \cap AV(S) \neq \emptyset$ for $j \le i < n$ for some j (if needed, we can re-order the conjuncts). We define Keeps and Alters as follows:

$$\begin{array}{lcl}
Keeps(S, Q) & = & Q_1 \wedge \ldots \wedge Q_{j-1} \\
Alters(S, Q) & = & Q_j \wedge \ldots \wedge Q_n
\end{array}$$

Since $Keeps(S, Q)$ does not depend on any variable that can be changed by S, it must hold in the precondition iff it has to hold in the postcondition. We will use this to change the computation of *wp* for loops and procedure calls (changing *wp* for other statements is not necessary).

Definition 4. *wp with Keeps and Alters*
We use S to denote the entire statement, K to denote $Keeps(S, Q)$ and A to denote $Alters(S, Q)$. wp is redefined for loops and procedure calls by:

$wp(\{\textbf{inv:}I\ \textbf{dec:}B\} \quad = \quad K \wedge I$
$\quad \textbf{do} \quad G_1 \to S_1$
$\quad \Box \quad \ldots$
$\quad \Box \quad G_n \to S_n$
$\quad \textbf{od} \quad , Q)$

The following side-conditions have to be proved:

invariance:	$K \wedge I \wedge G_i \Rightarrow wp(S_i, I)$ *for any* $1 \le i \le n$
finalisation:	$K \wedge I \wedge \neg G_1 \wedge \ldots \wedge \neg G_n \Rightarrow A$
boundness:	$K \wedge I \wedge G_i \Rightarrow 0 \le B$ *for any* $1 \le i \le n$
progress:	$K \wedge I \wedge G_i \wedge B = C \Rightarrow wp(S_i, B < C)$ *for any* $1 \le i \le n$ *where C is a fresh constant*

boundness and progress only have to be proved if the optional B is provided

$wp(p(\overline{e}, \overline{v}), Q) \quad = \quad K \wedge p.\textbf{pre}[\overline{a}, \overline{x} := \overline{e}, \overline{v}]$

The following side-condition has to be proved:

correct use of p:	$K \wedge p.\textbf{post}[\overline{a}, \overline{x} := \overline{e}, \overline{v}] \Rightarrow Q$

Computing weakest preconditions is not sufficient to obtain all verification conditions for a program. Therefore, we will define the function *VC*, based on *wp* to compute verification conditions for procedures and the main program.

Definition 5. *Verification conditions*
The function VC computes the verification conditions that must be proved in order to establish the correctness of an entire program. It is defined as:

$$
\begin{array}{lcl}
VC(\ \textbf{program}\ n & = & VC(Proc) \\
\quad Proc & & \quad \cup \{\textit{correctness of } n{:}(\forall_{\overline{x}}.wp(S,Q))\} \\
\quad \textbf{var}\ \overline{x} & & \\
\quad \{\textbf{post:}\ Q\} & & \\
\quad |[\,S\,]|\ ) & & \\
 & & \\
VC(\ \textbf{procedure}\ p(\overline{a}, \textbf{var}\ \overline{x}) & = & \{\textit{correctness of } p{:}(\forall_{\overline{a},\overline{x}}.P \Rightarrow wp(S,Q))\} \\
\quad \{\textbf{pre:}\ P\ \textbf{post:}\ Q\} & & \\
\quad |[\,S\,]|\ ) & &
\end{array}
$$

Proving all side conditions obtained by computing *wp* and the conditions computed by *VC* establishes full correctness of the entire program.

