Developing Provably Correct Programs From Object-Oriented Components

Peter Müller

Fachbereich Informatik, Fernuniversität
Feithstr. 140, 58084 Hagen, Germany
Tel: (++49 2331) 987-4870
Email: Peter.Mueller@fernuni-hagen.de

Arnd Poetzsch-Heffter

Fachbereich Informatik, Fernuniversität
Feithstr. 140, 58084 Hagen, Germany
Tel: (++49 2331) 987-304
Email: Arnd.Poetzsch-Heffter@fernuni-hagen.de

Abstract

In object-oriented programming, component-based software construction is supported in four ways: by composition and specialization of classes, code inheritance, and genericity. Formal methods should allow to prove the correctness of a program component based on the specifications of its subcomponents without revisiting the implementations of these subcomponents. Essentially, two problems can occur: 1. Adding a new subtype S may violate the specification of its supertype T; thus components using T may be invalidated. 2. In certain cases, class invariants can be violated by adding new classes to existing components. This positional paper sketches the formal background needed to precisely formulate and investigate the above problems. It claims that behavioral subtyping is a solution to the first problem. As a possible solution to the second problem, it proposes techniques to make interface specifications more expressive, to restrict the form of invariants by semantical constraints (similar to behavioral subtyping), and to refine existing module concepts. It briefly discusses the relation of these techniques to code inheritance and genericity, and relates the problems to other work in the field.

Keywords: Components, formal specification, formal verification, object-oriented programs, behavioral subtyping, semantic modules

Workshop Goals: Discussing different approaches to the topic; learning from the experiences of practitioners; meeting people with similar focus; discussing possible cooperations.

1 Background

Based on a general background in programming language semantics (cf. [PH97a]) and its relation to programming logics, we investigated the formal meaning of interface specifications. This investigation focused on specification techniques for object-oriented programs (cf. [PH95]), because such languages support constructs for component-based programming. In particular, we developed formal foundations for class invariants such that (a) complex linked object structures can be handled and that (b) the invariants can be proved in a Hoare-style logic. Many available object-oriented class libraries work with side-effects; e.g., they support destructive updates on lists. We developed techniques beyond modifies-clauses to express such effects in interface specifications ([MPH97]). In [PH97b], we integrated these ideas within a formal framework for an object-oriented language with recursive classes and recursive methods and provided central verification techniques.

2 Position

The objective of this positional paper is to draw a picture of the techniques needed to prove the correctness of a program-component based on the specifications of its subcomponents without revisiting the implementations of these subcomponents. We claim that the development of such techniques plays a key role in laying the foundation of component-based software and system construction.

We draw the picture for object-oriented programs that may exploit side-effects and destructive updates. We focus on object-oriented programs for three reasons: (1) Object-oriented languages provide support for component-based program development (see below). (2) The role of object-oriented languages in industrial software development becomes more and more important. (3) Encapsulation and explicit interfaces ease the specification and verification of programs. We do not exclude side-effects, because they are at least important for the specification and verification of low level program components and occur in most of the existing class libraries.

Object-oriented programming languages provide means to support component-based program construction: Specialization through subtyping/subclassing, code inheritance, and encapsulation mechanisms to guarantee integrity constraints on data representations. In today's software construction, this support is unfortunately limited to syntactic checks. Programmers are able to invalidate supertype specifications by adding subtypes or invariants of existing classes by adding new classes. As an example for the first problem consider a subtype LS of a list type where method `insert` of LS does nothing. Such misbehavior can in general not be detected by syntactic checking only. As an example for the second problem consider objects of a class C referencing objects of a class D. Assume that the invariant for C states that there is never a chain of references from C objects to C objects. By adding a new subtype of D that has a C attribute this invariant can be invalidated.

We claim that the use of semantic-based specification and verification techniques can overcome the above problems and lead to a systematic construction of correct program components from correct subcomponents. In addition to this, a precise semantical understanding will help to improve programming language constructs towards a better language support for component-based software construction.

