An Architecture for Interactive Program Provers

Jörg Meyer and Arnd Poetzsch-Heffter

Fernuniversität Hagen, D-58084 Hagen, Germany
{Joerg.Meyer,Arnd.Poetzsch-Heffter}@fernuni-hagen.de

Abstract. Formal specification and verification techniques can improve the quality of programs by enabling the analysis and proof of semantic program properties. This paper describes the modular architecture of an interactive program prover that we are currently developing for a Java subset. In particular, it discusses the integration of a programming language-specific prover component with a general purpose theorem prover.

1 Introduction

Specification and verification techniques can improve the quality of programs by enabling the analysis and proof of semantic program properties. They can be used to show the absence of exceptions and to prove that a program satisfies certain interface properties or a complete interface specification. This is particularly interesting for the emerging market of software components. As we illustrate in Section 2, tool support is crucial for the application of such formal techniques.

The paper motivates and describes the modular architecture of an interactive program prover that we are currently developing for a Java subset. In particular, it discusses the integration of a programming language-specific prover component with a general purpose theorem prover. The goal of this research is to provide a powerful and flexible tool that

- supports complete a posteriori program verification;
- provides assistance in top-down program development, e.g. for deriving specifications of auxiliary procedures;
- allows one to specify and check certain simple, but in general undecidable properties, such as the absence of null pointer dereferencing and out-of-bounds access to arrays.

As illustrated by the last aspect, we are not only interested in algorithm verification, but as well in showing the absence of certain (language dependent) program errors. In particular, we have to deal with sharing and abstraction. We build on an object-oriented language, because it supports encapsulation on the level of types and because subtyping simplifies reuse.

Overview. The paper is organized as follows. Section 2 provides the technical background for specification and verification, motivates our approach, and discusses related work. Section 3 presents the overall architecture for the interactive verification environment. Section 4 focuses on the realization of the program prover component and describes its application by an example.

2 Verification of Realistic Programs

Verification of realistic programs is a fairly complex task. The goal of this section is to illustrate where this complexity comes from and to give an overview of tool-based approaches to cope with this complexity. The first subsection sketches state-of-the-art specification techniques and the involved formal background that has to be mastered by verification tools. The second subsection summarizes mechanical approaches to formal program verification from the literature.

2.1 Specifying Object-Oriented Programs

Program specifications should describe the behavior of program components in a *formal* and *abstract* way: Formality is a prerequisite for computer-aided verification. Abstraction is needed to achieve implementation independency and to simplify verification. In the following we summarize formal techniques to achieve abstraction in OO-languages.

We build on the Larch approach to program specification (cf. [GH93]) that uses type invariants and pre- and postconditions for procedures/methods. The following Java program fragment shows an interface type `Set`[1] and an implementation of this type based on arrays.

```
interface Set {
  boolean add( Object o );
  boolean contains( Object o );
  int size();
   ... }
```

```
class ArraySet implements Set {
  private Object[] elems;
  private int setsize;
  boolean add( Object o ){ ... }
  ... }
```

Since the interface `Set` may have several implementations and since it should be possible to modify implementations without changing the specification, the specification of `Set` cannot refer to any implementation parts, i.e., it has to be given in abstract terms. We specify the behavior of type *Set* using an abstract data type with main sort SET and the usual operations, and an abstraction function *aSet*. *aSet* maps a Set object X and an object store to a value of sort SET. The object store is needed to capture the objects referenced by X. Method `add` can thus be specified as follows where \$ is a variable denoting the current object store and the caret-operator yields the value of a variable in the prestate:

```
boolean add( Object o )
  pre  $\mathrm{o} \neq \mathrm{null}$
  post $\mathrm{result} = (\mathrm{o} \in \mathit{aSet}(\mathrm{this}, \$\hat{}\,)) \;\wedge\; \mathit{aSet}(\mathrm{this}, \$) = \{\mathrm{o}\} \cup \mathit{aSet}(\mathrm{this}, \$\hat{}\,)$
     $\wedge\; \forall\,\mathrm{Object}\, X :\; \neg \mathit{inRepSet}(X, \mathrm{this}, \$\hat{}\,) \;\Rightarrow\; \mathit{unchanged}(X, \$, \$\hat{}\,)$
```

The first conjunct of the postcondition states that `add` yields true if and only if the object to be inserted is already contained in the set. The second conjunct specifies that after execution the implicit parameter `this` refers to the enlarged set. The third conjunct describes that the states of all objects not belonging to the representation of the input set remain unchanged. The representation of an

[1] A simplified version of the Set type as contained in the Java library.

abstract value comprises all objects that are used to represent the value in the object store. Since abstraction functions and the predicate *inRepSet* depend on implementations, they have to be explicitly defined. E.g., the provider of class `ArraySet` has to define an abstraction function and the predicate *inRepSet* for objects of type `ArraySet`. (For a set represented by an ArraySet-object Y, the representation consists of Y and the referenced array object.)

The above example illustrates the basic aspects needed in a realistic framework for formal program specification and verification. A more detailed presentation of the specification techniques can be found in [PH97b,MPH99]. In summary, such a framework has to provide the following features:

– To express abstraction, it is necessary to define and reason about abstract data types that are specified outside the programming language (e.g. SET).
– To specify modifications of the object store and to formulate properties on the program level (e.g. absence of null pointer dereferencing), a formalization of objects and the object store has to be provided by the framework.
– The abstract and program levels have to be integrated to be able to specify abstraction functions, representation predicates, and abstract data types that are based on types of the programming language (e.g. the abstract data type SET refers to elements of type `Object`).

