Journal of Universal Computer Science, vol. 5, no. 3 (1999), 208-225
submitted: 1/11/98, accepted: 15/3/99, appeared: 28/3/99 © Springer Pub. Co.

Interactive Verification Environments for Object-Oriented Programs

Jörg Meyer
(Fernuniversität Hagen, Germany
Joerg.Meyer@fernuni-hagen.de)

Arnd Poetzsch-Heffter
(Fernuniversität Hagen, Germany
poetzsch@fernuni-hagen.de)

Abstract: Formal specification and verification techniques can improve the quality of object-oriented software by enabling semantic checks and certification of properties. To be applicable to object-oriented programs, they have to cope with subtyping, aliasing via object references, as well as abstract and recursive methods. For mastering the resulting complexity, mechanical aid is needed.
The article outlines the specific technical requirements for the specification and verification of object-oriented programs. Based on these requirements, it argues that verification of OO-programs should be done interactively and develops an modular architecture for this task. In particular, it shows how to integrate interactive program verification with existing universal interactive theorem provers, and explains the new developed parts of the architecture. To underline the general approach, we describe interesting features of our prototype implementation.

Keywords: Integration of verification systems, program verification, object-oriented programming

1 Introduction

Formal specification and verification techniques can improve the quality of object-oriented software by enabling semantic checks and certification of program properties. They can be used to show the absence of exceptions resulting e.g. from dereferencing null pointers or from out-of-bounds access to arrays. More generally, they can be used to prove that a program satisfies some interface properties or a complete interface specification. To include specification and verification into the program development process, programming environments are needed that provide integrated support for program specification, implementation, and verification. Programming environments supporting these tasks will be called *logic-based* in the following. This article focuses on the use and integration of tools to implement verification support within logic-based programming environments.

Applying formal specification and verification techniques to object-oriented programs has advantages and disadvantages when compared to a procedural setting: The advantages are that object-oriented languages support (a) type-local encapsulation constructs allowing the verification of data type invariants and (b) inheritance enabling code reuse and potential "proof reuse". Both constructs have to be proved very useful for application frameworks and component-based systems. The disadvantage is that subtyping, aliasing via object references, and abstract methods can make verification of object-oriented programs more complex than verification of procedural programs.

The classical technique for the verification of procedural programs is verification condition generation using weakest precondition transformation. Thus, in a first automatic step, the program-dependent aspects are eliminated. This way, the verification task is reduced to proving a formula in some general logic (e.g. first-order predicate logic). This can e.g. be achieved by the help of an interactive general theorem prover. Even for procedural programs that do not make use of pointer structures, the verification conditions can become very large and complex in practice. If proofs fail on first attempts, it is very difficult to locate the problematic program points. I.e. why [Guaspari et al. 1990] proposes more interactive support in program verification for procedural programs as well.

For the verification of object-oriented programs such interactive support is almost indispensable, in particular in the context of implementation independent, i.e. *abstract* specifications. Object-oriented programs make heavy use of method invocations and object references (from a verification point of view, object references cause the same complexity as pointers). In Section 2, we illustrate the resulting problems and argue that verification condition generation is not appropriate to verify abstractly specified object-oriented programs. As a consequence, we look for tactical program provers that support the interactive development of proof outlines that are centered around the program text. Within such tools, weakest precondition transformation is one proof strategy among others.

This article describes integration techniques and a modular architecture for logic-based programming environments. The architecture was designed together with a prototype environment for the specification and verification of sequential Java programs called JIVE[1]. The used specification technique is described in [Poetzsch-Heffter 1997b], the programming logic is presented in [Poetzsch-Heffter and Müller 1999]. For specifying and verifying program-independent properties, general theorem provers, like PVS and Isabelle (see [Crow et al. 1995] and [Paulson 1994]), will be integrated into the environment[2].

The presentation is structured into three sections: [Section 2] outlines our general approach to specification and verification of object-oriented programs and illustrates the special aspects of object-oriented program verification. In particular, we show why the classical verification condition generation technique is not appropriate for object-oriented programs. [Section 3] describes the modular system architecture and discusses design decisions. [Section 4] focuses on the system component that handles program-dependent verification aspects.

2 Specification and Verification of Object-Oriented Programs

This section describes the used specification and verification techniques. Based on this background, it discusses special requirements resulting from the verification of *object-oriented* programs and argues that interactive, program-centered verification is needed to meet these requirements.

[1] Jive stands for **J**ava **I**nteractive **V**erification **E**nvironment [see Müller et al. 1997].

[2] At the time, this paper was written, we used only PVS for proving program-independent properties.

2.1 Interface Specification of Object-Oriented Programs

Formal interface specifications serve three major purposes: 1. They provide formal, declarative, and abstract documentation of program properties. By abstract we mean here that the specification can be done without referring to the hidden implementation parts. 2. They can be used to prove that an implementation of the interface is correct. 3. They provide the basis to verify programs using the interface. Our specification technique builds on the two-tiered Larch approach [see Guttag and Horning (1993)]. Specifications consist of two major parts: (1) Program-independent specifications providing the needed background theories to express the program behavior (e.g., definitions of abstract data types, functions, etc.) and (2) program-dependent parts relating the implementation to general specifications. An interface specification of a class C consists in particular of (a) abstract attributes, (b) a class invariant, and (c) a specification for each public method of C. Class invariants essentially describe properties that have to hold for each object of a class in any state where the object is accessible from outside. Method specifications are given by pre-/postconditions and so-called *modifies-clauses*. In the following, we sketch the specification technique along with a small example (for a detailed description see [Müller and Poetzsch-Heffter 1999]).