Example 1. *power*
By means of a hello world example, we give a program to efficiently compute a^b on natural numbers. In this example, the power function is defined and implemented. Since recursive function definitions are only available in specifications, users are forced to call the procedure, which (in this case) is much more efficient.

```
...
{define pow(a, b : nat) : nat as
  match b with
        0      → 1
   □    suc(x)→ a * pow(a, x)
  end
end}

procedure power(A, B : nat, var p : nat)
{pre: true post: p = pow(A, B) ⇓}
|[var a, b : nat;
  p, a, b := 1, A, B;
  {inv: p * pow(a, b) = pow(A, B) dec: b}
  do b > 0 →
     if   even(b)  → a, b := a * a, b div 2
     □    ¬even(b)→ p, b := p * a, b − 1
     fi
  od
]|
...
```

The generated verification conditions are:

correctness of power:	$true \Rightarrow 1 * pow(A, B) = pow(A, B)$
finalisation:	$p * pow(a, b) = pow(A, B) \wedge \neg(b > 0) \Rightarrow p = pow(A, B)$
invariance:	$p * pow(a, b) = pow(A, B) \wedge b > 0 \Rightarrow$
	$even(b) \Rightarrow p * pow(a * a, b\ div\ 2) = pow(A, B)$
	$\wedge\neg(even(b)) \Rightarrow p * a * pow(a, b - 1) = pow(A, B)$
	$\wedge(even(b) \vee \neg(even(b)))$
progress:	$p * pow(a, b) = pow(A, B) \wedge b > 0 \Rightarrow$
	$even(b) \Rightarrow b\ div\ 2 < b \wedge \neg(even(b)) \Rightarrow b - 1 < b$
boundness:	$p * pow(a, b) = pow(A, B) \wedge \neg(b > 0) \Rightarrow 0 \leq b$

For a long period of time in the Eindhoven Computer Science curriculum students were taught to manually derive programs from specifications. This involved (amongst other skills) constructing and proving verification conditions like the ones in this example. The proofs themselves are an order of magnitude larger than the program being derived. Hence, the chance of mistake in the manually constructed proofs was at least as big as the chance of a mistake in the constructed program.

Apart from the convenience of automatic proof construction, tool support also provides more certainty about the absence of errors in the proofs. In fact, for somewhat larger programs tool support is mandatory, since manually constructing the verification conditions becomes infeasible, let alone proving them.

The verification conditions from the example can be sent directly to the proof repository. In order to solve them, induction will be needed.

5. A Modular Repository

The repository's task is to provide proofs for theorems on request. How it does this should not be the concern of the client application posing the requests. Hence, basically, the proof repository acts as a theorem prover. All verification conditions mentioned in Section 4 can be handled by our repository. Also, the inductive type definitions and recursive function definitions are passed as such.

5.1. The Architectural Modules

The basic architectural design consists of a number of modules: A connector, which connects an external automated theorem prover to the system; A proof assistant, to interactively construct proofs; A database, which stores proofs that have already been constructed; and a controller, which connects all components in a configurable way to the user application.

The user application communicates with the controller and asks for the proof of a theorem. If the controller finds the theorem in the database, it is returned to the application. If not, one or more theorem provers are launched to construct a proof. A connector component is used to translate the theorem into the format of the theorem prover. If a proof is found, it will be stored in the

database and returned to the application. If no proof is found, the proof assistant can be used to manually construct one or the application is notified.

In the following subsections, we describe the components in more detail. We assume that a proof request from the user application has the form $\Gamma \vdash P$, where Γ is the context of definitions and assumptions and P is the theorem to be proved in this context.

5.1.1. Connector

A connector is responsible for connecting an external theorem prover to the repository. In general it takes a request $\Gamma \vdash P$ and translates it into the form of the external prover. It then launches the prover to construct a proof and translates the output into the internal representation of the repository. We provide a separate connector for each prover, but all connectors provide the same interface.

Also, a connector solves the problem of slight differences in the logic supported by different provers. For instance, Simplify [5] offers internal support for arithmetic, like the operators $+$, $-$ and $*$, while Spass [6] and E [7] do not. We could neglect any extensions and only support first-order predicate logic with equalities, which is supported by all theorem provers considered. However, since built-in support for extensions is usually more efficiently than the corresponding axiomatizations, this would weaken our system.

Therefore, our repository supports any extension to the basic logic that can be defined by axiomatizations in first-order predicate logic. A connector recognizes the use of an extension and chooses to either use extensions of the external theorem prover or to provide the corresponding axiomatization. The axiomatization of a theory is given once in the internal representation of our repository and translated by a connector when needed.