Overview The rest of this section elaborates on the relation between component-based software construction and object-oriented programming. After that, we sketch the formal background needed to make later explanations sufficiently detailed. In section 3, we describe the problems involved with the relevant language features and sketch ideas for their solution. An overview of related work is given in section 4. Finally, we summarize expected benefits of our approach.

2.1 Component-Based Software

Our view of component-based software construction is as follows: We assume a set of basic components contained in a software library. Each basic component is a module which contains a set of classes. All classes contained in such libraries are assumed to be formally specified and verified. The interesting task of component-based software construction is to use these basic components to build more complex pieces of software. In particular, we want to study how correctness of such complex components can be derived from the specifications of the used basic components. In terms of modules, we want to prove the correctness of a new module based on the correctness of imported modules.

Object-oriented languages support three techniques to build complex components: (1) Composition, (2) inheritance, and (3) genericity. Composition means to put together different components to use the functionality of all of them. In most cases, existing components have to be customized before they can be combined with other components. This can be done by inheritance and genericity. By means of inheritance, new components can be gained from existing ones by inheriting their code and adding new functionality. E.g., a class for sorted lists can be developed by using code of a simple list class and adding a method `sort`. Genericity can be used to parameterize types. E.g., one can implement a generic class `list` which can be instantiated to lists of integers, booleans, or any other type. We will discuss the effects of these techniques on program correctness in section 3.

2.2 Formal Framework

Formal verification requires a formal semantics of the programming language and of the specifications. In this subsection, we give a quick overview of our framework. In particular, we present a formal semantics of class invariants. This semantics reveals the central problems with modular object-oriented programming. The reader may refer to [PH97b, MPH97] for a detailed description of the formal basis.

Specification and Verification Technique Our specification technique is based upon the two-tiered Larch approach (cf. [GH93]). Program specifications consist of two major parts: (a) A program-independent specification which provides the mathematical vocabulary (e.g., definitions of abstract data types) and (b) a program dependent part that relates the implementation to universal specifications. An interface specification of a class C consists of (a) a specification for each public method of C, and (b) a class invariant. Method specifications are given by pre- and postconditions. Class invariants describe properties that have to hold for each object of a class in any state where the object is accessible from outside.

In the implementation, values of abstract data types are in general represented by several linked objects. We consider such object structures as a whole. Going beyond Larch, the relation between object structures and values of abstract data types is made explicit by so-called abstraction functions (cf. [Hoa72]). Modification of an object X possibly changes the abstractions of all object structures referencing X. As a consequence, the treatment of side-effects becomes very important. The definition of abstraction functions and formal specification of side-effects require a formalization of the data and state model of the programming language. The data model of a programming language defines the objects and values that may be used in programs. The state of an object tells whether the object is allocated, and it assigns an object to each of its attributes. The collection of all object states at a point of program execution is called the current *object environment*.

For verification, we assume a Hoare-style programming logic (cf. [Hoa69]) as presented in [PH97b]. This logic is a formalization of the axiomatic semantics of the underlying programming language. Thus, correctness of a program is showed by translating its specification into Hoare triples and proving these triples in the logic.

Meaning of Specifications We want the class invariant of a class C to hold for all objects of type C in all states where these objects are accessible from outside. Thus, we require invariants to be invariant under execution of public methods. I.e., if the invariants hold for all objects in the precondition of *any public method*, they should hold in the postcondition. In particular, they have to hold for objects created during method execution.

