7

Modular Specification and Verification Techniques for Object-Oriented Software Components

Peter Müller and Arnd Poetzsch-Heffter

Fernuniversität Hagen, D-58084 Hagen, Germany
[Peter.Mueller, Arnd.Poetzsch-Heffter]@Fernuni-Hagen.de

7.1 Introduction

Component-based software development means reusing prefabricated components, adapting them to particular needs, and combining them to larger components or applications. Reusing components developed by other companies leads to a demand for high-level component specifications and for certification of the component quality. Most quality levels beyond syntactic and type correctness need techniques for formal specification and verification.

Component-based software development requires that the specification and verification techniques can handle modularity and adaptability. *Modularity* means that specifications need to support abstraction from encapsulated implementation aspects, that they remain valid under composition, and that they are sufficiently expressive for verifying properties of composed programs from the specifications of their components. *Adaptability* means that the programming and specification framework support techniques to adapt existing components to the needs and interfaces of other components.

In this article, we develop a formal modular specification and verification framework for OO-components. OO-programming provides a good basis for component technology. It supports adaption by subtyping, inheritance, and dynamic method binding. Classes form an appropriate basic unit for encapsulation and modularity on the level of types. To demonstrate our techniques, we use a Java subset as programming language and assume that a component is described by a package. Syntactically, composition corresponds to the import relation between packages.

The framework builds on specification variables to express abstractions, dependencies between abstract and concrete variables, pre-/postconditions for methods, and so-called modifies-clauses. The article makes two contributions: (1) It presents a formally founded, modular sound solution to the frame problem. (2) It develops a language-supported programming technique to control sharing and structure the object store. The rest of this section introduces the underlying approach and gives an overview of the article.

Specification Technique. Specifications of OO-components have to describe the functional behavior of methods, the effects to the object states, and invariant properties. The functional behavior is specified by pre-post pairs. The state modifications

are specified by a modifies-clause listing all variables that are possibly affected by a method execution. Invariant properties are usually expressed by predicates that have to be maintained by nonprivate methods. However, to focus on the central ideas, we omit type invariants in this article. We assume that all properties necessary for the verification of a method are explicitly specified in the precondition.

In general, specifications cannot refer to concrete fields/attributes of objects: (1) Modularity requires that fields may be private and thus hidden to the user of a component. (2) Adaptability by inheritance and specialization leads to additional fields in subclasses (*extended state*). Since these additional fields are not known in the superclass, they cannot occur in the modifies-clauses of the superclass methods. On the other hand, subclass methods have to satisfy superclass specifications, in particular the modifies-clauses. Thus, without further techniques, a subclass method would not be allowed to modify the fields added in the subclass.

To solve these problems, we use abstract fields (corresponding to specification variables in procedural settings). Abstract fields allow one to express abstractions on the object state and to hide implementation details. The value of an abstract field depends on concrete fields and possibly other abstract fields. These dependencies are explicitly declared in the specification. Having the right to modify an abstract field F includes the right to modify any of the fields F depends on. Since subclasses can introduce dependencies for their new fields, subclass methods can gain the right to modify the extended state without violating the modifies-clause of the superclass.

Composition and Formal Foundation. Composition of packages leads to two particular problems: (1) Due to dynamic dispatch, a package $\mathcal{P}$ might invoke methods that are not present during the verification of $\mathcal{P}$. (2) An abstract field F declared outside $\mathcal{P}$ might depend on a field which is modified by a method of $\mathcal{P}$. Since F is not visible in $\mathcal{P}$, this side-effect cannot be handled within the verification of $\mathcal{P}$. For problem 1, we follow the classical solution and enforce behavioral subtyping. That is, subtype methods have to satisfy the specification of the corresponding supertype methods. This article addresses problem 2. The central contribution is a technique to control dependencies. By structuring the object store, we can exclude unwanted dependencies while retaining a powerful program model.

Attacking the above problems requires a precise notion of the meaning of interface specifications. In our work, the meaning of specifications is founded in an axiomatic language semantics (cf. [PHM98, PHM99]). Based on this semantics, we have proved that our specification constructs allow for modular verification.