The **J**ava **i**nteractive **v**erification **e**nvironment Jive that is described in this paper supports all of the above features (cf. Section 3 and 4).

2.2 Computer-Aided Program Verification

In the literature, we can essentially find three approaches to computer-aided program verification.

Verification Based on Language Semantics. This technique works as follows: Translate the program into a general specification framework (e.g. HOL) in which the semantics of the programming language is defined. Then state the program properties directly within this framework and use the rules of the language semantics for verification. This techniques is e.g. applied in [JvdBH$^+$98].

The advantage of this approach is that existing frameworks equipped with powerful tools can be used without extensions (e.g., PVS [COR$^+$95] or Isabelle [Pau94]). Only the translation process has to be implemented (and verified). The main disadvantage is that specification and verification on the semantics level is very tedious, because the abstraction step gained by an axiomatic language definition once has to be done in semantics level proofs again and again.

Verification Condition Generation. VCG was the classical approach to program verification. Based on a weakest precondition semantics, a program specification is transformed into a (weakest) precondition formula guaranteeing that a program satisfies its postcondition; i.e., program dependent aspects are eliminated by the wp-transformation. The precondition formula (verification condition) can be proved by a general theorem prover. This is the technique used in systems like e.g. the Standford Pascal Verifier [Com79].

The advantage of this approach is again that program dependent aspects are eliminated automatically and that the proper verification task can be done using standard tools. The disadvantage is that in realistic settings (cf. Section 2.1) the verification conditions become huge and very complex. There are essentially three reasons for that: (recursive) procedure/method calls, aliasing operations on the object store, and abstraction. As an example, consider an invocation site of method `add` of interface `Set`. If we use the specification of `add` to compute a weak precondition for a formula **Q**, the resulting precondition has about the size of the method specification plus the size of **Q**. The reason for this is that the postcondition of `add` does usually not match **Q** and that simplification is not trivial. Having several method invocations in a statement sequence easily leads to unmanagable preconditions.

Working with verification condition generation has two further disadvantages: a) If the generated condition cannot be proved, it is often difficult to find out why this is the case and which program statement causes to fail (cf. [GMP90] for a discussion of this issue). b) VCG is fairly inflexible and only applicable in an a-posteriori verification. E.g., in top-down program development, one would like derive the needed properties of a used method from the specification of the calling method.

Interactive Program Verification. Interactive program verification applies ideas from general tactical theorem proving to programming logics like e.g. to Hoare logic (see below) or dynamic logic (cf. [Rei95]). The main advantage of this approach is that program proofs can be developed around the program, i.e. as annotations to the program. The intuition about the program can be directly used for the proof. Strengthening and weakening steps can be applied where most appropriate. In particular, such steps can be done before and after method invocations to adapt method specifications to the needs at the invocation site. This way the described problem of VCG can be avoided. In addition to this, it is usually easy to detect program errors from failing proofs. Furthermore an advantage is that the interactive program prover "knows" the programming language and can provide appropriate views. On the other hand, a language dependent program prover has to be developed which is quite an engineering challenge.

Because of the described disadvantages of the first and second approach, we decided to construct an interactive, tactical prover for program verification. Within this prover, weakest precondition techniques can be implemented by tactics and used where appropriate without giving up flexibility. Depending on the goals, other combinations of the verification approaches sketched above are investigated in the literature. E.g. in [Gor89], Gordon demonstrates the development of programming logics based on a general HO-theorem prover showing the strength of HO-reasoning. However, he cannot exploit the specific relation between programs and program proofs. In the Extended Static Checker for Java (ESC/Java, cf. [DLNS98]), the translational approach is combinded with VCG for automatic checking of a restricted class of specifications.

3 An Architecture for Interactive Program Provers

This section presents our architecture for interactive program provers. The architecture is based on the following requirements: The functional properties as described in Section 2.1 have to be fulfilled. The newly implemented program prover component should only be responsible for proof tasks that are directly related to the program. General specification and verification aspects should be delegated and performed by state-of-the-art theorem provers. This comprises in particular the definition of the object store and abstraction functions. In the following, we first analyse the consequences of combining general provers with language-specific provers. Then, we explain the overall architecture.

Communicating Provers. There are two architectural alternatives for combining a general prover and a programming language specific prover component within a verification system: a) The general theorem prover is encapsualed by the system and hidden to users. b) Both prover components are accessible by users and the communication between them is visible. For the following reasons, we decided for the second alternative. It provides more flexiblity and is easier to react to new developments w.r.t. the general theorem prover. The implementation is less expensive. For third party users, the programming interfaces of existing theorem provers are not sufficiently powerful to control all prover operations by an external process. The disadvantage of this solution is that users of the resulting system have to handle two interactive components: the program prover component and the theorem prover component.

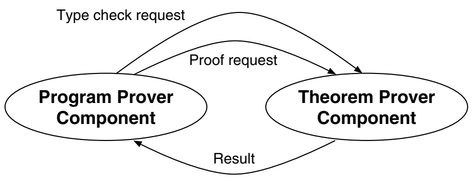


Fig. 1. Basic prover components.