Certainly, invariants need not hold in all intermediate states during execution of C's methods; in particular during the construction of linked object structures, invariants are usually violated. We require invariance only for public methods, because this guarantees that the invariant of a class C holds outside the execution of methods of C and because we want to allow private methods to perform auxiliary operations violating, e.g., well-formedness properties. To use class invariants as invariants in the proof technical sense, they have (a) to be true in possible initial program states and they have (b) to be invariant under *all* public methods. Requirement (a) is trivially satisfied because no user-defined objects are allocated in initial program states. Requirement (b) is the proof obligation resulting from invariants. Although requirement (b) is as well justified from an operational point of view — a method m_C of class C can call a method m_D of class D and thus manipulate D-objects —, the literature often assigns a weaker meaning to invariants which makes verification much more difficult and leads to unintuitive situations.[1]

We define the formal semantics of specifications by interpreting them as triples in a Hoare-style logic. Thus, the connection between specifications and programs is precisely defined. Let us assume a program **P** with classes $C_1 \ldots C_n$. The specification of **P** consists of the class invariants INV_{C_i} and of a pre-postcondition-pair (P_m, Q_m) for each method m. To verify **P**, we have to prove a triple of the following form for each public method m, where \$ denotes the current object environment:

$$\{\, P_m \wedge \bigwedge_{i=1}^{n} INV_{C_i}(\$)\,\} \text{ meth m } \{\, Q_m \wedge \bigwedge_{i=1}^{n} INV_{C_i}(\$)\,\}$$

3 Customization of Components

In this section, we discuss the problems caused by composition, inheritance, and genericity. We give ideas for their solution and present directions for further work in this area.

3.1 Composition

Recall, that specifying a class invariant means that this invariant has to be preserved by all public methods of a program. Assume a module M with classes C_1 to C_m. M does not import any modules, i.e., M is a basic component. Verifying M means to prove the triple above for each public method of M.

Consider a module N which imports M. N contains the classes C_{m+1} to C_n. I.e., the resulting program **P** is a composition of the modules M and N containing the classes C_1 to C_n. What triples have to be shown to verify **P**? Again, we have to prove that every public method of **P** preserves each invariant of any C_i. For most practical applications, this is equivalent to showing that (1)

[1]The weaker meaning requires each method to preserve the invariant of the class it belongs to. Assume two objects X and Y of classes C_X and C_Y that reference a common object Z of class C_Z. Obviously it is possible to violate the invariant of C_Y via methods of C_X and C_Z (which do not need to preserve inv_Y). Thus, we cannot make use of inv_Y during the verification of Y because it may not hold.

C_M::m preserves INV_M, (2) C_M::m preserves INV_N, (3) C_N::m preserves INV_M, and (4) C_N::m preserves INV_N, where C_M::m and C_N::m denote a public method m of a class declared in module M or N, respectively and INV_M and INV_N denote the conjunction of all class invariants in module M or N.

Obligation 1 is fulfilled as M is verified. Obligation 4 causes no further problems as it can be solved locally in module N (maybe using some specifications of M). Obligation 2 means to show properties of an imported module. This is not desirable as the imported code may come from a library and, thus, be not accessible to the verifier. Obligation 3 results in a huge amount of proof obligations for complex components as it contains one triple for each class in the import path of the module. So, in large systems, hundreds or thousands of triples have to be proved. In the next two paragraphs, we will discuss possible solutions to the problems caused by obligations 2 and 3.

3.1.1 Proving Properties of Imported Code (Obligation 2)

If imported code comes from a software library, clients of that code may only have access to interfaces and specifications, not to the implementation. Thus, we have to develop techniques that allow one to prove obligations of the second kind (see above) without revisiting imported code. This can be achieved by two approaches: Preservation of the invariants can be proved using specifications of imported methods, or can be achieved by requiring the invariants of new classes to fulfill certain semantical constraints.

Method Specifications To deduce the preservation of invariants from method specifications, these have to be sufficiently strong and detailed. E.g., if the specification of method m states that m only modifies objects of class C, and if objects of a class D cannot reach C objects in memory, the invariant of D cannot be violated by m[2]. This example shows two important aspects of method behavior: Sharing properties and invariance properties.

Sharing Properties describe how objects in memory are shared by different object structures. E.g., the last object of a singly linked list is shared by all other list objects of the list structure. Specification of sharing properties is crucial for verification because, in general, modification of one object changes the abstract value of all structures that share this object. Basic constructs to express sharing properties are reachability of objects or disjointness of object structures.