Overview. Sec. 7.2 provides the formal foundations for programs and specifications. The specification of functional behavior is sketched in Sec. 7.3. The focus of Sec. 7.4 is on techniques to specify modifications of the object store. Sec. 7.5 presents a new method to control dependencies and sharing. A discussion of related work and the conclusions are contained in Sec. 7.6 and 7.7.

7.2 Foundations for Programs and Specifications

This section summarizes the relevant aspects of the used programming language and the specification primitives, and explains the formal semantics underlying specifications and proofs.

Programs and Specifications. To demonstrate the developed techniques, we use a sequential Java subset that contains classes, interfaces, subtyping, inheritance, dynamic binding, recursive methods, and instance variables. For simplicity, we assume that fields are `private`, and omit static fields.

A software component is assumed to be described by a Java package. A package declares a set of types and can import other packages. A type is declared either as an interface or as a class. As a running example, we use types from a tiny window toolkit, in particular an interface `Component`, representing the abstract behavior shared by all window components, and a class `Box`, representing a rectangular area on the screen (subclasses could for example extend `Box` to contain images):

```
interface Component {
   public abstract Dimension prefSize;
   public abstract Dimension defaultSize;
   public void reset();
   ...
}

class Box implements Component {
   private int width, height;
   private rep X.prefSize by [ $(X.width), $(X.height) ];
   public  rep X.defaultSize by [ 40, 70 ];
   public  void reset()  { width = 40; height = 70; }
   ...
}
```

Components have a preferred and a default size which are specified by the abstract fields `prefSize` and `defaultSize`. Abstract fields are part of the specification and may have types defined in the specification framework. The type *Dimension* in the example represents a pair of integers (denoted by [x, y]). Components provide a public method `reset` to set the preferred size to the default size. `Box` is a simple implementation of `Component`, that is, it inherits the abstract fields and provides implementations for the methods. `Box` declares two concrete fields `width` and `height` to store the dimensions. The first line with the keyword `rep` specifies that the preferred size of a `Box` is the pair of its fields. (We use $ to refer to the current object store in specifications. That is, $\$(X.F)$ denotes the object which is referenced by field F of object X in the current state. Explicitly denoting the execution state clarifies the meaning of specifications when two execution states are involved.) The second representation-clause specifies the default size. To keep the example small, we omitted aspects not related to the size of a component and assume that each type is contained in one package.

A package corresponds to a software component. It composes the imported packages and possibly adds further types. To reflect this view of composition, we require that the import relation is a partial order; that is, mutually recursive types have to be contained in one package. If a package P_1 imports a package P_0, all public program and specification elements of P_0 are visible in P_1, in particular those imported by P_0 from other packages. The *context* of a program or specification element consists of the package it is declared in and all directly or indirectly imported packages. Private program and specification elements are accessible only within their types and must not be used in public elements. For example, the representation-clause of `prefSize` in class `Box` must be private, because it refers to private fields.

In summary, the specification of a type consists of its abstract fields, depends-clauses (shown below), representation-clauses, and the method specifications. A method specification consists of a pre-post pair and a modifies-clause. In this article, we focus on these specification primitives. More elaborated specification constructs are, for example, discussed in [Lea96, PH97].

Formal Foundations. A formal foundation for the specification framework is necessary since (1) the semantics of specifications, in particular of modifies-clauses and dependencies, relies on some subtle points that require formal precision, and (2) a mathematical framework is needed to verify general properties of specification constructs, in particular their modularity properties. As formal foundation, we use an axiomatic semantics of the programming language. The meaning of interface specifications is expressed by triples of the corresponding programming logic (cf. [PH97, PHM99]). For brevity, we focus on the aspects relevant for this article.

To reason about execution states, a formal model of states is required. A state describes (1) the current values for the local variables and for the method parameters, and (2) the current object store. The types and values of the Java subset are formalized by the following data types:

data type

$$\begin{array}{rcl} \mathit{Type} & = & \mathit{booleanT()} \\ & | & \mathit{intT()} \\ & | & \mathit{nullT()} \\ & | & \mathit{ct(\ CTypeId\ )} \\ & | & \mathit{it(\ ITypeId\ )} \end{array}$$

data type

$$\begin{array}{rcl} \mathit{Value} & = & \mathit{b(\ Bool\ )} \\ & | & \mathit{i(\ Int\ )} \\ & | & \mathit{null()} \\ & | & \mathit{ref(\ CTypeId, ObjId\ )} \end{array}$$

$\tau : \mathit{Value} \to \mathit{Type}$

Besides the predefined types `boolean`, `int`, and the type for the `null` reference, there are class and interface types where *CTypeId* and *ITypeId* denote suitable sets of type identifiers. The subtype relation on sort *Type* is defined as in Java and denoted by $\preceq$. Data type *Value* represents the set of values for the types. Values constructed by *ref* represent references to objects. The sort *ObjId* denotes some suitable set of object identifiers to distinguish different objects of the same type. The function τ yields the type of a value.

