Constructing Verification Environments for Object-oriented Programs

Jörg Meyer and Arnd Poetzsch-Heffter

Fernuniversität Hagen, D-58084 Hagen, Germany
[Joerg.Meyer, Poetzsch]@fernuni-hagen.de

Abstract. Formal specification and verification of object-oriented programs is a complex task as it has to cope with subtyping and aliasing via object references. This complexity cannot be mastered without mechanical support for its subtasks: specification of program interfaces; specification and use of auxiliary data structures; verification of program annotations and of program independent lemmas. This extended abstract presents a modular architecture which integrates specialized system components for these different subtasks. In addition to this it describes in some detail the component that is responsible for interactive verification of program annotation.

1 Introduction

Object-oriented programming languages form a practical important language class; in particular because they support reuse and adaption via inheritance, which have shown to be very useful for application frameworks and component-based architectures. Three reasons make formal verification techniques interesting for oo-programs: 1. Verification techniques are needed for software certification in the developing software component industry which is based on oo-technology. 2. The potential of reuse in oo-programming carries over to reusing proofs. 3. *Formal* verification is important because of the complexity of the underlying languages. Without tool support large program proofs are tedious to handle and can hardly be kept error-free.

Interactive environments that support specification and verification of programs are called *logic-based programming environments* (LBPE) in the following. LBPEs for oo-languages are software tools, which facilitate at least the following tasks: specification of program interfaces; specification and use of auxiliary data structures; verification of program annotations and program independent lemmas. Some of these tasks are highly program-dependent, others can be solved by general theorem provers.

In this extended abstract we describe the modular architecture that we have designed for a programming environment that supports the specification and verification of simplified Java programs. The program environment aims to facilitate specifications as described in [PH97b] and is based on the programming logic

presented in [PHM98]. For specifying and verifying program-independent properties, general theorem provers like PVS and Isabelle (cf. [Pau94] and [COR+95]) will be integrated into the environment.

The presentation is structured into two sections: Section 2 provides an introduction into the system architecture and discusses design decisions. Section 3 focuses on the component that handles program-dependent verification aspects.

2 A Modular System Architecture

Within this section, we describe an architecture for LBPEs. We show in particular what tools are needed for a LBPE and how they can be connected. Finally we sketch some problems arising from tool integration and the use of programming logic.

General Approach. The development of formal correct software is a task, which involves several subtasks like programming, specifying, or verifying. Thus the architecture of a LBPE must be flexible enough to embed all tools solving those subtasks. In particular these systems depend on parameters like the programming language, interface specification language, programming logic, general logical foundations, etc. Monolithic systems like the Stanford Pascal Verifier (cf. [Suz80]) or Penelope (cf. [GMP90]) have fixed those parameters, so a slightly change in one of them enforces modification of the whole system.

To overcome the problems of monolithic systems, we propose a system of loosely coupled components (cf. fig. 1), which offers the following advantages: 1. To make the application of LBPEs as simple and convenient as possible, the most powerful, efficient, and automated techniques should be used. Without reusing implementations for these techniques, LBPEs would be too expensive. 2. Incorporation of existing tools allows one to benefit from the progress made in the related research areas by replacing a component by a new one. 3. A strict separation of tasks eases the adaption and generation of certain system components. 4. In an industrial context, LBPEs 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 a collection of interacting tools anyway.

Components and Subtasks. As shown by fig. 1, we assume three main components. The program component handles all tasks that center around one program unit. The theorem prover component is used for specification and verification of general properties. The management component keeps track of the consistency of the overall process. In the following, we describe the tasks of each component:

***P**rogram **C**omponent.* The program component PC enables program editing, syntax analysis, interface specification and program verification. We collect these jobs within one component, because they are all programming language dependent. The PC is structured into two subcomponents: the program editor (PE)

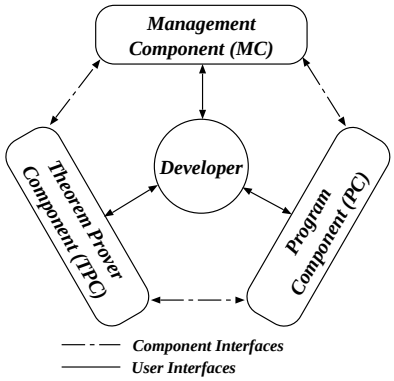

Fig. 1. General system architecture of loosely coupled tools

and the program proof component (PPC). The program editor supports syntax analysis, context check and program specification. The implementation of the PE is generated using the Cornell Program Synthesizer (cf. [RT89]). The program proof component handles interactive Hoare-style and weakest-precondition proofs, i.e. proofs that center around the program text. As the PPC is the central part of LBPEs, we discuss its features in more detail in section 3.