The communication between the two prover components is illustrated in Figure 1. The program prover sends type checking and proof requests to the general theorem prover. Type checking of pre- and postformulas is done in the general theorem prover, as these formulas contain logical variables and make use of abstract data types specified as theories in the syntax of the general theorem prover. Proof requests result from strengthening and weakening steps in program proofs. They are formulas not refering to a special program part and are verified online or offline in the theorem prover component. The information whether such proof obligations have been proven or not is sent back to the program proof component. This way, the program proof component can check whether program proofs are complete.

The communication between program prover and theorem prover is based on a communication interface that allows one to send formulas from the program prover to the theorem prover. Type check requests differ from proof requests in that they are usually[2] solved automatically whereas proof requests typically need user interaction.

[2] The subtyping of the used specification language (PVS) is in general undecidable.

The Jive Architecture. This subsection describes the overall architecture of Jive. Jive supports the sequential kernel of the Java language including recursive methods, classes, abstract classes, interfaces, thus inheritance and subtyping, static and dynamic binding, aliasing via object references, and encapsulation constructs. Exception handling and some of the predefined data types (in particular float and char) are not yet supported (cf. [MPH99] for a precise description). In addition to the Java subset, Jive supports annotations like that shown in Section 2.1. The common language for programs and annotations is called Anja (annotated Java). In the following, we explain the architectural components and the input sources of proof sessions based on the overview given in Figure 2.

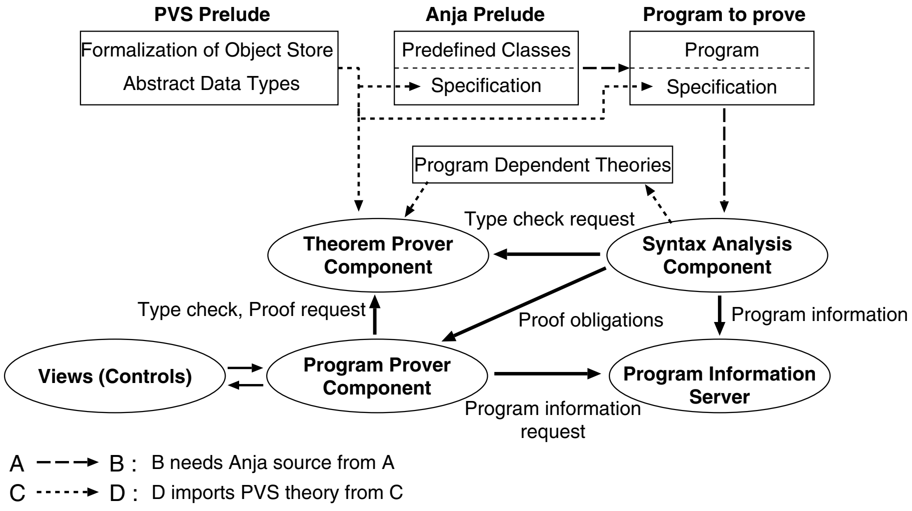

PVS Prelude
Formalization of Object Store
Abstract Data Types
Anja Prelude
Predefined Classes
Specification
Program to prove
Program
Specification
Program Dependent Theories
Type check request
Theorem Prover Component
Syntax Analysis Component
Type check, Proof request
Proof obligations
Program information
Views (Controls)
Program Prover Component
Program Information Server
Program information request
A ---→ B : B needs Anja source from A
C ·····→ D : D imports PVS theory from C

Fig. 2. The Jive architecture

System Components. The architecture is based on five components: 1.) The syntax analysis component that reads in and analyzes annotated programs and generates the program proof obligations. 2.) The program information server that makes the static program information gathered in the analysis phase available to other parts of the system. 3.) The program prover component managing the program proofs. 4.) Views to visualize program proofs and to control proof construction. 5.) The theorem prover to solve program independent proof obligations. In our current implementation, we use PVS for general theorem proving.

The program proof component encapsulates the construction of program proofs. It provides two things: (1.) A container which stores all information about program proofs and (2.) an interface which provides operations to create and modify proofs within this container. Since the content of the proof container represents the program proof state, it is strongly encapsulated to the rest of the system. Modifications of the proof state can only be achieved by operations of the container interface (see Section 4). Therefore correctness of proofs is ensured by the correctness of the basic container operations.

During program proof construction, various information about the underlying program is needed by the program proof component: The structure of the abstract syntax tree, results of binding and type analysis, and the program unparsing for visualization. This kind of information is provided by the program information server. In contrast to a compiler frontend, all information computed during static program analysis has to be available online after the analysis.

Proof Setup. The verification of a program is based on three formal texts: 1.) The PVS prelude containing two parts: (a) the formalization of the object-store; (b) the specification of abstract data types used in program annotations. Whereas the former part is program independent, the latter may be program dependent. 2.) The Anja prelude containing the specifications of predefined and library classes and interfaces. 3.) An Anja program, i.e. a program in our Java subset together with a suitable interface specification. Annotations are formulated in a language based on the specification language of the underlying theorem prover, i.e. PVS in our case. As illustrated in Section 2, they may refer to program variables and use abtract data types specified in the PVS prelude.