The state of an object is given by the values of its fields. We assume a sort *Location* for the (concrete and abstract) fields of all objects and functions

$$loc: \ Value \times FieldDeclId \to Location \cup \{undef\}$$

where $loc(X,F)$ is defined as follows: If X is an object reference of a class type with field F, the corresponding location is returned. Otherwise *loc* yields *undef*. In the following, we write $X.F$ for $loc(X,F)$. *FieldDeclId* is the sort of field identifiers where two fields with the same name declared in different classes are represented by distinguished symbols. The type in which a field F is declared is called the *domain type* of F, denoted by $dtype(F)$; the type of F is called its *range type*. Accessibility of field F in type T according to the context conditions of Java is denoted by $accessible(F,T)$. In particular, F is only accessible in T if the domain type of F is imported by T's package. For simplicity, we require abstract fields to be public. To keep formulations short, we will speak of "a location L declared/accessible in type T" meaning that $L = loc(X,F)$ for some object X and some field F where F is declared/accessible in T.

The state of all objects and the information whether an object is alive (that is, allocated) in a program state is formalized by an abstract data type ObjectStore with sort *Store* and the following functions:

$\_\langle\_ := \_\rangle$	: *Store* × *Location* × *Value*	→	*Store*
$\_(\_)$	: *Store* × *Location*	→	*Value* ∪ *SpecSort*
$\_\langle\_\rangle$	: *Store* × *CTypeId*	→	*Store*
new	: *Store* × *CTypeId*	→	*Value*
alive	: *Value* × *Store*	→	*Bool*

$OS\langle L := V\rangle$ yields the store that is obtained from OS by updating concrete location L with value V. $OS(L)$ yields the value of location L in store OS. For concrete locations, the result of $OS(L)$ is of sort *Value*; for abstract locations, the function yields a value of sort *SpecSort* which denotes the supersort of all sorts defined in the specification framework. Object creation is formalized by two functions: $OS\langle TID\rangle$ yields the object store that is obtained from OS by allocating a new object of type *ct(TID)*. $new(OS, TID)$ yields the reference to this object. If V is an object reference, $alive(V, OS)$ tests whether the referenced object is alive in OS.

The value of an abstract location $X.F$ in store OS is determined by its represents-clause. For a represents-clause `rep X.F by r(X,$)` declared in type T, we generate an axiom

$$\forall X, OS : \tau(X) \preceq T \wedge alive(X, OS) \Rightarrow OS(X.F) = r(X, OS)$$

The value of an abstract location $X.F$ can be modified only by updating a concrete location that $X.F$ depends on. In particular, we assume abstract locations not to depend on the liveness of objects. The other axioms of abstract data type ObjectStore are given in [PHM98].

In program specifications, we have to refer to types, fields, and variables, which is enabled by introducing constant symbols for these entities. In particular, we

provide constant symbols for the types (of sort *Type*), fields (of sort *FieldDeclId*), and parameters as well as program variables (of sort *Value*). Furthermore, the symbol $ (of sort *Store*) is used to denote the current object store. A method m of type T is denoted by $T{:}m$.

7.3 Specification of Functional Behavior

A method specification has to cover two aspects: the functional behavior and the frame properties of the method. The specification of the functional behavior expresses the genuine task of a method, in particular the result a method returns. Frame properties describe which parts of the environment are left unchanged by a method execution. In this section, we describe the specification of functional behavior. Specification of frame properties is treated in the next section.