[Fig. 1] shows a specification of a Java interface type. The type `Collection` taken from the Java package `java.util` is an interface for implementations of object sets (equality between objects is defined w.r.t. object identifiers[3]). In the interface specification, this is expressed by the abstract attribute `set` of type *ObjectSet* where *ObjectSet* is a sort specified in a general specification framework (e.g. as a PVS specification). Type `Collection` has a method to add objects to the set (the other methods are omitted). The properties of `add` are specified by a pre-/post-pair and a modifies clause. (1) If the parameter `o` is non-null, the result of `add` is true iff the value of `set` in the prestate contains `o`; after execution, `set` contains `o` and the elements it contained in the prestate. (2) The modifies-clause specifies that execution of `add` modifies only the abstract attribute `set` and has no side-effects on other objects.

```
interface Collection {
  abstract ObjectSet set;
  boolean add(Object o);
    pre  o ≠ null
    post result = (o ∈ set^)  ∧  set = {o} ∪ set^
    mod  set
  ...
}
```

Figure 1: Example Specification

[3] I.e. the specification technique is sufficiently strong to handle objects by their identities.

2.2 Relating Interface Specifications and Verification

Modular verification has two sides: (1) Verify the correctness of an implementation w.r.t. its interface specification. (2) Verify properties of programs that use the interface. For both tasks, interface specifications have to be embedded into a programming logic. We use a Hoare-style programming logic because (1) it can be applied quite intuitively, (2) it is sufficiently expressive for our needs, (3) it enables interactive verification close to the programs, which is important in the context of large programs [see Subsection 2.3], and (4) it is fairly well known which is an important prerequisite for industrial application. Our programming logic is essentially an extension of the partial correctness Hoare logic for procedural programs. Details of the logic are given in [Poetzsch-Heffter and Müller 1999].

We demonstrate the embedding of interface specifications into our Hoare logic with the specification of method `add`. To keep things simple, we do not consider class invariants here. The pre-/postcondition is translated into the following triple:

$$\begin{array}{l} \quad \mathrm{f}\ \mathrm{o} \neq \mathrm{null}\ \wedge\ \$(\texttt{this.set}) = \mathit{PRESET}\ \mathrm{g} \\ \texttt{Collection} : \texttt{add(Object o)} \\ \quad \mathrm{f}\ \mathrm{result} = (\mathrm{o} \in \mathit{PRESET})\ \wedge\ \$(\texttt{this.set}) = \{\mathrm{o}\} \cup \mathit{PRESET}\ \mathrm{g} \end{array}$$

The identifier `this` refers to the implicit method parameter. The dollar sign is a global program variable denoting the current object store. Object stores model the state of the objects. Object store OS applied to an attribute location yields the value of the attribute in OS; i.e. $\$(\texttt{this.set})$ denotes the current value of attribute `set`. In addition to this, there are operations to update an object store, to allocate a new object, and to test whether an object is *alive*, i.e. allocated in an object store (see [Poetzsch-Heffter and Müller 1998] for details). The formal meaning of the modifies-clause of method `add` reads: If some object Z is alive in the prestate and an attribute L of Z is not used to implement `this.set`, then it has the same value V in pre- and poststate:

$$\begin{array}{l} \quad \mathrm{f}\ \mathit{alive}(Z, \$)\ \wedge\ Z.L \notin \mathit{down}(\texttt{this.set}, \$)\ \wedge\ \$(Z.L) = V\ \mathrm{g} \\ \texttt{Collection} : \texttt{add(Object o)} \\ \quad \mathrm{f}\ \$(Z.L) = V\ \mathrm{g} \end{array}$$

The set of attributes used to implement `this.set` is denoted by $\mathit{down}(\texttt{this.set}, \$)$ [see Müller and Poetzsch-Heffter 1999]. In summary, the specification of method `add` is formally expressed by two triples.

The details of interface specifications for object-oriented programs are not important for this article. However, it is essential to understand the shape of formulas resulting from such specifications, because the verification environment has to cope with their complexity. The above example gives a sufficient impression of the technical aspects resulting from object stores and aliasing via object references. A further source of complexity is subtyping and recursion. A treatment of these aspects is out of the scope of this article [see Poetzsch-Heffter 1997b].

2.3 Interactive, Program-centered Verification

The classical approach to the verification of procedural programs is based on verification condition generation. This way the proper verification task is shifted to general theorem proving. In this subsection, we argue that this technique — although important and helpful — is not sufficient for the verification of object-oriented programs. It has to be complemented by interactive, program-centered verification support. Thus, logic-based programming environments have to integrate program-centered proof techniques with general theorem proving.

Adapting the weakest precondition transformation to object-oriented programming would mean to provide specifications for all methods and invariants for all loops. Then, verification conditions can be generated and one can try to prove them by general theorem provers. In practice, this technique has already turned out to be problematic for realistic procedural programs without pointers (cf. the work on the Penelope system, e.g. [Guaspari et al. 1990]). The situation in object-oriented program verification is more difficult, because of aliasing in object stores, the extensive use of possibly recursive methods, and the use of sometimes complex abstraction functions.