***T**heorem **P**rover **C**omponent.* 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.

***M**anagement **C**omponent.* Because the global system state is distributed to the system components, we provide the MC to manage all system information: 1. Displaying proof status. In particular, the MC is able to decide whether a program is completely verified. 2. Displaying proofs. Usually there are several proofs for each program component (e.g., verification of functional behavior and invariance properties). Furthermore, analysis of proofs helps to detect proof strategies for certain combinations of statements (cf. sect. 3). 3. Program modifications and extensions often require to repeat a proof under slightly changed conditions. The MC provides re-run facilities to adapt existing proofs to modified conditions. 4. Based on programs, specifications, and proof information, the MC can generate complete formal program documentations. 5. Providing flexibility. Storing proof obligations and (possibly incomplete) proofs allows the user to switch

between proofs. Therefore, it is e.g. possible to do a proof in the programming logic and verify all occurring program-independent obligations afterwards, or to interleave these steps.

Problems from Integration. Modular loosely coupled architectures provide more flexibility and allow to integrate existing technologies and systems. However, it causes as well problems that do not occur in monolithic systems. These problems are briefly discussed in the following.

The Theorem Prover Language. The general specification language underlying the TPC is a system parameter that is not only used in the TPC, but as well in the PC: The interface specification language and program annotation language (pre-, postconditions, invariants, etc.) is based on it, in order make resulting proof obligations provable within the TPC. Thus, we have to parameterize the PC with respect to the language of the TPC. To adapt the PC to different TPC's, we investigate generation techniques known from compiler construction (cf. [PH97a]). In addition to this, the communication protocol between the PC and the TPC enables to use services of the TPC for tasks occurring in the PC; e.g. the sort-check of pre- and postconditions can be delegated from the PC to the TPC. In order to enable certain differences between the annotation language used in the PC and the specification language of the TPC, the communication protocol supports syntactic transformations of formulas in both direction of the communication.

Distribution of Specification Parts. The specification of a program consists of several parts that are distributed to different system components: Program parts are specified within the PC using abstract data type definitions, which are usually described within the TPC. For proof purposes these information has to be combined with information about the proof state of other program parts which is stored within the MC. The system architecture has to provide glue to put this things together using the component interfaces.

Modification of Programs, Specifications, and Proofs. We do not assume a fixed program development strategy, but want to enable different development scenarios (top-down, bottom-up, framework-oriented, etc). Flexibility in the development strategy means for LBPEs that they have to support the modification of programs, program specifications, general specifications, and proofs whenever this is appropriate. Thus, the management component must be capable of dealing with invalidating changes and has to coordinate the development steps between the other architecture components. In addition to this, invalidation and reuse of proofs has to be supported with techniques as described in [RS92]. The goal is to guarantee the correctness of proof information and to reuse a maximum of information at program changes.

3 The Program Proof Component

This section discusses features of the PPC, a subcomponent of the PC. Most program verification systems are based on weakest-precondition-transformation or eliminate program-dependent aspects by other techniques. Our experience has shown that for more complex programming languages, in particular for oo-languages, such elimination techniques lead to unmanageable proof obligations. Thus, certain parts of the proofs should be centered around the program text and should enable user interaction. A similar observation was made in the Penelope project (cf. [GMP90]), even if the class of programs considered there was still relatively simple (e.g. no pointers). We focus here on the PPC, because there exist already candidates or techniques to realize the other components[1].

Compared to general interactive theorem provers, programming language provers need special features: 1. The proof state is not restricted to one proof tree, but consists of a set of (incomplete) proof trees for a given program (recall that we consider proof trees for Hoare-logic). 2. The program syntax provides an additional structuring mechanism of proofs; it relates different proof trees, is the backbone for proof strategies (see below), and provides a skeleton for the proofs. Thus, a program proof tree always has to keep references to the abstract syntax tree and vice versa. 3. 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 he/she can state a proof goal as annotation of a procedure or method and stepwise show that the body satisfies this annotation (backward proof).