The functional behavior of a method is specified by a pre-post pair. For example, the functional behavior of the method `reset` in interface `Component` could be specified as follows:

```
void reset()
  pre   true
  post  $(this.prefSize) = $(this.defaultSize)
```

This specification demonstrates three important issues: (1) The behavior of `reset` is described in an abstract way without referring to any implementation. (2) The specification of functional behavior relies on frame properties: To reflect the intention that `prefSize` is set to `defaultSize` (and not vice versa), one has to specify that `defaultSize` remain unchanged. (3) Functional behavior may consist of wanted side-effects: `reset` modifies only the abstract field `prefSize` of the `this` object and does not return a value.

Behavioral Subtyping. Since objects of subtypes of a type T may occur in all places where T-objects are allowed, we have to guarantee that subtypes of T behave according to the specification of T. Otherwise we couldn't prove properties of types using T based on the specification of T. For this article, it is sufficient to require that the precondition of the supertype method implies the precondition of the subtype method and that the postcondition of the subtype method implies the postcondition of the supertype method. A detailed discussion of behavioral subtyping can be found in the chapter by Leavens and Dhara and in [DL96].

7.4 Specification and Verification of Frame Properties

This section investigates techniques to specify and verify frame properties. Based on abstract fields with explicit dependencies, we develop a formal semantics of

modifies-clauses which meets the requirements of modular specification and verification. Finally, we discuss the application of the technique and illustrate a remaining problem.

7.4.1 Specification of Frame Properties

After explaining the frame problem, we identify the requirements imposed by modular specification and verification. We describe how explicit dependencies of abstract locations can be used to enhance the modifies-clause technique such that it is suitable for modular software development.

The Frame Problem. Specifications of functional method behavior in general leave large parts of method behavior unspecified. For example, `reset`'s specification does not specify whether the method affects `this.defaultSize` or `this.prefSize`. Since all objects of a program execution share one object store, method executions can have side-effects on the locations of all reachable objects. For verification, it is crucial to have precise information about side-effects. Those aspects of method behavior that concern the absence of side-effects are called the *frame properties* of a method. The problem of specifying frame properties is called the *frame problem.*

Modularity Requirements. If the entire program is known, the specification of frame properties is relatively simple since the developer has complete knowledge about all types and fields. Thus, it is possible to directly specify all locations that are modified by a method execution. Within a modular setting, the frame problem is more complicated for the following reasons:

(i) *Information hiding:* Private fields must not be contained in interface specifications. Thus, abstraction techniques have to be used to specify frame properties.

(ii) *Extended state:* The specification of frame properties must be loose enough to allow overriding methods to modify the extended state. On the other hand, they have to be rigorous enough to guarantee behavioral subtyping (see Sec. 7.3).

(iii) *Open Programs:* A program is called *open* if it can be extended by adding new types. that is, in component-based software development, every component is an open program since it is built to be extended. Therefore, it is never possible to know all types and fields, and frame properties cannot be specified by listing all modified locations.

In the sequel of this subsection, we describe how the modifies-clause technique can be enhanced to cope with these requirements.

Modifies-Clauses. The modifies-clause of a method m describes a set of (concrete and abstract) locations that characterizes all locations that may be modified by m. To cope with the modularity requirements above, we use the following informal meaning of modifies-clauses:

A method m may modify locations mentioned in its modifies-clause and those locations, the locations in the modifies-clause *depend on*.

An abstract location L_a *depends* on a location K if the value of L_a is represented in terms of the value of K. The problems described above can be solved by making these dependencies explicit as suggested in [Lei95]:

(i) *Information hiding:* Instead of mentioning a location $X.F$ with a nonpublic field F in modifies-clauses, it is now possible to use an abstract location depending on $X.F$. Therefore, the permission to modify $X.F$ is granted without violating encapsulation.
(ii) *Extended state:* Subclasses are allowed to introduce new dependencies for an inherited abstract field. Thereby, subtype methods can get the permission to modify fields of the extended state.
(iii) *Open Programs:* This problem will be solved by imposing suitable restrictions on dependencies (see Sec. 7.4.2 and 7.5).

Explicit Dependencies. We require dependencies to be explicitly declared as part of the (public or private) interface of a type. For verification, these declarations are used to generate axioms specifying the depends-relation on locations. Since each program extension may declare further dependencies, the depends-relation in an open program is heavily underspecified.