As a result of this approach, our repository provides a very rich language to the user application. For example: One often needs lists, sets, stacks, etc. These can easily be axiomatized in first-order logic, but many basic theorems about them are also needed. Since not all theorem provers support these datatypes, one might be tempted to choose one specific prover at an early stage. By using the repository instead, one can use all extensions and connect to different provers as needed. Different provers have different specialties, but the user application can use them all.

Currently, the repository provides native support for the theorem provers Spass [8], Simplify [5], Z3 [9], and any theorem prover that accepts the TPTP [10] input format (like E-prover [7]). Also, through the TPTP webservice, more than 50 theorem provers (including different versions) are directly available. In case of Simplify and Z3, the repository uses their internal support for integer numbers instead of their translations to first-order axioms. This includes the operators $+$, $-$ and $*$.

How the translated theorem is sent to the prover depends on the interface provided by the external prover. Currently, a file is generated that contains the theorem in the external provers native format. The theorem prover is then launched as an external process on this file. The resulting output is parsed and included in the result. In case of the TPTP webservice, the file is sent over the internet to the TPTP servers and no external process is started.

5.1.2. Proof Assistant

First-order logic is semi-decidable, meaning that although every valid theorem can (in theory) be proved automatically, an invalid theorem might cause an infinite proof-attempt. Hence, if after a given amount of time no proof has been found, one cannot distinguish between the theorem being incorrect or the amount of time being insufficient. A theorem prover therefore hardly ever gives other reasons than 'insufficient resources' (including time) if it fails to find a proof.

When the automatic theorem provers fail (for example, if induction is needed), it is necessary to (partially) construct a proof manually. For this, an interactive proof assistant is provided (an extension of the prover used in Cocktail [11]). In this proof assistant, single proof steps called tactics are used to split the main theorem in smaller sub-goals. These sub-goals can in turn be passed to the external automated theorem provers to complete the proof. If needed, the entire proof can be constructed manually by the interactive prover.

5.1.3. Theories

The repository supports first-order logic with equalities, because it (1) is a well-known logic supported by many systems, (2) is at least semi-decidable and (3) is powerful enough to express meaningful theorems.

Several systems exist that have more efficient native support for theories often expressed in first-order logic (e.g. [5] has support for integer numbers and [12] is specialized in Clean programs). Since we want to exploit this special support, any theory that can be expressed in first-order logic is added as a theory module in the proof repository. A theory module consists of the first-order definitions and theorems needed to define this theory for external provers that do not natively support it. When a connector translates a theorem to an external prover, this theory module is added to the translation only if the external prover has no native support for the theory. Otherwise, the translation will exploit the native support, yielding better performance.

The theories module is not really a module as such. It is more like a library of translations from logic extensions to basic first-order logic that is used by connectors to deal with logic extensions that are not supported natively by the external theorem prover.

5.1.4. Database

The repository provides a database in which proven theorems can be stored for re-use. Instead of storing each variant of a theorem, a normal form of the theorem is stored. When searching the database, we do not look for exactly the same theorem, but for a theorem which is "more general", i.e. a theorem that implies the theorem we seek. As a result, renaming program variables or adding assumptions does not invalidate previous results. Also, obviously equivalent formulas are considered equal (e.g. $P \Rightarrow Q$ and $\neg P \vee Q$). The normal form used and the notion "more general" is defined below.