From the described sources, the syntax analysis component generates three things: 1. The program proof obligations which need to be proven to guarantee that the program fulfills its specification. They are entered into the proof container. 2. Program dependent theories formalizing some of the declaration information of the program for the theorem prover. 3. The abstract syntax tree decorated with information of the static analysis. It is managed by the program information server.

After syntax and static analysis, the system is set up for interactive proof construction. The user constructs program proofs using basic proof operations and tactics (see Section 4). The views and controllers provide access to the proof state. Program independent proof obligation are verified with the general theorem prover. The program prover monitors the overall proof process and signals the completion of proof tasks.

4 The Program Prover

Program proofs in Jive are based on a Hoare logic for object-oriented programs (cf. [PHM99]). Hoare logic is sufficient for our purposes and enables us to visualize proof parts as program annotations. This section first describes how the basic proof operations of the proof container are derived from the logic and how tactics can be formulated. Then, it explains the user interface for proof construction and visualization and sketches a simple proof done within Jive.

4.1 Mechanizing the Programming Logic

As the supported programming language provides recursive methods, the Hoare logic deals with *sequents* of the form $\mathcal{A} \vdash \{ \mathbf{P} \} \text{ comp } \{ \mathbf{Q} \}$ where $\mathcal{A}$ denotes a set of method specifications (the *assumptions*), **P** and **Q** are first-order formulas, and comp denotes a statement or a method, the so-called *program component* of the sequent. Program components are represented by references to the abstract program syntax tree. A rule in the logic consists of a finite number of antecedents and a sequent as conclusion. The antecedents are either sequents or first-order formulas. Rules without antecedents are called axioms. As examples, we show the rule to verify if-statements and the assumpt-intro-rule to introduce assumptions:

$$\frac{\mathcal{A} \vdash \{\, e \wedge \mathbf{P} \,\}\ \text{stmt1}\ \{\, \mathbf{Q} \,\}\ ,\ \ \mathcal{A} \vdash \{\, \neg e \wedge \mathbf{P} \,\}\ \text{stm2}\ \{\, \mathbf{Q} \,\}}{\mathcal{A} \vdash \{\, \mathbf{P} \,\}\ \text{if (e)}\ \{\ \text{stm1}\ \}\ \text{else}\ \{\ \text{stm2}\ \}\ \ \{\, \mathbf{Q} \,\}} \qquad \frac{\mathcal{A} \vdash \mathbf{A}}{\mathbf{A}_0\ ,\ \mathcal{A} \vdash \mathbf{A}}$$

Basic Proof Operations. As usual, proof trees are constructed from rule instances. A tree node has as many children as the rule has antecedents. There are two ways to construct proof trees. 1. A *forward proof step* takes several proof trees and combines them with a new root node. 2. A *backward proof step* adds a new node to one of the leaves. A proof tree is *closed* if all leaves are instances of axioms or first-order formulas that are proved by the theorem prover component.

To gain the flexibility explained in Section 2, JIVE supports operations for forward and backward proof steps. These operations have to be distinguished, because different directions require different context checks for formulas, program components, and parameters. The *if-rule* serves as an example: Forward proving combines two proof trees $\mathcal{S}_1$ and $\mathcal{S}_2$ to a new proof tree, backward proving refines a proof goal $\mathcal{G}$ of an if-statement into two subgoals for the then- and else-branch. The context conditions of the *if_forward* and *if_backward* operations are as follows:

Forward Proof:

1. $\mathcal{S}_1$ and $\mathcal{S}_2$ have to be roots of proof trees.
2. The assumptions of $\mathcal{S}_1$ and $\mathcal{S}_2$ have to be equal.
3. e has to be a conjunct of $\mathcal{S}_1$.
4. $\neg e$ has to be a conjunct of $\mathcal{S}_2$.
5. The preconditions of $\mathcal{S}_1$ and $\mathcal{S}_2$ have to be equal modulo the conjuncts e and $\neg e$ resp.
6. The postconditions of $\mathcal{S}_1$ and $\mathcal{S}_2$ have to be equal.
7. `stmt1` and `stmt2` have to be the then- and else-branch of the same if-statement.

Backward Proof:

1. $\mathcal{G}$ has to be the leaf of a proof tree.
2. The program component of $\mathcal{G}$ has to be an if-statement.

Proof operations are executed as follows: First, the context conditions are checked. If they are met, the proof operation is applied and leads to a new proof tree, and thus to a modified state in the proof container. Otherwise, an appropriate exception is raised that can be used in tactics (see below). Because operations first check all necessary context conditions, correctness of proofs is provided by the correctness of operations. The JIVE system is currently based on a Hoare logic with 26 rules and axioms. Thus, it provides 52 basic proof operations. In addition, JIVE provides a *cut* operation to remove, a *copy* operation to copy, a *paste* operation to combine proof tree parts, and operations to inspect parts of the proof tree or to navigate within proof trees. These operations allow for comfortable interactive work with program proof information, e.g., they enable one to cut off failing proof parts.