An underspecified depends-relation has an important advantage over context-dependent depends-relations (that is, depends-relations that are—besides so-called *residues*—fully specified by the dependencies visible in a given context, cf. [Lei95]): Context-dependent depends-relations lead to a context-dependent meaning of modifies-clauses (see Subsec. 7.4.3) which causes two problems: (1) It is difficult to cover context-dependent specifications by Hoare-style programming logics. (2) Proofs for a smaller context do not necessarily carry over to larger contexts; modular soundness has to be proven for such techniques.

For modular verification, the (under-)specification of the depends-relation must guarantee four properties:
1. Consistency with Representation: Explicit dependencies reveal information about the locations that represent the value of an abstract location without giving away its actual representation. To be useful to decide which modifications of the object store might change the value of an abstract location, the value of an abstract location may only depend on locations it is declared to depend on. This is achieved by a proof obligation for every represents-clause.
2. Expressiveness: To prove that a location cannot be modified by a method, it is

crucial to know which locations do not depend on each other. Since the depends-relation is underspecified, this information cannot directly be concluded from the specification of the depends-relation. We have to generate axioms expressing that certain locations do not depend on each other.

3. Modular Soundness: A verification technique is modular sound if all proofs of an open program stay valid for every *admissible* program extension. An extension is admissible if it satisfies behavioral subtyping and the additional axioms for the specification of the depends-relation. In our framework, modular soundness results from using underspecification: The programming logic can only verify properties of a program that hold in all admissible program extensions. To be reasonable, our technique has to ensure that the new axioms are consistent with the existing specification.

4. Modularity: With open programs and arbitrary dependencies it is neither possible to verify a method m w.r.t. its modifies-clause nor can the negation of the depends-relation be specified. Therefore, dependencies have to be restricted to allow for modular verification.

We discuss the above properties along with the formalization of dependencies and modifies-clauses in the next subsection.

7.4.2 Formalization of Dependencies

In this subsection, we present a formalization of the depends-relation.

Declaration of Dependencies. Dependencies are declared by so-called *depends-clauses* of the form `depends` $X.A$ `on` L, where X is a variable of sort *Value* and A is an abstract field. L is a location of the form $X.F$ or $\$(X.c).F$ where F is a concrete or abstract field and c is a concrete field. Following [LN97], we call these forms of dependencies *static* and *dynamic* dependencies, resp. These two forms provide enough expressiveness to specify linked object structures. On the other hand, they are restrictive enough to allow for syntactically checkable modularity rules (see below). The dependencies of an abstract location $X.A$ may be declared in several depends-clauses. Since A describes an abstraction of objects of the domain type of A, depends-clauses for $X.A$ may only occur in subtypes of the domain type of A. The access mode of a depends-clause is the most restrictive access mode of a mentioned field.

Modularity Rules. If a method *T:m* modifies a concrete location L_c, one has to prove that all abstract locations L_a that are affected by this modification are covered by *T:m*'s modifies-clause. To enable such proofs, we have to enforce that L_a is accessible in all types in which L_c is. Since abstract fields have to be public, it suffices to enforce the following *authenticity* rule for every location L: *A location L must be accessible in the domain type of each location it depends on.* The requirement is trivially true for concrete locations, which depend only on themselves.

If program extensions can declare arbitrary dependencies, it is impossible to specify the negation of the depends-relation for an open program. Therefore, we introduce the following *visibility* rule: *If a location $X.F$ depends on $Y.E$, the declarations of the involved depends-clauses must be contained in each package that contains both the declarations of F and E.*

Due to the structure of the depends-clauses, both requirements can be checked syntactically.

Formalization of the Depends-Relation. The depends-relation is formalized by the function

$$\_ \xrightarrow{-} \_ : \quad \mathit{Location} \times \mathit{Store} \times \mathit{Location} \to \mathit{Bool}$$

For the definition of $\longrightarrow$, each depends-clause of the form `depends X.A on X.F` or `depends X.A on $(X.c).F` contributes an axiom of the form

$$\forall X, OS : \tau(X) \preceq T \Rightarrow X.A \xrightarrow{OS} X.F \qquad \text{or}$$
$$\forall X, OS : \tau(X) \preceq T \Rightarrow X.A \xrightarrow{OS} X.c \ \wedge \ X.A \xrightarrow{OS} OS(X.c).F$$

where T is the type in which the depends-clause is declared, L is a location, and OS an object store. We assume $\longrightarrow$ to satisfy these axioms, and to be reflexive and transitive for all object stores.