To illustrate a typical proof pattern where interaction can help, we consider a simplified part of the Java AWT[4]: The class `Component` is the superclass of all user interface components. A `Container` is a component that is used to build tree structures of components. To implement the tree structures, components have an attribute `parent` referencing the component up in the tree; containers have an attribute `children` of type `Collection` referencing the contained components and methods to add and remove components:

```
class Component {
  Container parent;
  ...
}
```

```
class Container extends Component {
  Collection children;
  boolean removeComp(Component c) {...}
  Component addComp(Component c) {...}
  ...
}
```

To verify properties of the AWT (e.g. the correctness of layout algorithms), one has to prove (beside other things) that method `addComp` does not violate the reference structure of the component tree. To illustrate the verification of such properties, we consider the proof that the following formula is invariant under execution of method `addComp`, i.e. that each component is always in the children set of its parent container:

$$\forall \mathit{Component}\ C:\ \$(C.\mathtt{parent}) \neq \mathrm{null} \ \Rightarrow\ C \in \$(\$(\$(C.\mathtt{parent}).\mathtt{children}).\mathtt{set})$$

Let us assume that `addComp` has the implementation given in [Fig. 2], that the precondition of `addComp` guarantees that the parameter `c` is not null, and that we have some appropriate specification for `removeComp`. Even for this simple example, wp-transformation techniques applied to the above formula yield a fairly complex formula of more than ten lines. The complexity mainly results from the two methods calls: Since their specifications do not match the needed postconditions, we have to use adaption techniques (see e.g. [Gries(1981)], Section 12.2) leading to large preconditions. For implementations of realistic size, the generated verification conditions are often unmanageable: (a) They cannot be proven automatically, mainly because of the use of abstractions and the complexity of

[4] Abstract Window Toolkit, the framework to implement graphical user interfaces.

```
boolean addComp(Component c) {
  if(c.parent != null) {
    c.parent.removeComp(c);
  }
  c.parent = this;
  children.add(c);
  return c;
}
```

Figure 2: Implementation of `addComp`

object stores. (b) It is often very painful to prove them interactively, because of their size and because of universal quantifications over object stores. That is why we develop interactive, program-centered verification support.

$\{\ \mathrm{c} \neq \mathrm{null} \wedge \$(\mathrm{c}.\mathtt{parent}) \neq \mathrm{null}\ \Rightarrow\ (\mathrm{c} \in \$(\$(\$(\mathrm{c}.\mathtt{parent}).\mathtt{children}).\mathtt{set}))\ \}$
`if( c.parent != null ) {`
`c.parent.removeComp(c);`
`}`
$\{\ \mathrm{c} \neq \mathrm{null}\ \}$
`c.parent= this;`
$\{\ \mathrm{c} \neq \mathrm{null} \wedge \$(\mathrm{c}.\mathtt{parent}) = \mathtt{this}\ \}$
$\Rightarrow$
$\{\ \exists PS : \mathrm{c} \neq \mathrm{null} \wedge \$(\$((\mathtt{this}.\mathtt{children}).\mathtt{set}) = PS \wedge \$(\mathrm{c}.\mathtt{parent}) = \mathtt{this}\ \}$
——————————————— ↓ [ex.-rule [see Appendix 4]]
V
$\{\ \mathrm{c} \neq \mathrm{null} \wedge \$(\$((\mathtt{this}.\mathtt{children}).\mathtt{set}) = PS \wedge\ \wedge \$(\mathrm{c}.\mathtt{parent}) = \mathtt{this}\ \}$
$\Rightarrow$
($\mathrm{c} \neq \mathrm{null} \wedge \$(\$(\mathtt{this}.\mathtt{children}).\mathtt{set}) = PS \wedge$) (P_1)
$alive(\$(c.\mathtt{parent}), \$) \wedge \mathrm{c}.\mathtt{parent} \notin down(\$(\mathtt{this}.\mathtt{children}).\mathtt{set}, \$) \wedge$
$\$(c.\mathtt{parent}) = \mathtt{this}$ (P_2)
`children.add(c);`
æ œ
$\$(\$(\mathtt{this}.\mathtt{children}).\mathtt{set}) = \{\mathrm{c}\} \cup PS \wedge$ (Q_1)
$\$(\mathrm{c}.\mathtt{parent}) = \mathtt{this}$ (Q_2)
$\Rightarrow$ [interactive reformulation to adapt postcondition]
$\{\ \mathrm{c} \in \$(\$(\$(\mathrm{c}.\mathtt{parent}).\mathtt{children}).\mathtt{set})\ \}$
——————————————— ↑
$\{\ \mathrm{c} \in \$(\$(\$(\mathrm{c}.\mathtt{parent}).\mathtt{children}).\mathtt{set})\ \}$
`return c;`
$\{\ \mathrm{c} \in \$(\$(\$(\mathrm{c}.\mathtt{parent}).\mathtt{children}).\mathtt{set})\ \}$
$\Rightarrow$ [interactive simplification]
$\{\ \$(\mathrm{c}.\mathtt{parent}) \neq \mathrm{null}\ \Rightarrow\ \mathrm{c} \in \$(\$(\$(\mathrm{c}.\mathtt{parent}).\mathtt{children}).\mathtt{set})\ \}$

Figure 3: A proof outline

The scenario for the above proof task in an interactive program verification environment is as follows. The user decides to make a case distinction: (1) $\mathrm{c} = C$ and (2) $\mathrm{c} \neq C$. Since the proof techniques for both cases are similar, we demonstrate only case (1). Starting with the last program statement and the postcondition, the proof outline shown in [Fig. 3] is interactively developed with the system by stepping backward through the program. Roughly speaking, each line corresponds to the application of one proof strategy. The proof outline gives only an overview over the proof; several rule applications are missing. In particular, the precise handling of logical variables is not visible in the proof outline.