Tactics. Program proofs are constructed by successively using proof operations as described above. To simplify proof construction, sequences of proof operations, e.g. to apply a weakest precondition strategy, can be combined to form tactics. As an example, we show a tactic that eliminates the assumptions of an open leaf of the proof tree by iterating the *assumpt-intro-rule* unless all assumptions are eliminated[3]. Since the proof operations of JIVE are implemented in Java (see Section 4.4), tactics are formulated as Java programs invoking proof operations. The `getPre()`, `getPost()`, and `getComp()` operations return the precondition, the postcondition and the program component of the triple `t`:

```
public ProofTreeNode eliminate_assumptions(ProofContainer c,
                     ProofTreeNode ptn) throws ContextException {
  Enumeration e =  ptn.getAssumptions().elements();
  while(e.hasMoreElements()) {
    Triple t = (Triple)e.nextElement();
    ptn = c.assumpt_intro_backward(
                     ptn,t.getPre(), t.getCompRef(), t.getPost() );
  }
  return ptn;
}
```

4.2 User-Interfaces of the Program Prover

Interactive program proof construction with proof operations enforces users to work explicitly with several kinds of information like formulas, program structures, textual program representation, etc. A graphical user interface is required (1.) for an appropriate visualization of the proof state and of the related information as well as (2.) for convenient selection, input, and application of operations and tactics. Currently JIVE provides a so-called *tree view* and a *text view* to program proofs[4]. Of course, both views can be used together within one proof.

Tree View. The tree view (Figure 4 shows a screen shot) presents the information contained in the proof container in a graphical way. All parts of proof trees can be examined, selected, copied, combined, extended and deleted. Compared to the text view (see below), the tree view shows all details of proof trees. In particular, it enables the user to inspect proof steps that cannot be presented as proof outlines and shows the complete proof structure. It supports the structural operations on trees (*cut*, *copy*, *paste*) and proof operations that take several trees as arguments which are not related to one program component. Since proof trees are in general large structures, tree views enable to work with scalable clippings of trees, i.e., the user interface displays only relevant information.

[3] Applying the *assumpt-intro-rule* backward eliminates an assumption.
[4] At time of submission of this paper, the text view was still under construction.

Text View. To provide a more compact view to program proof information and to enable an embedding of program proof information into the program text, text views display selected proof information within a textual program representation (Figure 3 shows a screen shot). This technique is based on so called proof outlines (cf. [Owi75]). The text view allows the user to consider proofs (or at least most of the central proof steps) as annotations to the program text. This turns out to be very helpful for interactive program proofs as the program structure is well-known to the user and simpler to handle. In particular, it allows the direct selection of program components which is needed for forward proofs. In addition to this, well-designed proof outlines are a good means for program and proof documentation.

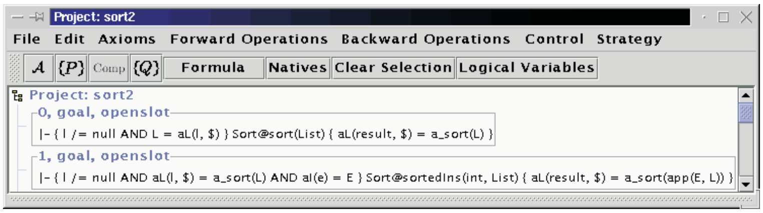
4.3 Using the Program Prover

In this section, we illustrate the use of the program prover by an example explaining in particular the interaction of automated and manual proof construction. Using an interface of a singly linked integer list, we want to prove a simple recursive sort method:

```
interface List {
 public List rest()
  pre  aL(this,$) = L;
  post aL(result,$) = rst(L)
          AND result /= null;
  pre  $=OS;
  post $=OS;

 public int first()
  pre  aL(this,$) = L;
  post aI(result) = fst(L);
  pre  $=OS;
  post $=OS;

 public boolean isempty()
   pre  aL(this,$)=L;
   post aB(result)=isempt(L);
   pre  $=OS;
   post $=OS;
}
```

```
class Sort {
 public static List sort(List l)
  pre  l/=null AND L=aL(l,$);
  post aL(result,$)=a_sort(L);
  {
   List lv,res;
   boolean bv; int ev;

   bv = l.isempty();
   if(bv) res = l;
   else {
    lv = l.rest();ev = l.first();
    lv = Sort.sort(lv);
    res = Sort.sortedIns(ev,lv);
   }
   return res;
  }
 static List sortedIns(int e, List l)
  pre  aL(l,$)=a_sort(L) AND aI(e)=E
          AND l/=null;
  post aL(result,$) = a_sort(app(E,L));
  { ... }
}
```

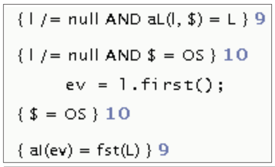
In the given Anja source, we use logical variables (written with capital letters) to bind values in the precondition for use in the postcondition. Each list method is specified by two pre-post-pairs. The first pair expresses the functional behavior using the abstraction function `aL` mapping objects to an abstract list data type. The second pair states that each method does not change the object store. Method `sort` of class `Sort` implements insertion sort using `sortIns` as auxiliary method; `a_sort` sorts an abstract list. We sketch the proof of method `sort` assuming the correctness of the list interface. `sort` is a static method, i.e. behaves like procedures in imperative languages. The logical aspects of dynamic binding cannot be discussed here (cf. [PHM99]). Starting the JIVE system with the given example yields the following two proof obligations for methods `sort` and `sortedIns`:


Project: sort2
File Edit Axioms Forward Operations Backward Operations Control Strategy
$\mathcal{A}$ {P} Comp {Q} Formula Natives Clear Selection Logical Variables
Project: sort2
0, goal, openslot
1, goal, openslot