These constructs are quite coarse grained. [PH95] shows, that more elaborated techniques can be used to specify sharing properties of certain data representations on an abstract level. Thus, we have to study typical storage layouts to develop such techniques. In particular, these techniques should allow us to prove the absence of sharing of certain objects and thus to deduce the preservation of invariants.

Invariance Properties are used to describe side-effects of methods by specifying which parts of the object environment remain unchanged under method execution. They can be specified in two ways: One can enumerate all objects that might be modified (or created or deleted) under method execution in a so-called modifies-clause. I.e., all objects not mentioned have to be left unchanged. An other possibility is to directly specify which parts of the object environment stay untouched. This can be done by relating the pre- and poststate of the method. [PH97b] introduces two relations between object environments stating that (a) all objects reachable from a certain object remain unchanged or (b) all objects reachable from any object of a certain type remain unchanged.

The advantages of the direct technique are: (1) Sharing can be taken into account. In general,

[2] We assume that invariants may only state properties of reachable objects. An object X is reachable from object Y if there is a chain of references from Y to X.

modification of one object affects the abstract values of all object structures referencing this object. This can hardly be expressed by modifies-clauses. (2) Relations between the pre- and poststate can be included into the usual pre-post-specification and do not require an extra modifies-clause. This makes the use of invariance properties in correctness proofs easier as they fit into the schema of Hoare-triples. (3) Object modification, creation, and deletion can be handled by the same technique. To exploit these advantages, more sophisticated techniques for relating object environments are needed. In particular, we want to specify side-effects in a very detailed and abstract way.

Specification Methodology might be the most important research topic in this area. To bring specification and verification techniques to practice, we need guidelines for programmers telling which properties of methods have to be specified to enable verification. Most work on specification methodology focuses on the specification of functional behavior. But as we showed in the last two paragraphs, verification of component-based software requires the specification of invariance and sharing properties as well. Programmers usually have detailed knowledge about the algorithms they implement. But they are not really aware of side-effects and the sharing properties of the data structures they use. Thus, we have to develop specification techniques that allow them to intuitively specify these properties.

Semantical Constraints Not every first-order formula is acceptable as an invariant. E.g., invariants should not make statements about all allocated objects. In most cases, this would lead to unimplementable specifications. Thus, the expressivity of invariants has to be restricted. As such restrictions are semantical properties, they are called *semantical constraints*. A simple semantical constraint is that the invariant of an object X may only state properties of objects reachable from X.

By putting stronger constraints on invariants, we can achieve that the invariants of new classes cannot be violated by imported methods. E.g., if an invariant only states properties of objects of classes that are declared in the current module (i.e., not imported), this invariant can only be violated by methods of these classes or their subclasses, i.e., not by imported methods.

The development of such semantical constraints will allow one to add new classes to a program and neither have to revisit imported code nor have to prove the preservation of invariants based on the specification of imported methods. Besides the technical aspects, we have to study the influence of such restrictions on programming methodology.

3.1.2 Reducing Proof Obligations for New Code (Obligation 3)

Although it is theoretically possible to prove that every new method preserves the invariant of each imported class, this is not desired because of two reasons: (1) In large systems the proof obligations become too big and, thus, unmanageable. (2) To support encapsulation of implementations, it is desirable to hide implementation-dependent parts of invariants from client modules. In the next two paragraphs, we show how behavioral subtyping can be used to reduce verification effort, and present the idea of semantic module concepts that allow one to drop the proof obligations about *semantically private classes*.

Behavioral Subtyping Informally, behavioral subtyping is defined as follows: S is a behavioral subtype of T, if (1) each public method of S fulfills the specification of the corresponding method of T, in case the self object is of type S, and (2) if the invariant of S implies the invariant of T for all S objects.