At two positions in the proof outline, the user has provided guidance to the system. In the simplification step near the end, he or she has used the knowledge that $(c.`parent`) = `this`, thus that c's parent is non-null. The central step was the adaption of the postcondition of the invocation of `add` to two postconditions specifying `add` [see Subsection 2.2]: Q_1 is an instantiation of the triple resulting from the pre-post-clause; Q_2 is an instantiation of the the modifies-clause embedding. The remaining steps can be done almost automatic by the system.

In this section, we gave a short overview to our specification and verification technique. By the examples, we wanted to show (1.) that the complexity of specification and verification for object-oriented program can hardly be managed without powerful tool support, and (2.) why suitable verification tools should support interactive verification centered around the program text.

3 A Modular System Architecture

This section describes the subtasks, the formal data, and the components of logic-based programming environments. We explain the tool support for the subtasks and how these tools are integrated into a modular architecture for logic-based programming environments. As mentioned in the introduction, the described architecture is realized in the prototype enviroment called JIVE.

3.1 Subtasks

Logic-based programming environments for object-oriented programs have to perform five major tasks:

1. *Specification.* They have to support interface specifications.
2. *Program editing and syntactical analysis.* They should support syntax analysis and static checking of programs.
3. *Program verification.* They should provide flexible and powerful support for interactive verification.
4. *Proving program-independent theorems.* Interface specifications make use of general specification parts (in the example of [Section 2], we used the sort *ObjectSet*). Program proofs lead to program-independent proof obligations. Therefore, an interface to general specification framework and theorem provers must be provided.

As stated above the development of formal correct software is a task, which involves several subtasks like programming, specifying, or verifying. The variety of subtasks reaches from simple information management or working with syntactical program- or formula structures up to theorem proving. User interaction during proofs and development cycles (see example in [Section 2]) complicates the situation. In contrast to batch-oriented proof tools where all subtasks are completed in a sequential style, the sequence of tasks is determined by user interaction. This means for logic-based programming environments that it has to control the different subtasks in order to keep the specification and proof data consistent. This is in particular necessary, if subtasks are delegated to different tools. An example from our environments for this situation is the typechecking of interface specification formulas. They are developed in subtask 1, typechecked with the tool performing subtask 3 using theories of subtask 4. The splitting of operations carries over to splitting up the specification and proof data to different system parts. In the following, we describe the different kinds of specification and proof data managed by our proof environment.

3.2 Specification and Proof Data

Logic-based programming environments combine several different subtasks each of which works with different information. Essentially, they have to handle the following formal data:

- The structure of the program to be proved, i.e the abstract syntax trees.
- The results of static program checks (e.g. type information). Treating object-oriented languages, this information is needed to derive proof obligations which result from subtyping and inheritance.
- The interface specifications of the programs.
- General data types and functions used in the interface specifications.
- Abstraction functions.
- A formalized data and state model for the underlying programming language (in particular the object store).
- Lemmata which simplify properties of the formal object store and the program dependent data types.
- Hoare triples.
- Program proofs.

We group these data into three classes: program-related data, program-independent data, and program proof data. An interactive verification environment has to guarantee the consistency between these different kind of data and has to keep track of all made changes. In the following, we give a short overview of the data flow and data dependencies within our environment.

The scenario of verifying a program starts with a given specified program. In addition to that, the data types and functions used in the interface specification have to be given. From this, two things are generated: (1) A theory describing static program properties which make e.g. type information available for theorem proving and (2) the Hoare-triples embedding the specification into the programming logic (see [Subsection 2.2]). Now the users interactively construct proofs for the generated Hoare-triples. Occurring formulas have to be typechecked according to the general data and function definitions. Implications occurring in weakening or strengthening steps have to be proved using the underlying general theorem prover. [Fig. 4] shows these dependencies. The above described scenario leads to the following requirements for the handling of the specification and proof data by the system:

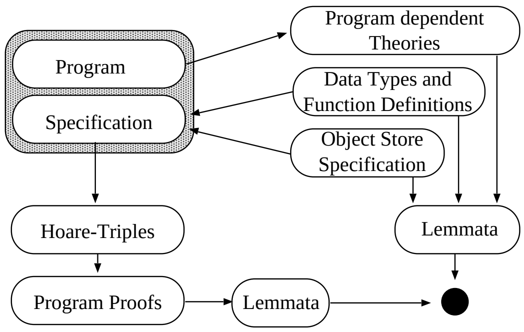

Figure 4: Data dependencies

- All subtasks are related to the general specification language; in particular, the interface specification language, the program annotation language (pre-, postconditions, lemmas) and the subtasks regarding program-independent theories are based on it. Therefore it must be general enough to be usable for all subtasks.
- As Hoare-style proofs are based on the abstract program syntax, it has to be accessible from the different system components.
- Because static program information is needed within theories, we need a parameterization mechanism to adapt the static theories to different programs.
- The use of Hoare-style logic entails to prove lemmas in the general specification framework. To fulfill these proof obligations they have to be embedded into the parameterized static theories during runtime. To combine different theories a module concept for theories is needed.

How the described subtasks and the management of the different data can be integrated within a modular verification environment is explained in the following subsection.

3.3 Architecture and Tools

In this subsection, a modular architecture for logic-based programming environments is presented that fulfills the above requirements. Along with the presentation, we describe the tools that we used to implement our prototype Jive. We start with a discussion about reusing and integrating existing tools versus building a monolithic system.