We first define a Database Normal Form (DbNF) that puts all conjuncts, disjunct and universal quantifications together and eliminates implications and existential quantifications. This will allow searching the database in such a way that the exact order of these conjuncts, disjuncts and universal quantifications is irrelevant. The DbNF of a formula is computed as follows:

1. We start by computing the negation normal form of the formula, thereby eliminating $\Rightarrow$ and putting negations only in front of atomic formulas.
2. Next we replace all $\exists$-quantifications by skolem-functions. Equivalence of formulas is no then longer dependent on where exactly the $\exists$ formula is inserted. That is, $\exists x.(P(x) \wedge Q)$ and $(\exists x.P(x)) \wedge Q$ become equal because they both have the skolem form $P(s()) \wedge Q$, where s is the skolem function replacing x. There are several ways to skolemize a formula. We use the rewrite rule $(\exists x.A) \rightarrow A[x := s(x_1, \ldots, x_n)]$, where s is a fresh skolem function of arity n and $\{x_1, \ldots x_n\} = FV(A) \setminus \{x\}$.
3. Then we turn the formula into head-normal form by putting all $\forall$ quantifiers in front of the formula.
4. Finally, we compute the conjunctive normal form of the unquantified parts.

We now have a formula of the form $\forall x_1, \ldots x_n.C_1 \wedge \ldots \wedge C_m$, where every C_i is a disjunction $L_1 \vee \ldots \vee L_k$ of literals. The C_i are called clauses.

We also define a Query Database Normal Form (QDbNF), which is only slightly different from DbNF. The QDbNF is needed, since the skolem functions in the database will be different from the ones in the query, yet they represent existentially quantified variables that might match. Therefore, in the QDbNF we keep the existential quantifiers, but skolemize universal quantifications.

The QDbNF of a formula P is computed as follows: Perform steps 1 till 3 on the formula $\neg P$, yielding P', which is a skolemized head-normal form in which negations only occur in front of atomic formulas. We then apply step 1 again on $\neg P'$ and finally apply step 4. The QDbNF has the form $\exists x_1, \ldots x_n.C_1 \wedge \ldots \wedge C_m$, where every C_i is again a clause.

It is semi-decidable if a stronger theorem is already in the database. Therefore, we define a notion of "more general", which is a more restrictive, but computable claim about theorems. Instead of checking if a stronger theorem is already in the database, we will check if a more general theorem is in the database.

Definition 6 (More general clause).
Let $L = L_1 \vee \ldots \vee L_n$ and $L' = L'_1 \vee \ldots \vee L'_m$ be clauses and let θ be a mapping from $FV(L) \cup FV(L')$ to ground terms. L is said to be more general than L' according to θ if there exists a mapping $\sigma : \{1, \ldots, n\} \rightarrow \{1, \ldots, m\}$ such that $\theta(L_i) = \theta(L'_{\sigma(i)})$ for $i = 1, \ldots, n$. Note that $\langle\rangle \vdash \theta(L) \Rightarrow \theta(L')$.

Definition 7 (More general CNF formula).
Let $P = P_1 \wedge \ldots \wedge P_n$ and $Q = Q_1 \wedge \ldots \wedge Q_m$ be formulas in conjunctive normal form (CNF) and let θ be a mapping from $FV(P, Q)$ to ground terms. P is said to be more general than Q if there exists a mapping $\sigma : \{1, \ldots, m\} \rightarrow \{1, \ldots, n\}$ such that $P_{\sigma(i)}$ is more general than Q_i for $i = 1, \ldots m$. Note that $\langle\rangle \vdash \theta(P) \Rightarrow \theta(Q)$.

Definition 8 (More general (Q)DbNF formula).
Let P and Q be formulas, such that the DbNF of P is $\forall x_1, \ldots x_n.P'$ and the QDbNF of Q is $\exists y_1, \ldots y_m.Q'$. P' and Q' are in CNF. P is said to be more general than Q if there exists a mapping θ from $x_1, \ldots, x_n, y_1, \ldots y_m$ to ground terms such that P' is more general than Q'.

Definition 9 (More general theorem).
A theorem $\Gamma \vdash P$ *is defined to be "more general" than theorem* $\Delta \vdash Q$ *if* P *is more general than* Q *and for every formula in* Γ *there is a more general formula in* Δ.

Theorem 1 (More general implies stronger).
If $\Gamma \vdash P$ *is more general than* $\Delta \vdash Q$*, then* Δ *is stronger than* Γ *and* P *is stronger than* Q*, hence* $\Gamma \vdash P$ *is stronger than* $\Delta \vdash Q$.