Consistency with Representation. Based on $\longrightarrow$, we can formalize the proof obligation to check whether the dependencies of a location K are completely declared (see Subsec. 7.4.1). If K is associated with a representation and the modification of a location L affects the value of K, K must be declared to depend on L (the obligation is trivially true for concrete locations):

$$\forall OS, L, Y : \quad OS(K) \neq OS\langle L := Y\rangle(K) \quad \Rightarrow K \xrightarrow{OS} L$$

Negation of the Depends-Relation. The modularity rules allow us to specify the negation for all pairs of locations declared in an open program $\mathcal{P}$ as follows: A location $X_0.F_0$ depends on a location $X_n.F_n$ if there is a sequence of locations $X_0.F_0, \ldots, X_n.F_n$ such that $X_i.F_i$ directly depends on $X_{i+1}.F_{i+1}$. Due to the structure of depends-clauses, X_{i+1} is either identical to X_i or obtained from X_i by reading location $X_i.c_i$. Therefore, we can characterize each path from X_0 to X_n by a regular expression $a_0 \ldots a_{n-1}$ where a_i is either ε or a *FieldDeclId* c_j. The set of all paths from an object X to any object Y such that $X.F$ depends on $Y.E$ can also be described by a regular expression $R(F, E)$. Union and Kleene closure are used to describe alternative and cyclic paths. The visibility rule guarantees that $R(F, E)$ is completely determined by the depends-clauses declared in any context that contains the declarations of F and E. Therefore, $R(F, E)$ can be specified by enumeration for all pairs (F, E) of a given context. We assume we have a function

$$\rho : \mathit{Value} \times \mathit{Value} \times \mathit{Store} \times \mathit{RegExpr} \to \mathit{Bool}$$

that expresses reachability of objects via paths described by a regular expression (for example, $\rho(X, Y, OS, cd) \Leftrightarrow Y = OS(OS(X.c).d)$).

To specify the negation of the depends-relation of an open program $\mathcal{P}$, we generate an axiom for every pair (F,E) of fields declared in $\mathcal{P}$:

$$\forall X, Y, OS : \neg\rho(X, Y, OS, R(F, E)) \Rightarrow \neg(X.F \xrightarrow{OS} Y.E)$$

These axioms are available for verification of type T if all fields named in R are accessible in T.

By the axioms for $\longrightarrow$ and its negation, the depends relation is completely specified w.r.t. all pairs of fields declared in an open program. However, parts of the specification may not be available for verification due to encapsulation.

Consistency of Program Extensions. To avoid contradictory specifications, the axioms generated for a new package P have to be consistent with the axioms generated for the imported packages. The specification of $\longrightarrow$ stays valid since all depends-clauses of an imported package are contained in the extended program. The specification of the negation stays valid since the value of $R(F, E)$ remains unchanged for all pairs of attributes (F, E) declared in an imported package: (1) types of P must not introduce direct dependencies among locations declared in imported types (visibility rule), and (2) types of P must not declare dependencies from locations declared in P on locations declared in imported types (authenticity rule). Therefore, neither direct nor transitive dependencies among locations declared in imported types can be introduced by P. That is, admissible extensions of an open program refine the specification of the depends-relation and its negation.

7.4.3 Formalization of Modifies-Clauses

Syntax. The modifies-clause for a method *T:m* with implicit parameter *this* and explicit parameters $p_1, \ldots, p_n$ has the form `modifies` $M(this, p_1, \ldots, p_n, \$)$ where M is an expression of the specification language of sort *set of Location*. To be able to denote locations reachable from the parameters, M may depend on the current object store. We denote the set of locations of *T:m*'s modifies-clause with parameter objects $P_0, \ldots, P_n$ in store OS by $\mu(T{:}m, P_0, \ldots, P_n, OS)$.

Downward-Closures. If a method is allowed to modify a location K then it may also modify all locations K depends on. Following [Lei95], we call the set of locations on which K depends the *downward-closure* of K. Formally, the downward-closure δ of a set S of locations consists of all locations on which an element of S depends in an object store OS:

$$\delta(S, OS) =_{def} \{L \mid \exists K \in S : K \xrightarrow{OS} L\}$$

Like the depends-relation it refers to, the downward-closure is underspecified for open programs and context-independent. We use the term "a location L is covered by the modifies-clause of a method m in a store OS" to express that L is an element of the downward-closure of m's modifies-clause in OS. We omit the object store if it's clear from the context.