3.3.1 Monolithic vs. Loosely Coupled Architectures

Logic-based programming environments can either be designed as monolithic systems or as a set of loosely coupled heterogeneous tools. There are four reasons why we designed an architecture for loosely coupled tools: 1. To make the application of logic-based programming environments as simple and convenient as possible, the most powerful, efficient, and automated techniques have to be used. Without reusing implementations for these techniques, such tools would be too expensive. 2. Incorporation of existing tools allows one to benefit from the progress made in each of the related research areas by replacing a component by a new one. This way logic-based programming environments can be customized to the particular requirements of a software manufacturer. 3. A strict separation of tasks eases the adaption and generation of certain system components. 4. In an industrial context, logic-based programming environments have to be integrated into conventional software development systems. I.e., they have to interact with compilers, debuggers, and so on. Thus, loose architectures fit naturally into existing software development environments as these are collections of interacting tools anyway.

As described above logic-based programming environments depend on parameters like the programming language, interface specification language, programming logic, general logical foundations, etc. Existing monolithic systems like the Stanford Pascal Verifier [see Suzuki (1980)] or Penelope [see Guaspari et al. 1990] have fixed those parameters, so a slight change in one of them enforces modifications of the whole system, whereas loosely coupled tool-sets enable the change of system parts. Non-monolithic systems have two major drawbacks: 1. Users have to interact with different components. 2. Data processed by one component has to be made accessible to other components. To deal with the former aspect, a uniform user interface supports the access to all components. The latter problem is solved by providing a management component that serves as information broker.

Taking all arguments of the above discussion into account we developed a loosely coupled system architectures, because it provides more flexibility and allows to integrate existing technologies and systems.

3.3.2 Tools, Components and Interfaces

In [Subsection 3.2], we identified three kinds of information: program-related data, program-independent data, and program proof data. As the first two areas are based on well known techniques, sophisticated existing tools can be used to cover this areas. Programming language dependent tasks are performed by using tools like Max [see Poetzsch-Heffter 1997a], the Cornell Synthesizer Generator [see Reps and Teitelbaum (1989)] or Eli [see Gray et al. 1992]. The mentioned tools are especially practical, because they allow (1.) analysis and generation and (2.) the use of internal data structures during runtime to serve as a program information server. For the formalization of program-independent theories and general theorem proving the use of systems like PVS [see Crow et al. 1995] and Isabelle [see Paulson 1994] is possible. For the handling of program proofs we developed a new tool which provides the needed functionality. It is described in [Section 4].

The main construction problem of a logic-based programming environment is to combine used tools in a suitable tool architecture. According to the different subtasks and data, we identify three main components: The **p**rogram **c**omponent handles all tasks that are center around one program unit. The **t**heorem **p**rover **c**omponent is used for specification and verification of general properties. The **m**anagement **c**omponent keeps track of the consistency of the overall process. In the following, we describe the tasks of each component:

Program Component. The program component PC combines all tasks that directly depend on the programs. It is structured into two subcomponents, the program information server (PIS) and the program proof component (PPC). The PIS supports syntax analysis, context checking, and program specification. It provides type information and structural information about the programs. It extracts the specification from programs, maps it to pre-/post pairs, generates program-dependent theories and provides needed program structure information. We generate the PIS component for ANJA[5] from a ANJA-specification with the Max-system. The program proof component handles interactive Hoare-style and automated strategy based proofs. As the PPC handles the interactive verification aspects, we discuss its features in more detail in [Section 4].

Theorem Prover Component. The purpose of the TPC is to handle program-independent specifications and proofs. The TPC has to fulfill two requirements: (a) It has to support a powerful framework for formulating general, program-independent specifications (e.g., by many-sorted higher-order formulae). Such specifications are needed to express the interface properties of programs and to close the gap between program specifications and more abstract specification layers. (b) It has to provide powerful (interactive) verification support to proof theorems and lemmas about specification properties. Such lemmas can be simply derived specification properties, can result as proof obligations from the application of Hoare-rules, or can relate properties of different abstraction layers.

Management Component. Because the global system state is distributed to the system components, we provide the MC to manage global system information. Beneath technical tasks like control of startup and shutdown of used tools it serves as a basis to manage different programs and their proofs. As object-oriented software development is basically component based, the MC provides management facilities to share proof information between software components. Furthermore, it enables consistency checks between components and the generation of program-proof documentations.

Interfacing the Components. [Fig. 5] summarizes the overall structure of the architecture. The components are integrated based on component interfaces developed in Java. Java enables a graded attachment of tools from socket based communication and the Java native method interface up to Java method invocations depending on how tight tools are coupled to the system. Kernel system parts are implemented in Java using Java method invocations as interface. Loosely coupled parts (e.g. the theorem prover component) use socket connections and can run remotely if necessary. The management component is the central tool controlling these interfaces.

4 The Program Proof Component

Most tools mentioned up to this point build on well understood techniques except the task of the interactive handling of program proofs. This section discusses features of the PPC, a subcomponent of the PC. Program verification systems

[5] Annotated Java, the Java subset we use enhanced by specification parts.

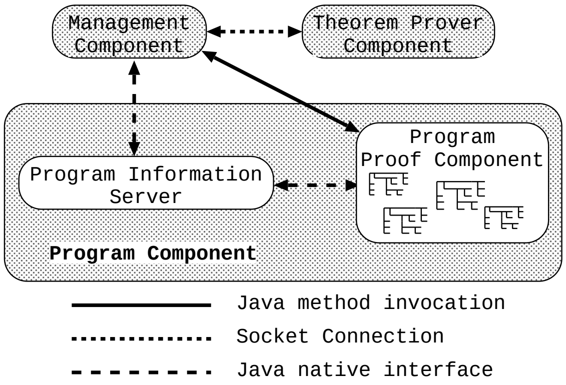

Figure 5: The Jive architecture

are usually based on weakest precondition transformation or eliminate program-dependent aspects by other techniques. The example in [Section 2] has shown that for more complex programming languages, in particular for object-oriented languages, such elimination techniques lead to unmanageable proof obligations. Thus, certain parts of the proofs should be created automatically and some should be performed with user interaction. A similar observation was made in the Penelope project [see Guaspari et al. 1990], even if the class of programs considered there was still relatively simple (e.g. no pointers).