We consider a theorem to be in the database if a more general theorem is in the database, which is a valid conclusion because of theorem 1.

In order to store a proven theorem $\Gamma \vdash P$ in the database, we first compute the QDbNF of all formulas in Γ and the DbNF of P. This yields $\Gamma' \vdash P'$, which is actually stored.

When a query $\Delta \vdash Q$ is looked up in the database, we compute the DbNF of all formulas in Δ and the QDbNF of Q. The unquantified parts of the required normal forms of all formulas to find a more general theorem in the database are now directly available. The required substitutions are found by using a simple Robinson unification algorithm [13], which also solves the problem of the order of universal quantifications. By using an incremental, substitution free unification algorithm [14], the efficiency is comparable to syntactic comparison. To find the required mappings between conjuncts and disjuncts, we use trial and error (trying for each disjunct and conjunct every opposing disjunct and conjunct) yielding a quadratic search. This is made almost linear by imposing an ordering (like lexicographical path ordering, LPO) on the terms within the clauses.

5.1.5. Controller

The task of the controller is twofold: It provides an API to the user application to pose queries and it manages the tools it has available to answer them.

The queries posed by the user application must have the form $\Gamma \vdash P$, where Γ is a list of declarations, definitions and assumptions called the context and where P is a formula.

To answer the query, the controller first consults the database (if available). If the theorem is not in the database, external automated theorem provers are launched. The launching order and the timeout for each theorem prover can be configured within the controller. It is also possible to launch all theorem prover simultaneously and see which one find an answer to the query first.

The answer to a query can be *True*, meaning that P holds in the given context Γ, *False*, meaning that P does not hold in Γ, *unknown*, meaning that the repository processed the query completely, but cannot find an answer, and *error*, meaning that some error occurred while processing the query. The answer also contains other information, like the name and version of theorem prover that provided the result, the time taken to construct the proof and, if desired, the literal proof-output of the theorem prover.

Access to the repository is provided in two ways: (1) By a direct API of method calls of a Java object and (2) By commands issued through a socket stream to the repository (i.e. the repository acts as an internet service). While the first way is more direct and usually easier to implement for the user application, the second way opens a whole new perspective to use the repository. A special network connector object is provided that allows one repository to consult another

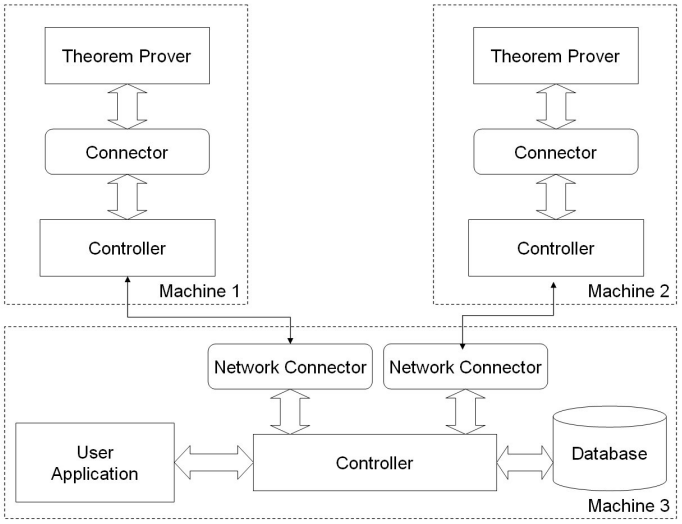

Figure 3: Using a separate machine for each theorem prover

repository as if it were a theorem prover. A controller then only configures the provers available on the local machine. By connecting the local repository to other repositories on other sites, the entire setup of the system becomes configurable. We will elaborate on this in Section 5.2.

5.2. Configurations

The main user applications we had in mind when designing the repository were tools for proving correctness of programs. Usually these tools are linked to a specific prover. For instance, ESC/Java [1] and Boogie are linked to Simplify [5] and ZAP [15], the Omnibus system [16] is linked to PVS [17], Perfect Developer [3] and Cocktail [11] use their own internal provers, etc.