Enforcing all subtypes to be behavioral subtypes reduces the effort of verification dramatically as it is sufficient to show the preservation of invariants of classes which are leaves of the subtyping tree. In usual programming languages, determining whether a class is a leaf is not possible until the whole program context — and, thus, the whole subtyping tree — is known. This is never the case for components. Therefore, we require all superclasses to be abstract. I.e., concrete classes can never be superclasses. Thus, all concrete classes are leaves of the subtyping tree. We do not have to prove the invariant of abstract classes as there is no implementation for that class and, thus, objects of this class will never exist. To sum up, enforcing behavioral subtyping reduces verification effort to proving the preservation of the invariant of all concrete classes. As a consequence of this technique, subtyping and code inheritance have to be discerned conceptually. We discuss this aspect in section 3.2.

In the context of semantical program development, enforcing behavioral subtyping is no further restriction to the programmer: Point (1) of the definition above is required anyway because due to dynamic binding, methods of subtypes can be executed in contexts where methods of the supertype were expected. Thus, they have to behave identically in these contexts. Otherwise, the context could not be verified. Aspect (2) is already implied by the semantics of invariants (see section 2.2). One can formally prove that the semantics presented in section 2.2 is equivalent to the proof obligations sketched in the last paragraph (the proof itself is beyond the scope of this paper and, thus, not presented here).

Semantic Modules Syntactic module concepts support the encapsulation of classes. E.g., a module providing a doubly linked list can be implemented based on a public class for the head of the list and a private class C which stores the list elements. As the list can only be accessed via the head, C can be hidden from clients of the module. From a semantic point of view, this is no longer true, as clients of the module might violate the invariant of C or vice versa: A private class C may contain an attribute a of a public type D. Thus, the invariant of C, denoted by inv_C, may state properties of the objects stored in a. All methods of the module M containing C and D preserve inv_C. But, a client module N may contain a subtype of D providing a method that violates inv_C. Thus, we need a module concept which provides *semantically private classes*, i.e., classes that can be hidden from clients of the module in the sense that their invariants are not part of the interface of the module.[3] This helps to solve the two problems sketched above: (1) Semantically private classes do not contribute to the proof obligations for client modules, which eases verification. (2) It is no longer necessary to reveal details of the implementation by exporting invariants which specify properties of the data representation.

Semantical privacy can be achieved by enforcing certain syntactical restrictions of the implementation, by putting semantical constraints on invariants. To illustrate this approach, we introduce a predicate P on types. $P(T)$ holds, iff T is concrete and P holds for the type of each attribute of T. The invariant of a type for which P holds can not be affected by program extensions as subtyping is ruled out. This simple syntactical limitation is of course too restrictive as it forbids the use of subtyping. By combining this idea with a simple constraint on the invariant, the situation becomes better: The invariant of T may only state properties of objects which are reachable via attributes whose types fulfill P. This combination enables semantical privacy and a restricted use of subtyping.

Pushing the constraints on invariants further leads to the following notion: The invariant of a class C has only access to the invariants of reachable objects, not to their implementation. In combination with behavioral subtyping, this constraint assures the preservation of the invariant. The disadvantage of this approach is that it leads to an increasing number of types as showed by the following example: Assume, we have a class `fraction` with two attributes `numerator` and `denominator` of type `int`. The invariant states that the denominator is non-zero. Now we implement a class C which has an attribute a of class `fraction`. We want to specify that the fraction stored in a is non-negative. Because of the constraint sketched above, this can not be

[3]Note, that a class can be semantically private despite being syntactically public.

specified in the invariant of C as it is a property of `fraction`. Thus, we have to build a new class which essentially behaves like `fraction` but has a stronger invariant.[4] This approach fully supports subtyping.

This discussion reveals two interesting research topics: (1) We need an appropriate mix between syntactical and semantical constraints. Therefore, we have to study implementations of software libraries to find the balance between the (partly contradictory) aims of supporting subtyping, providing semantical private types, keeping the amount of types low, and making the whole approach applicable in practice. (2) As shown in the example above, constraining the invariants may lead to lots of types with different specifications but very similar (or identical) implementations. Thus, we need techniques to share code between classes and derive the correctness of an importing class from the correctness of the exporting class. This is discussed in the next section.