Verification is a fairly complex task. Therefore the user should be provided with the possibility to adapt the systems capabilities to his special problems. Following the idea of LCF, we provide a set of basic operations which are fundamental in the sense that all necessary proof steps (e.g. uses of logical rules, composition of tactics etc.) can be performed using these operations. The use of basic operations enables to give a precise descriptions of proof states and proof state transitions, which is a prerequisite to use techniques as described in [Reif and Stenzel 1993] and [Reif et al. 1998] for the reuse of proofs and proof parts during proof development cycles. These techniques are especially useful for object-oriented programs where reuse of software components carries over to the reuse of proofs.

Compared to general interactive theorem provers, program provers need special features: 1. The proof state is not restricted to one proof tree, but consists of a set of (possibly incomplete) proof trees for a given program (e.g. in a forward proof, proof trees establishing properties of single statements might be combined to a proof tree for a compound statement). 2. Forward and backward proofs should be supported within the PPC: A user may start with annotations of elementary statements, i.e. with axioms, and derive new properties from already proven ones (forward proof), or she/he can state a proof goal as annotation of a procedure or method and stepwise show that the body satisfies this annotation (backward proof). 3. As extensions of programs are essential for reuse in object-oriented languages, a proof environment must be capable to handle proof obligations for later added subtypes [see Poetzsch-Heffter and Müller 1998].

In the rest of this section, we show the requirements to the proof tree data structure for program proofs, show some differences to ordinary theorem provers, and give a short overview over the interaction model of the program proof environment.

4.1 Proof Skeletons

Hoare-logic program proofs are based on the abstract program syntax. The abstract syntax tree is the backbone of program proofs and provides an additional structuring mechanism (a skeleton) for program proofs. Program proofs are represented by trees. Inner nodes in proof trees represent Hoare-triples. Leaf nodes are either Hoare-triples representing an open proof goal or an axiom instance of the programming logic, or formulas representing a proof obligation for the underlying general theorem prover. Because Hoare-triples are statements about program constructs, a program proof tree has references to the abstract syntax tree. The proof trees and the abstract syntax tree are linked via the so called PROG-references [see Fig. 6].

In general, the structure of a program proof is similar to the abstract syntax tree. There are three exceptions, where the structure of program proof trees differs from the abstract syntax tree: 1. Rules whose PROG-reference in the antecedent and in the consequent are the same. In this case the abstract syntax tree node is referred by PROG-references of the antecedent and the consequent which results in a linear proof step [see Fig. 6]. An example for this is the classical

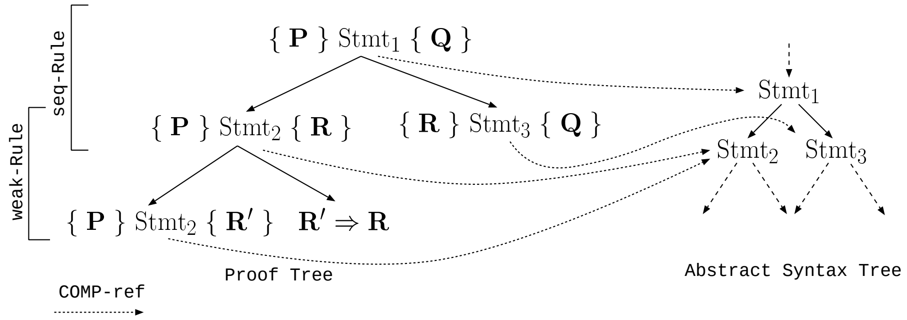

Figure 6: Program Proof Tree and Abstract Syntax Tree

weakening-rule [see Appendix 1]). 2. The programming logic contains rules which require to prove different properties about the PROG-reference of the antecedent. The conjunct-rule serves as an example for this. 3. The PROG-reference refers to a language construct, which does not exist in the underlying program. An example for this are references to inherited methods of classes. If `T:m`, a method declared in type `T`, is inherited by a type `S`, the inherited method `S:m` can occur in rules, although it does not have a counterpart in the abstract syntax tree.

4.2 Modeling the Proof State

The program proof state is modeled via so called *proof container*. Proof containers are the central part of the PPC component and contain every proof (tree) (completed or not) of a proof session. At the beginning of a session, the container is filled with all proof obligations generated from the program specification by the management component. The principle underlying the generation of triples from interface specifications is sketched in [Subsection 2.2]. The generation step also handles class invariants and further aspects of the interface specification language. A program is said to be proved, if all generated proof obligations are fulfilled.

During the proof process, the container is used to store auxiliary proofs. Take the following scenario as an example where such functionality is needed: During a forward proof it is possible to derive properties of programs by combining proof information about program parts. To derive automatically certain properties of a program (e.g. the absence of null pointer references), the user instantiates axioms of the programming logic with parts of the program and stores them into the proof container. The proof system can lookup this information automatically, if it is needed in other proof steps.