By connecting these systems to the repository instead, they are able to benefit from the strengths of several theorem provers. Also, proofs only have to be constructed once, since the database prevents the same proof from being constructed repeatedly. Reconstruction of proofs proved to be a drawback of systems that rely solely on automated theorem provers, like ESC/Java and Perfect Developer (see [18]).

Also, the modules can be used to boost computational power: One can configure one machine for each theorem prover, using only one connector and the controller (without database). The machine running the user application then consults the other machines through the special network connector instead of running its own theorem provers (see Figure 3). One can also setup one additional node that only runs the database to implement a central repository that is used by several user applications. An extreme case of this situation is to run the entire repository as Software as a Service.

Using the modules with the database only locally and with just one prover is already useful to

support reusing proofs. This will already speed up a user application by avoiding the needless reconstruction of proofs.

6. Handling inductive types and recursive functions

Inductive datatype definitions and recursive function definitions are supported by the repository and its built-in proof assistant. As useful as this kind of definitions are to the user, they are not part of standard first-order logic and not generally supported by first-order theorem provers. The reason for this is that proving properties of inductive type definitions usually requires induction proofs, but induction proofs only prove the property for elements of the inductive type with a finite size. Theorem provers do not assume that all models only map to values of finite size and hence, they do not use induction. We only consider programs about elements of finite size, so we can safely allow induction proofs in our system.

Since the theorem provers do not support inductive types, their definitions must be translated into a set of first-order formulas to send to the theorem provers.

We use a translation function *to1st* to translate inductive datatypes and recursive functions into first-order logic.

Definition 10. *Translation function to1st*
Inductive type definitions and recursive specification function definitions are translated into sets of first-order declarations and axioms (assumptions) by the translation function to1st:

$$
\begin{aligned}
&\mathit{to1st}(\textbf{type } t \textbf{ is } C_1(\overline{p_1}) \square \ldots \square C_n(\overline{p_n}) \textbf{ end}) = \\
&\quad t : *_s; \\
&\quad C_1(\overline{p_1}) : t; \\
&\quad \ldots \\
&\quad C_2(\overline{p_n}) : t; \\
&\quad \mathit{Cons\_t} : (\forall v : t.(\exists \overline{a_1}.t = C_1(a_1)) \vee \ldots \vee (\exists \overline{a_n}.t = C_n(a_n)));
\end{aligned}
$$

$*_s$ is used to denote the set of all sets (i.e. $t : *_s$ declares that t is a type). The Cons_t axiom enables the external provers to make case distinctions when proving properties over elements of t (every element of t can be written as the result of one of its constructors). Note, that Cons_t can be derived from the inductive definitions by an induction proof.

$$
\begin{aligned}
&\mathit{to1st}(\textbf{define } f(\overline{p}) : t \textbf{ as } M \textbf{ end}) = \\
&\quad f(\overline{p}) : t; \\
&\quad T(f(\overline{p}), id, M)
\end{aligned}
$$

where id is the identity mapping (the empty substitution). T translates the match-expressions for a function to a set of assumptions. $T(F, \theta, E)$ with F a function, θ a substitution and E a (match-)expression is defined by:

$T(F, \theta, \textbf{match } v \textbf{ with } C_1(\overline{a_1}) \rightarrow E_1 \,\square \ldots \square\, C_n(\overline{a_n}) \rightarrow E_n \textbf{ end}) =$
$T(F, [v \mapsto \theta(C_1(\overline{a_1}))] \circ \theta, E_1);$
$\ldots$
$T(F, [v \mapsto \theta(C_n(\overline{a_n}))] \circ \theta, E_n)$

$T(F, \theta, E) = C(\theta(F) = \theta(E))$ *if* E *is not a match-expression*

where $C(P)$ *denotes the universal closure* $(\forall FV(P).P)$*. These declarations and assumptions are such that any first-order theorem prover is able to handle them.*