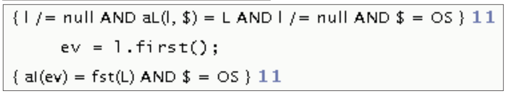
```
|- { l /= null AND L = aL(l, $) } Sort@sort(List) { aL(result, $) = a_sort(L) }
|- { l /= null AND aL(l, $) = a_sort(L) AND al(e) = E } Sort@sortedIns(int, List) { aL(result, $) = a_sort(app(E, L)) }
```

We start the verification of `sort` by applying the SWP-tactic to the goal 0. This tactic realizes a simple weakest-precondition-strategy. For the example, it reduces the method specification to a pre-post-pair for the statement sequence in the body. In our logic, this corresponds to two elementary proof steps that in particular conjoin "this $\neq$ null" to the precondition. Then, the SWP-tactic tries to verify the resulting pre-post-pair by forward proof steps starting with the rightmost innermost statement (according to the AST of the program), i.e., it starts at the end of the program text and works to the beginning. In our case, it automatically handles the return-statement and the then-branch of the if-statement. In the else-branch, it cannot procede because the postcondition of the if-statement does not match the postcondition of the specification for `sortedIns`. The corresponding proof state is illustrated by the screen shot of the text view in Figure 3. The system uses colors to distinguish program text from annotations and open proof slots (red) from verified triples (green). In the figure, this is indicated by brackets left to the screen shot. The red triple corresponds to the open leaf of the overall proof. The two green triples correspond to needed proof parts that have been constructed by the SWP-tactic.

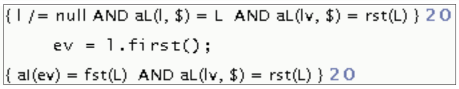
The proof state after termination of a tactic can be interactively manipulated by the user. He or she can add further proof parts using other tactics or basic proof operations and then rerun the original tactic on the original goal. Typically, user interaction is necessary at method invocation sites, because there may be more than one specification for a method and it is not obvious which of them has to be used or how they have to be combined. How to handle such situations is demonstrated with the invocation of method `first` in the example. We show three states of a forward proof: 1. After instantiating the method specifications at the invocation site (one basic proof step). 2. After conjoining the resulting triple (one basic proof step). 3. After adding of an invariant term (one basic proof step) and eliminating the logical variable OS (several basic proof steps, done by a tactic). For space reasons, we leave out the surrounding window context:

- State 1: 

```
{ l /= null AND aL(l, $) = L } 9
{ l /= null AND $ = OS } 10
      ev = l.first();
{ $ = OS } 10
{ al(ev) = fst(L) } 9
```

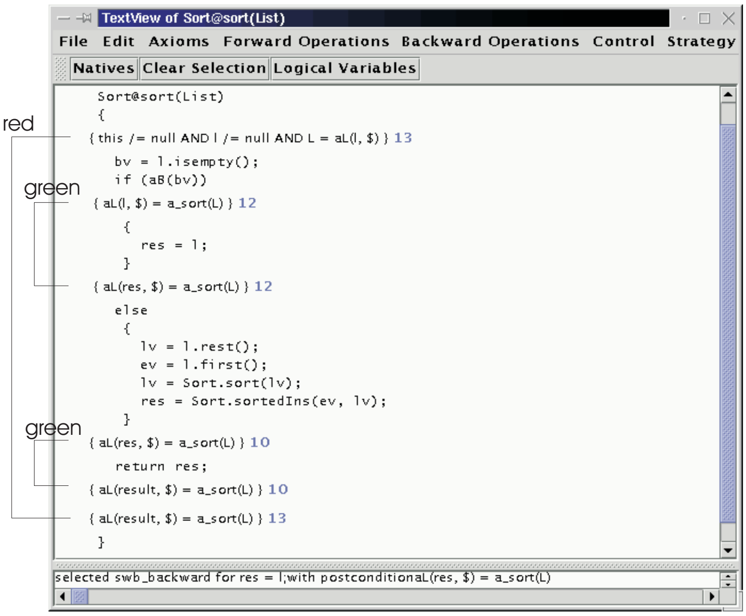
- State 2: 

```
{ l /= null AND aL(l, $) = L AND l /= null AND $ = OS } 11
      ev = l.first();
{ al(ev) = fst(L) AND $ = OS } 11
```

- State 3: 

```
{ l /= null AND aL(l, $) = L  AND aL(lv, $) = rst(L) } 20
      ev = l.first();
{ al(ev) = fst(L)  AND aL(lv, $) = rst(L) } 20
```

The other method invocations in the else-part of method `sort` are processed accordingly. The overall proof of method `sort` is constructed by approx. 90 basic proof operations where currently 20 of them are performed by tactics; 10 mostly trivial implications from weakening and strengthening remain to be verified by the general theorem prover. The tactics are still fairly primitive. With improved tactics and a refined specification technique, we aim to drastically reduce the amount of needed user interaction.