Another important property is the sharing of proof parts. Proving forward means to combine proof trees to one proof tree with a new root. According to the interaction model there exist several possibilities to handle such situations: 1. Combine the trees to a tree with a new root and store only the new tree in the proof container. 2. Like 1., but combine copies of the trees. This enables us to reuse this trees for other purposes. 3. Share parts of proof trees between trees. The different possibilities are provided by control operations as described in the next subsection.

4.3 Proof Operations

Proof trees are build from instances of the axioms and rules of the programming logic [see Fig. 6]. The system provides two kinds of basic operations to manipulate the proof state: *control* and *logical* operations. Control operations are those operations that do not contribute to the construction of proofs. They allow to inspect the proof state, to delete parts of a proof, and to input and output proofs. In addition to that, they provide the basic level for the interaction model (see [Subsection 4.4]).

Logical operations correspond to the application of a rule or an axiom. Essentially, there are two ways to apply rules. In a forward proof, they allow to verify new facts from given ones. In a backward proof, they enable to refine a proof goal. For the interactive development of proof outlines, both proof directions are helpful. I.e. why our system supports the manipulation of proof trees at the root as well as at the leaves. As the gained flexibility is constrained by the abstract syntax tree of the program, the increased complexity of the proof state remains manageable by the user.

Since the forward application of a rule needs other parameters and has to fulfill other requirements than the corresponding backward application, the system provides two logical operations for each rule, a forward- and a backward-variant. E.g. consider the conjunct rule given in the appendix. The *conjunct-forward* operation takes two proof trees having the same PROG-reference at the root. The *conjunct-backward* operation splits a goal into two subgoals. It is only applicable if the pre- and postconditions are conjunctions. In general, a logical operation is only applicable, if the parameters fulfill certain requirements, e.g. PROG-references refer the correct types of abstract syntax nodes and the structure of pre- and postconditions coincides with the formula patterns of the rule.

Since the construction of proofs is only based on the logical operations, the correctness of the logical operations is crucial for the correct functioning of the prover. The logical operations are derived from the rules and axioms of the programming logic. As our logic has 26 rules and axioms, this results in 52 logical operations. Each operation is implemented as a Java method thoroughly checking its complex parameters before applying the proof step.

Strategies. The basic proof operations are not meant to be applied by the user. They are only used as atomic operations based on which proof strategies can be defined. Strategies allow to formulate more complex, non-atomic operations on the proof state. E.g., weakest precondition transformation can be expressed as a strategy. Mechanisms allowing to define strategies can be found in most interactive theorem provers. The special aspect occurring in our program prover is that the formulation of strategies needs to refer as well to the abstract syntax tree of the program e.g. to find the last innermost statement in a block.

4.4 User Interaction

Many aspects of our interaction model are inherited from interaction models found in general interactive theorem provers; i.e. through an interactive user interface, the user manipulates a proof state. We assume that there is a current *focus* within the proof tree where the next proof step should be applied. This focus can be moved to other "open" slots in the proof.

Within the area of program proving, visualization and presentation is more complicated than in the area of general theorem proving, because the user has to keep two structures in mind, the program structure and the proof structure. As stated in [Subsection 4.1], the skeleton of program proofs is based on the abstract program syntax. On the other hand the program text simplifies the orientation of the user in the program. To allow a maximum of flexibility at the user interface we provide different views to the proof tree data structure. The views allow representation and interaction with the proof tree structures according to their abstraction level. As demanded in [Nipkow and Reif 1998] this helps to keep proofs comprehensible and human-oriented, which is a prerequisite for user interaction. For our prototype we provide a tree view and a text view.

Tree View. A tree view is a graphical representation of a proof container at the user interface. There are some reasons why a view with a direct interface to the proof tree is essential: 1. They allow the direct manipulation of proof trees by logical or control operations. 2. A proof tree provides complete information of the proof structure whereas textual representations in form of proof outlines tend to hide structural information. 3. Since proof trees are in general big structures, tree views enable to work with scalable clippings of proof trees, i.e. the user interface displays only required information. The currently used tree view is implemented using the Java Swing API.

Text View. A text view provides primarily an interface to the program text. This view is needed for the following tasks: 1. Our interaction model allows to do several different proofs about the same program construct. To start new proofs, the user must have access to the program syntax tree, e.g. to select program parts. 2. For certification and documentation purposes, proof trees can be displayed in a way, that selected proof information is embedded into the program text. The mapping between abstract program and proof syntax and concrete annotated program text is done by the user interface based on information from the program information server and the program proof component.

5 Conclusion

We presented a modular architecture for logic-based programming environments and sketched properties of our prototype environment Jive. The architecture integrates general theorem provers into interactive verification environments for object-oriented programs. The components of the architecture are loosely coupled by interfaces which enable the exchange of proof information. We argued that interactive, program-centered verification is crucial for mastering the complexity of object-oriented programs (or procedural programs with pointers). Within such interactive provers, weakest precondition transformation techniques are realized as proof strategies.

The goal of this research is to provide tools for the formal specification and verification of object-oriented programs. We consider such tools as a base technology for several lines of future research:

- The degree of automation in our system is still too low. To provide more powerful strategies, more sophisticated techniques are needed for dealing with the object store and the modifies-clauses.
- Based on the verification experience, we started to improve the specification and programming methodology in a way that simplifies the verification process.
- An interesting approach would be to combine our fine-grained technology on the level of programs with the more abstract specification techniques used for object-oriented design [see OMG (1997)]. Thus, it could be formally verified that an object-oriented program satisfies its design specification. This would enable certification of software components on the level of design specifications.

Using techniques from compiler construction, we aim to build the system in a generic fashion from descriptions of its different parameters (programming and specification language, logical rules and axioms, general theorem prover). The goal w.r.t. this aspect is to enable flexible enhancements and adaption of the system.