3.2 Code Inheritance and Genericity

Recall from section 3.1.2 that we conceptually discern subtyping and code inheritance. I.e., we consider subtyping and inheritance as two different techniques although they may have the same syntax in the programming language.

A code inheritance concept designed to support verification is more than a technique to copy implementations. Specifications and correctness proofs can be inherited along with program code. Consider a method m of class C implemented in module M. If M is verified it is already showed that $C::m$ preserves all invariants of M. Consider a module N with class D inheriting m. To verify D, it is no more necessary to prove that $D::m$ preserves the invariants of M as this was already showed for $C::m$. Thus, verification effort can be reduced.

Like composition, subtyping, and code inheritance, current programming languages support parameterized classes (or templates) on a syntactical level only. We want to derive the correctness of an instantiation of a parameterized class from the class itself and the type parameter. I.e., the aim is to specify and verify the template in a way that correctness is preserved for all possible instantiations of the template. Thus, we require all possible actual type parameters to be behavioral subtypes of the formal parameter type. This asserts that all methods of the type parameter behave in the expected way.

Consider a generic list with a method `sort`. To verify `sort`, it is required for every actual type parameter to provide a method `less`. Again, it is not sufficient to check syntactically the presence of this method. Verification requires to semantically guarantee that `less` establishes an ordering relation on the objects of that type. I.e., all actual type parameters have to be behavioral subtypes of a class defining method `less` with appropriate specification.

In general, genericity is not even well understood on a syntactic level. A lot of work has to be done to study different forms of genericity (e.g., different forms of parameters, like types or methods), and to develop a semantical notion of this technique that goes far beyond textual copies.

4 Comparison

Researchers have only recently begun to study the problems and possible solutions described above. As mentioned in the introduction, related work focuses on the development, specification, and verification of basic components. We summarize this work in the following.

[4]The stronger invariant requires stronger preconditions for some methods, e.g., the multiplication method. Thus, the new class is not a behavioral subtype of `fraction`.

Specification Techniques The two-tiered approach to specifications was introduced by the Larch project (cf. [GH93]). We took this framework as a basis of our work. As described in section 3.1.1, we use a different technique to specify invariance properties which is more appropriate for verification. For the same reason, we use explicit abstraction functions to relate object structures and abstract values. Larch uses implicit abstraction. The formal semantics of specifications used in this paper was presented in [PH97b]. A similar semantics is used in the Larch/C++ language (cf. [Lea96]), based on our work.

Verification Techniques The verification technique assumed in this paper goes back to [Hoa69]. Programming logics for object-oriented languages can be found in [AL97, PH97b]. [AL97] focuses on the soundness of a logic for an object-based language, whereas [PH97b] concentrates on the relation between specification and verification of object-oriented programs. [Lei95] deals with the construction of verified modular programs. As he does not use the rigor formal semantics of invariants, he does not face many of the problems discussed in this paper.

Language Features Since the notion of behavioral subtyping was introduced in [Ame87], much work has been done in this area. [DL96] discusses the relation of behavioral subtyping and specification inheritance. In particular, this approach is interesting in the context of code inheritance (see section 3.2). The separation of subtyping and code inheritance is realized in the Sather language (cf. [Omo94]). The presented technique has no formal semantics and is, thus, not suitable for verification. Sather supports genericity on a syntactical level by enforcing actual type parameters to be *syntactic* subtypes of the formal parameter type and inspired thus the technique sketched in section 3.2. Module concepts of existing programming languages work on a syntactical basis only. None of them supports semantic modules or semantical privacy.