Example 2. *trees For example, consider the following inductive definition of trees and the functions to compute the number of leafs and nodes in a tree:*

type *tree* **is** $leaf(x : int) \,\square\, node(L, R : tree)$ **end**
{**define** $leafcount(t : tree) : int$ **as**
match t **with**
$leaf(x) \quad \rightarrow 1$
$\square \quad node(L, R) \rightarrow leafcount(L) + leafcount(R)$
end
end}
{**define** $nodecount(t : tree) : int$ **as**
match t **with**
$leaf(x) \quad \rightarrow 0$
$\square \quad node(L, R) \rightarrow nodecount(L) + nodecount(R) + 1$
end
end}

The translation into first-order logic then becomes:

$tree : *_s;$
$leaf(x : int) : tree;$
$node(L, R : tree) : tree;$
$Cons\_tree : (\forall t : tree.(\exists x : int.t = leaf(x)) \vee (\exists L, R : tree.t = node(L, R)));$
$leafcount(t : tree) : int;$
$(\forall x : int.leafcount(leaf(x)) = 1);$
$(\forall L, R : tree.leafcount(node(L, R)) = leafcount(L) + leafcount(R);$
$nodecount(t : tree) : int;$
$(\forall x : int.nodecount(leaf(x)) = 0);$
$(\forall L, R : tree.nodecount(node(L, R)) = nodeount(L) + nodecount(R) + 1;$

which are indeed first-order axiomatic definitions that can be handled by any theorem prover.

If we now request a proof for $(\forall t : tree.leafcount(t) = nodecount(t) + 1)$ *from the repository, no theorem prover is able to proof it, by lack of induction (note that the property does not hold for infinite trees). We then indicate in the built-in proof assistant that we want to prove the main goal by induction (just selecting the correct tactic). This yields two new subgoals (see the screenshot*

in Figure 4), namely the base and the induction cases:

$$(\forall x : int.leafcount(leaf(x)) = nodecount(leaf(x)) + 1)$$
$$(\forall L, R : tree.(leafcount(L) = nodecount(L) + 1) \Rightarrow$$
$$(leafcount(R) = nodecount(R) + 1) \Rightarrow$$
$$(leafcount(node(L, R)) = nodecount(node(L, R)) + 1))$$

Both of these can be proved by automated theorem provers connected to the repository.

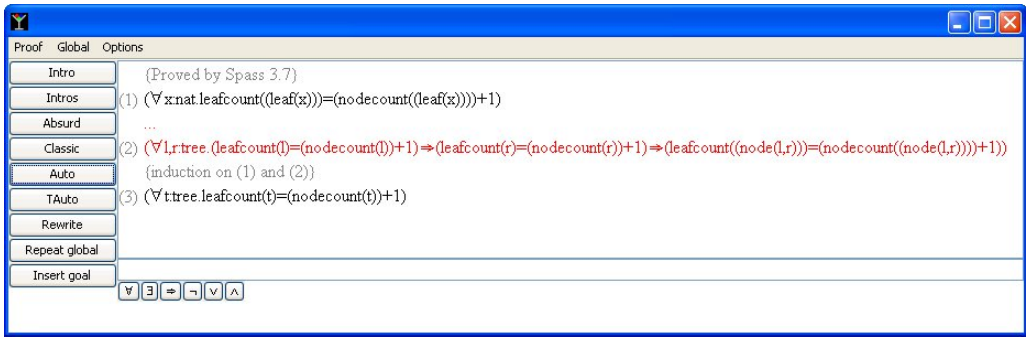

Figure 4: A screenshot of the proof assistant doing induction on trees.

7. Conclusions and Future Work

The language as described fulfils its purpose as a test suite for the proof repository. It can be used to express smaller algorithmic examples that lead to interesting verification conditions that cannot always be proved fully automatically.

The interactive theorem prover supports the inductive types and recursive definitions directly, allowing the user to split some of the harder theorems into simpler sub theorems that are then handled by the repository.