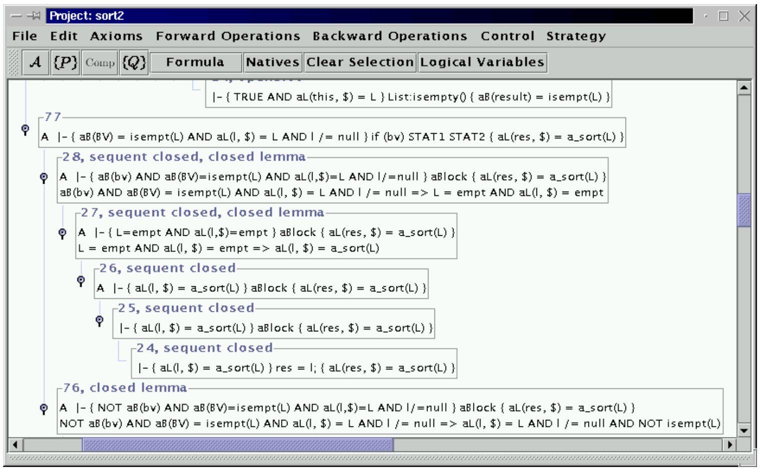

TextView of Sort@sort(List)
File Edit Axioms Forward Operations Backward Operations Control Strategy
Natives Clear Selection Logical Variables
red
green
green

```
Sort@sort(List)
{
{ this /= null AND l /= null AND L = aL(l, $) } 13
   bv = l.isempty();
   if (aB(bv))
{ aL(l, $) = a_sort(L) } 12
     {
        res = l;
     }
{ aL(res, $) = a_sort(L) } 12
   else
     {
        lv = l.rest();
        ev = l.first();
        lv = Sort.sort(lv);
        res = Sort.sortedIns(ev, lv);
     }
{ aL(res, $) = a_sort(L) } 10
   return res;
{ aL(result, $) = a_sort(L) } 10
{ aL(result, $) = a_sort(L) } 13
}
```

```
selected swb_backward for res = l;with postconditionaL(res, $) = a_sort(L)
```

Fig. 3. The text view after applying the SWP-tactic.

The presented example shows three important aspects of the system: 1. Forward and backward proving is combined to gain the flexibility needed to encode typical proof strategies. 2. User-guided and automated proof construction by tactics can be freely mixed so that automation can be done where possible and user interaction can focus on the critical points. 3. Different views are needed that allow one to inspect the proof parts on different levels of abstraction. Even in the simple example, there exist up to 7 proof fragments at one time with 17 open proof slots and proof tree depth up to 20. This amount of information is conveniently abstracted by the text view. If detailed information is needed, the user can refer to the tree view (see Figure 4).


Project: sort2
File Edit Axioms Forward Operations Backward Operations Control Strategy
𝒜 {P} Comp {Q} Formula Natives Clear Selection Logical Variables

```
|- { TRUE AND aL(this, $) = L } List:isempty() { aB(result) = isempt(L) }
77
A  |- { aB(BV) = isempt(L) AND aL(l, $) = L AND l /= null } if (bv) STAT1 STAT2 { aL(res, $) = a_sort(L) }
28, sequent closed, closed lemma
A  |- { aB(bv) AND aB(BV)=isempt(L) AND aL(l,$)=L AND l/=null } aBlock { aL(res, $) = a_sort(L) }
aB(bv) AND aB(BV) = isempt(L) AND aL(l, $) = L AND l /= null => L = empt AND aL(l, $) = empt
27, sequent closed, closed lemma
A  |- { L=empt AND aL(l,$)=empt } aBlock { aL(res, $) = a_sort(L) }
L = empt AND aL(l, $) = empt => aL(l, $) = a_sort(L)
26, sequent closed
A  |- { aL(l, $) = a_sort(L) } aBlock { aL(res, $) = a_sort(L) }
25, sequent closed
|- { aL(l, $) = a_sort(L) } aBlock { aL(res, $) = a_sort(L) }
24, sequent closed
|- { aL(l, $) = a_sort(L) } res = l; { aL(res, $) = a_sort(L) }
76, closed lemma
A  |- { NOT aB(bv) AND aB(BV)=isempt(L) AND aL(l,$)=L AND l/=null } aBlock { aL(res, $) = a_sort(L) }
NOT aB(bv) AND aB(BV) = isempt(L) AND aL(l, $) = L AND l /= null => aL(l, $) = L AND l /= null AND NOT isempt(L)
```

Fig. 4. A clipping of the proof tree view

4.4 Technical Issues

This section describes implementation issues concerning the Jive system.

System Implementation. As shown in the table below, the Jive system is implemented using a variety of tools and languages, mostly Java. Java combines several necessary properties, which are useful to implement heterogeneous tool environments. In particular, we make use of the Java Native Interface for C, of the API for TCP interfaces to connect to the theorem prover, and of the Swing library as graphical user interface. The central program proof component with the proof operations (as Java Methods) and auxiliary implementations parts such as formula handling is completely implemented in Java. Tactics are implemented as Java classes and can be dynamically loaded into the system. All other components are attached using the above mentioned interfaces.

Generative reuse techniques. One major design decision was to use as much as possible generative techniques to implement the system. This is possible because many subtasks to be solved are directly derived from compiler construction. 1. We use flex and bison for the syntax analysis of Anja programs. 2. The program information server is based on attributed abstract syntax trees for programs and annotations. It is generated by the MAX tool (cf. [PH97a]). 3. ANTLR [PQ95] is a compiler generation tool for Java and is used to examine the structure of formulas given as arguments to proof operations. ANTLR is used as it can directly produce Java objects as output.