Proof Trees. Similar to theorem provers, program proofs are organized as trees. Tree nodes represent Hoare-triples and leafs are either (1.) open proof obligations as Hoare-triples, (2.) axiom instances of the programming logic, or (3.) program independent proof obligations. Tree nodes are connected by instances of programming logic rules. A proof tree is constructed by applying *logical* or *control operations* to it. As we support forward and backward proofs, proof trees can be extended at leafs and at the root. Furthermore the assembling of proofs by proof parts is enabled by special properties of the logic and supported by control operations of the proof tool. The correctness of a proof is derived from the correctness of atomic operations, so a proof can be considered as a sequence of atomic operations with a given Hoare-triple as proof start.

The Interaction model. 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. Proof steps consist of either *logical* or *control operations*. Logical operations are the application of rules or axioms to extend the current proof tree whereas control operations enable focus-setting or switching from one proof tree to another.

[1] This does not apply to the MC; but the MC is beyond the interest of this workshop.

Combining Steps to Strategies. Combining of proof steps to strategies allows to formulate different strategies for e.g. forward and backward proofs. As an example, the weakest precondition strategy corresponds to a special strategy, which can be coded by a sequence of operations. In our framework, weakest precondition transformation is considered as a special proof strategy. In particular for the verification of oo-programs, the interaction model of a program verification environment has to support other strategies as well. The main reasons why more flexibility is necessary are as follows: 1. Complexity of oo-languages makes intermediate simplification steps indispensable. 2. Support for different development strategies of specifications and programs. 3. Weakest precondition techniques for languages with subclassing are not fully developed. Moreover special styles of program development could lead to different proof strategies. Thus we need a strong mechanism for the combination of proof operations to strategies.

Examples. The following two examples show the need of flexible strategies in oo- program-proofs: a) The behavior of methods is described by pre-post pairs of the form $\{P\}$ `m()` $\{Q\}$. Proving one triple, usually entails the proof of properties of statements occurring in the body of `m()`, which can be reused in other proofs concerning properties of `m()`. b) Consider two verification scenarios for the verification of a method `m`. 1. We assume specifications of all methods used in the body of `m`. Given the postcondition of `m`, we try to determine a weak precondition. A proof strategy supporting such a scenario would implement a wp-strategy with interactive simplifications. In our logic, the wp-strategy corresponds to a forward proof, i.e. it starts with axioms for elementary statements, exploits method specifications to verify method invocations, uses strengthening in simplification steps, and derives the annotation of compound statements from the annotations of their constituents. 2. We assume a specified method `m`. The goal is to determine the properties of methods used in the body of `m()` that are needed to verify `m`'s specification. Here a backward proof is appropriate, i.e. the proof runs down the syntax tree of `m`'s body.

4 Conclusion

We presented a modular architecture for logic-based programming environments. The architecture was designed to integrate general theorem provers into interactive verification environments for oo-programs. The components of the architecture are loosely coupled by interfaces which enable the exchange of proof information. We discussed how such systems can be adapted to different theorem provers and investigated problems resulting from delegating the tasks to different components.

The goal of this research is to construct generic logic-based environments for oo-languages that can be adapted to different programming languages, programming logics, and theorem provers. Currently, we build the prototype environment JIVE(**J**ava **I**nteractive **V**erification **E**nvironment, cf. [MMPH97]) for a subset of Java.

References

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

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

[MMPH97] P. Müller, J. Meyer, and A. Poetzsch-Heffter. Programming and interface specification language of JIVE — specification and design rationale. Technical Report 223, Fernniversität Hagen, 12 1997.

[Pau94] L. C. Paulson. *Isabelle: A Generic Theorem Prover*, volume 828 of *Lecture Notes in Computer Science*. Springer-Verlag, 1994.

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

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

[PHM98] A. Poetzsch-Heffter and P. Müller. Logical foundations for typed object-oriented languages. In D. Gries and W. De Roever, editors, *Programming Concepts and Methods (PROCOMET)*, 1998. (to appear).

[RS92] W. Reif and K. Stenzel. Reuse of proofs in software verification. Technical report, Universität Karlsruhe, 1992.

[RT89] T. W. Reps and T. Teitelbaum. *The Synthesizer Generator*. Springer-Verlag, 1989.

[Suz80] N. Suzuki, editor. *Automatic Verification of Programs with Complex Data Structures*. Garland Publishing, 1980.