References

[Crow et al. 1995] Crow, J., Owre, S., Rushby, J., Shankar, N., and Srivas, M.: "A Tutorial Introduction to PVS" (1995).

[Gray et al. 1992] Gray, R. W., Heuring, V. P., Levi, S. P., Sloane, A. M., and Waite,W. M.:
"Eli: A complete, flexible compiler construction system"; Communications of the ACM, 35, 2(1992), 121–131.

[Gries 1981] Gries, D.: "The Science of Programming"; Springer, Heidelberg / New York (1981).

[Guaspari et al. 1990] Guaspari, D., Marceau, C., and Polak, W.: "Formal verification of Ada programs"; IEEE Transactions on Software Engineering, 16, 9(1990), 1058–1075.

[Guttag and Horning 1993] Guttag, J. V. and Horning,J. J.: "Larch: Languages and Tools for Formal Specification"; Springer, Heidelberg / New York (1993).

[Müller et al. 1997] Müller, P., Meyer, J., and Poetzsch-Heffter, A.: "Programming and interface specification language of JIVE— specification and design rationale"; Technical Report 223, Fernniversität Hagen (1997).

[Müller and Poetzsch-Heffter 1999] Müller, P. and Poetzsch-Heffter, A.: "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).

[Nipkow and Reif 1998] Nipkow, T., and Reif, W.: "Introduction"; In W. Bibel and P. H. Schmitt, editors, "Automated Deduction–A basis for Application", Kluwer Academic Publishers, 2, (1998), 3–11.

[OMG (1997)] OMG: "Object Constraint Language"; (1997), Available from `ftp://ftp.omg.org/pub/docs/ad/97-08-08.pdf`

[Paulson 1994] Paulson, L. C.: "Isabelle: A Generic Theorem Prover"; Lecture Notes in Computer Science, 828, Springer-Verlag, (1994).

[Poetzsch-Heffter 1997a] Poetzsch-Heffter, A.: "Prototyping realistic programming languages based on formal specifications"; Acta Informatica, 34, 1997, 737–772.

[Poetzsch-Heffter 1997b] Poetzsch-Heffter, A.: "Specification and verification of object-oriented programs"; Habilitation thesis, Technical University of Munich (1997).

[Poetzsch-Heffter and Müller 1998] Poetzsch-Heffter, A. and Müller, P.: "Logical foundations for typed object-oriented languages"; In D. Gries and W. De Roever (editors), "Programming Concepts and Methods", PROCOMET (1998).

[Poetzsch-Heffter and Müller 1999] Poetzsch-Heffter, A. and Müller, P.: "A programming logic for sequential Java"; In D. Swierstra, editor, Lecture Notes of Computer Science, *ESOP '99*, Springer-Verlag (1999).

[Reif and Stenzel 1993] Reif, W., and Stenzel, K.: "Reuse of Proofs in Software Verification"; In R. Shyamasundar, editor, "Foundation of Software Technology and Theoretical Computer Science", Springer LNCS, 761, (1993), 284–293.

[Reps and Teitelbaum 1989] Reps, T. W. and Teitelbaum, T.: "The Synthesizer Generator"; Springer-Verlag (1989).

[Suzuki 1980] Suzuki, N., editor: "Automatic Verification of Programs with Complex Data Structures"; Garland Publishing (1980).

[Reif et al. 1998] Reif W., Schellhorn, G., Stenzel, K., and Balser, M.: "Structured specifications and interactive proofs with KIV"; In W. Bibel and P. Schmitt, editors, "Automated Deduction—A Basis for Applications", Kluwer Academic Publishers, (1998).

Appendix

This appendix contains the Hoare-style rules of the programming logic used within our JIVE system which are referred in the text:

weakening-rule:

$$\frac{\{\ \mathbf{P}\ \}\ \mathrm{PROG}\ \{\ \mathbf{Q}\ \} \qquad \mathbf{Q}\ \Rightarrow\ \mathbf{Q}'}{\{\ \mathbf{P}\ \}\ \mathrm{PROG}\ \{\ \mathbf{Q}'\ \}} \tag{1}$$

conjunct-rule:

$$\frac{\{\ \mathbf{P}_1\ \}\ \mathrm{PROG}\ \{\ \mathbf{Q}_1\ \} \qquad \{\ \mathbf{P}_2\ \}\ \mathrm{PROG}\ \{\ \mathbf{Q}_2\ \}}{\{\ \mathbf{P}_1 \wedge \mathbf{P}_2\ \}\ \mathrm{PROG}\ \{\ \mathbf{Q}_1 \wedge \mathbf{Q}_2\ \}} \tag{2}$$

sequence-rule:

$$\frac{\{\ \mathbf{P}\ \}\ \mathrm{STM1}\ \{\ \mathbf{Q}\ \} \qquad \{\ \mathbf{Q}\ \}\ \mathrm{STM2}\ \{\ \mathbf{R}\ \}}{\{\ \mathbf{P}\ \}\ \ \mathrm{STM1\ STM2}\ \ \{\ \mathbf{R}\ \}} \tag{3}$$

ex-rule:

$$\frac{\{\ \mathbf{P}\ \}\ \mathrm{PROG}\ \{\ \mathbf{Q}[Y/Z]\ \}}{\{\ \exists Z : \mathbf{P}\ \}\ \mathrm{PROG}\ \{\ \mathbf{Q}[Y/Z]\ \}} \tag{4}$$

where Z, Y are arbitrary, but distinct logical variables.