Integration of the PVS Theorem Prover. As explained above, Jive uses PVS for type checking and for the verification of non-Hoare formulas. We use the techniques described in [But97] for the communication between PVS and the program prover. The connection to the proof system is implemented as a TCP connection with the PVS host system Emacs. Because of restrictions in the interface of the PVS system, our current implementation enforces that the user acknowledges in PVS that a proof obligation sent by the program prover has been received. Support for asynchronous communication would be desirable to reduce the need for user interaction.

Implementation State and Further Work. The current version of Jive enables one to verify specified program properties of Anja programs as described in Section 3 and 4. We implemented tactics to support weak precondition reasoning for statements and tactics for simplifying method calls. As further implementation steps we consider: 1. The development of more powerful tactics to make reasoning more comfortable. 2. Improvements of the user interface, in particular of the text view. 3. Enhancements to the programming logic, in particular for exception handling.

Tool/Language	Lines of code
Java code	13078
MAX specification	2524
C Code	1768
flex & bison specification	1218
ANTLR specification	854
Emacs lisp code	274
PVS standard prelude for Jive	482
Anja standard prelude for Jive	158

5 Conclusion

Verification of program properties for object-oriented programming languages requires tool support, because the formal framework can hardly be handled by hand. In this paper, we presented the architecture of the interactive program verification environment Jive. Jive combines properties of different approaches to software verification. It divides program proving into a program proving and a theorem proving task. This enables the use of existing theorem provers for the proof steps that are not related to the program. For the program proving tasks, we described a technique to implement a given Hoare logic by basic proof operations supporting forward and backward proofs. Based on these proof operations, powerful tactics can be defined. We sketched the main user interface aspects of the system and described some implementation issues. (Current information about Jive can be obtained at www.informatik.fernuni-hagen.de/pi5/jive.html)

Acknowledgments We thankfully acknowledge helpful comments from the reviewers and fruitful discussions with Peter Müller.

References

But97. B. Buth. An interface between Pamela and PVS. Technical report, Universität Bremen, 1997. Available from `http://www.informatik.uni-bremen.de/~bb/bb.html` 76

Com79. Computer Science Department, Stanford University. *Stanford PASCAL Verifier - User Manual*, 1979. 66

COR$^+$95. J. Crow, S. Owre, J. Rushby, N. Shankar, and M. Srivas. *A Tutorial Introduction to PVS*, April 1995. 65

DLNS98. D. L. Detlefs, K. R. M. Leino, G. Nelson, and J. B. Saxe. Extended static checking. Research Report 159, Digital Systems Research Center, 1998. 66

GH93. J. V. Guttag and J. J. Horning. *Larch: Languages and Tools for Formal Specification*. Texts and Monographs in Computer Science. Springer-Verlag, 1993. 64

GMP90. D. Guaspari, C. Marceau, and W. Polak. Formal verification of Ada programs. *IEEE Transactions on Software Engineering*, 16(9):1058–1075, September 1990. 66

Gor89. M. J. C. Gordon. Mechanizing programming logics in higher order logic. In G. Birtwistle and P.A. Subrahmanyam, editors, *Current Trends in Hardware Verification and Automated Theorem Proving*. Springer-Verlag, 1989. Kopiensammlung. 66

JvdBH$^+$98. B. Jacobs, J. van den Berg, M. Huisman, M. van Berkum, U. Hensel, and H. Tews. Reasoning about Java classes. In *Proceedings of Object-Oriented Programming Systems, Languages and Applications (OOPSLA)*, 1998. Also available as TR CSI-R9812, University of Nijmegen. 65

MPH99. P. Müller and A. Poetzsch-Heffter. Modular specification and verification techniques for object-oriented software components. In G. Leavens and M. Sitaraman, editors, *Foundations of Component-Based Systems*. Cambridge University Press, 1999. 65, 68

Owi75. S.S. Owicki. Axiomatic proof techniques for parallel programs. Technical report, Computer Science Dept., 1975. 72

Pau94. Lawrence C. Paulson. *Isabelle: a generic theorem prover*, volume 828 of *Lecture Notes in Computer Science*. Springer-Verlag Inc., New York, NY, USA, 1994. 65

PH97a. A. Poetzsch-Heffter. Prototyping realistic programming languages based on formal specifications. *Acta Informatica*, 34:737–772, 1997. 75

PH97b. A. Poetzsch-Heffter. Specification and verification of object-oriented programs. Habilitation thesis, Technical University of Munich, January 1997. 65

PHM99. A. Poetzsch-Heffter and P. Müller. A programming logic for sequential Java. In D. Swierstra, editor, *ESOP '99*, LNCS 1576. Springer-Verlag, 1999. 69, 73

PQ95. Terence J. Parr and Russell W. Quong. ANTLR: A predicated-LL(k) parser generator. *Software—Practice and Experience*, 25(7):789–810, July 1995. 75

Rei95. W. Reif. The KIV approach to software verification. In M. Broy and S. Jähnichen, editors, KORSO*: Methods, Languages, and Tools for the Construction of Correct Software*, volume 1009 of *Lecture Notes in Computer Science*. Springer-Verlag, 1995. 66