Since all proofs are stored in the database, verification conditions that have not changed after a small change in the program are not proved again. This constitutes a dramatic speed-up when re-verifying a program after small changes have been made. The time allotted to verify the program can now be entirely spent on those verification conditions that are actually new or changed. Also, since the database has some limited deductive capabilities, renaming variables or adding new definitions does not lead to re-constructing proofs of verification conditions.

Next, we want to extend the language to include more advanced features (recursive procedures, references, arrays, etc.) in order to write more interesting programs. Also, the language would then become more interesting to serve as an intermediate language to verify real-world programs.

We also will test the repository in several different situations (e.g. educational settings) and configurations (local repositories, a central database, etc) to find out what more is needed for large scale application.

References

[1] C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, R. Stata, Extended static checking for Java, SIGPLAN Not. 37 (5) (2002) 234–245. doi:http://doi.acm.org/10.1145/543552.512558.

[2] M. Barnett, K. R. M. Leino, W. Schulte, Vol. 3362 of LNCS, Springer, 2005, Ch. The Spec♯ Programming System: An Overview, pp. 49–69.

[3] G. Carter, R. Monahan, J. Morris, Software refinement with Perfect Developer, Software Engineering and Formal Methods, IEEE, 2005, pp. 363–372.

[4] P. Morris, T. Altenkirch, N. Ghani, Constructing strictly positive families, in: CATS '07: Proceedings of the thirteenth Australasian symposium on Theory of computing, Australian Computer Society, Inc., Darlinghurst, Australia, Australia, 2007, pp. 111–121.

[5] D. Detlefs, G. Nelson, J. B. Saxe, Simplify: A theorem prover for program checking, Journal of the ACM 52 (3) (2005) 365–473.

[6] C. Weidenbach, D. Dimova, A. Fietzke, R. Kumar, M. Suda, P. Wischnewski, SPASS Version 3.5, Automated Deduction–CADE-22 (2009) 140–145.

[7] S. Schulz, E – A Brainiac Theorem Prover, Journal of AI Communications 15 (2/3) (2002) 111–126.

[8] C. Weidenbach, B. Afshordel, E. Keen, C. Theobalt, D. Topić, Spass theorem prover, in: URL: `http://spass.mpi-sb.mpg.de/`, Max-Planck-Institut für Informatik, 2007.

[9] L. De Moura, N. Bjørner, Z3: An efficient SMT solver, Tools and Algorithms for the Construction and Analysis of Systems (2008) 337–340.

[10] G. Sutcliffe, C. Suttner, The TPTP problem library, Journal of Automated Reasoning 21 (2) (1998) 177–203.

[11] M. Franssen, Cocktail: A tool for deriving correct programs, Ph.D. thesis, Eindhoven University of Technology (2000).

[12] M. de Mol, M. van Eekelen, A Proof Tool Dedicated to Clean, Applications of Graph Transformations with Industrial Relevance (2000) 254–257.

[13] J. Robinson, A machine oriented logic based on the resolution principle, Journal of the Association for Computing Machinery 12 (1) (1965) 23–41.

[14] M. Franssen, Implementing rigid E-unification, Computing Science Report 08–24, Eindhoven University of Technology (2008).

[15] T. Ball, S. Lahiri, M. Musuvathid, Zap: Automated theorem proving for software analysis, Tech. Rep. MSR-TR-2005-137, Microsoft Research (October 2005).

[16] T. Wilson, S. Maharaj, R. G. Clark, Flexible and configurable verification policies with omnibus, Journal on Software and Systems Modeling 7 (3) (2008) 257–272.

[17] S. Owre, J. Rushby, N. Shankar, PVS: A prototype verification system, in: D. Kapur (Ed.), 11th International Conference on Automated Deduction, Vol. 607 of Lecture Notes in Artificial Intelligence, CADE, Springer Verlag, 1992, pp. 748–752.

[18] H. Maassen, Verified design by contract : case studies, Master's thesis, Eindhoven University of Technology (2008).