5 Conclusion

Object-oriented languages support the development of software components by means of composition, code inheritance, and genericity. We summarized the problems caused by these techniques in the context of formal specification and verification based on a formal semantics of specifications. Solutions to these problems require further research in the fields of specification, verification, and language support. This will lead to contributions in the following areas: (1) Advanced specification techniques will enable to specify sharing and invariance properties on a high level. (2) Experiences in specification and verification methodology will improve the education of developers and enable the construction of automated verification tools for realistic programs. (3) Semantical module concepts will lead to a better understanding of dependencies between different modules and their specifications and provide more elaborated support of encapsulation. (4) The precise semantical understanding of software components will help to improve constructs for code inheritance and genericity.

References

[AL97] M. Abadi and R. Leino. A logic of object-oriented programs. In Michel Bidoit and Max Dauchet, editors, *TAPSOFT '97: Theory and Practice of Software Development, 7th International Joint Conference CAAP/FASE, Lille, France*, volume 1214 of *LNCS*, pages 682–696. Springer-Verlag, NY, 1997.

[Ame87] Pierre America. Inheritance and subtyping in a parallel object-oriented language. In Jean Bezivin et al., editors, *ECOOP '87, European Conference on Object-Oriented Programming, Paris, France*, pages 234–242, NY, June 1987. Springer-Verlag. Lecture Notes in Computer Science, Volume 276.

[DL96] K. K. Dhara and G. T. Leavens. Forcing behavioral subtyping through specification inheritance. In *Proceedings of the 18th International Conference on Software Engineering, Berlin, Germany*, pages 258–267. IEEE Computer Society Press, March 1996.

[GH93] J. V. Guttag and J. J. Horning. *Larch: Languages and Tools for Formal Specification.* Springer-Verlag, 1993.

[Hoa69] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, 583, 1969.

[Hoa72] C. A. R. Hoare. Proofs of correctness of data representation. *Acta Informatica*, 1:271–281, 1972.

[Lea96] G. T. Leavens. Larch/C++ reference manual. HTML version available from `http://www.cs.iastate.edu/~leavens/larchc++manual/lcpp_toc.html`, July 1996.

[Lei95] K. Rustan M. Leino. *Toward Reliable Modular Programs.* PhD thesis, California Institute of Technology, 1995.

[MPH97] P. Müller and A. Poetzsch-Heffter. Formal specification techniques for object-oriented programs. (To appear in *Informatik Aktuell*, Springer-Verlag), 1997.

[Omo94] S. M. Omohundro. The Sather 1.0 specification. Technical report, International Computer Science Institute, 1994.

[PH95] A. Poetzsch-Heffter. Interface specifications for program modules supporting selective updates and sharing and their use in correctness proofs. In G. Snelting, editor, *Softwaretechnik 95*, 1995.

[PH97a] A. Poetzsch-Heffter. Prototyping realistic programming languages based on formal specifications. (To appear in *Acta Informatica*), 1997.

[PH97b] A. Poetzsch-Heffter. *Specification and Verification of Object-Oriented Programs.* PhD thesis, Technische Universität München, 1997. (Habilitationsschrift).

Biography

Peter Müller is a member of the Lopex project at the University of Hagen, Germany. He works in the field of specification and verification of object-oriented programs. In particular, he studies the verification of component-based programs. Before that, he worked at the Technical University of Munich, where he received a diploma in computer science. Topic of his thesis was the formal semantics of Sather and the development of an operational assertion language.

Arnd Poetzsch-Heffter is Professor at the University of Hagen. Central goal of his research is to improve the programming process. This includes aspects concerning programming techniques and methods, language design, and system support. He received a Doctor in Computer Science from the Technical University of Munich in 1991 with a thesis about a new specification technique for static semantics of programming languages. This research was extended to a tool supported framework for complete language specification ([PH97a]). During a post-doc year at the CS department of Cornell University he developed an approach to integrate program specification and verification techniques for object-oriented programs based on formal language specifications. This is as well the topic of his Habilitation thesis (Technical University of Munich, 1997). Currently, his work concentrates on the construction of provably correct programs from correct components.