Meaning of Modifies-Clauses. The informal meaning of a modifies-clause M of method *T:m* is: *Every location that belongs to an object that is allocated in the prestate of T:m is either an element of the downward-closure of M or is left unchanged by T:m.* We formalize this meaning by translating a modifies-clause of method *T:m* into a pre-post pair which has to be conjoined to *T:m*'s functional specification. The free variables OS and $P_0, \ldots, P_n$ are used to address the prestate values of the parameters in the poststate. All free variables in specifications are universally quantified over the whole pre-post pair. That is, to verify *T:m* w.r.t. its modifies-clause, one has to prove that *T:m* meets the following pre-post pair for *any* OS, P_i, X, and F:

$$\texttt{pre}\ \ \$ = OS \wedge this = P_0 \wedge \textstyle\bigwedge_{i=1}^{n} p_i = P_i$$
$$\texttt{post}\ alive(X, OS) \Rightarrow X.F \in \delta(\mu(T{:}m, P_0, \ldots, P_n, OS), OS) \vee OS(X.F) = \$(X.F)$$

In addition to this pre-post pair, we require the verifier to show an additional proof obligation for every method invocation: If a method *T:m* invokes a method n, n's modifies-clause in n's prestate has to be a subset of the downward-closure of *T:m*'s modifies-clause in *T:m*'s prestate. (Technically, the requirement is enforced by incorporating an appropriate proof obligation into the invocation rule of the programming logic.) Basically, this requirement is a consequence of the above meaning of modifies-clauses.† However, making this property explicit allows us to prove the modularity lemma below.

Behavioral Subtyping. Like any other pre-post pair, the specification stemming from a modifies-clause has to observe the rules of behavioral subtyping. This can be proved by showing that the modifies-clause of an overriding method is a subset of the downward-closure of the modifies-clause of the overridden method.

7.4.4 Verification of Frame Properties

In the last subsection, we introduced a specification technique for frame properties that is inherently modular sound. However, the usage of an underspecified depends-relation gives rise to the question whether the pre-post pair stemming from a modifies-clause can be proven in a modular way. Therefore, this subsection presents a lemma that allows for modular verification of modifies-clauses.

† It is a slightly stronger requirement for methods that modify a location and reestablish its initial value.

Modularity Lemma. To verify a method *T:m* w.r.t. its modifies-clause, it is sufficient to prove the following pre-post pair.

$$\begin{array}{ll} \texttt{pre} & \$ = OS \wedge this = P_0 \wedge \bigwedge_{i=1}^{n} p_i = P_i \\ \texttt{post} & alive(X, OS) \wedge accessible(F, T) \Rightarrow \\ & X.F \in \delta(\mu(T{:}m, P_0, \ldots, P_n, OS), OS) \vee OS(X.F) = \$(X.F) \end{array} \qquad (*)$$

Proof Sketch. We show that authenticity guarantees that the corresponding property holds for all locations *not* accessible in T:

$$\begin{array}{ll} \texttt{pre} & \$ = OS \wedge this = P_0 \wedge \bigwedge_{i=1}^{n} p_i = P_i \\ \texttt{post} & alive(X, OS) \wedge \neg accessible(F, T) \Rightarrow \\ & X.F \in \delta(\mu(T{:}m, P_0, \ldots, P_n, OS), OS) \vee OS(X.F) = \$(X.F) \end{array}$$

The outline of the proof is as follows: *T:m* can modify a (concrete or abstract) location $X.F$ only by (1) location update or (2) method invocation.†

Assume that *T:m* updates a location L. If $X.F$ depends on L, authenticity ensures that F is accessible in T. Therefore, the above property is trivially true for $X.F$. Otherwise, $X.F$ is not affected by updates of L.

If a method n invoked by *T:m* modifies $X.F$, $X.F$ is covered by n's modifies-clause since n is assumed to meet its specification. As described above, n's modifies-clause in n's prestate has to be a subset of the downward-closure of *T:m*'s modifies-clause in *T:m*'s prestate. Therefore, $X.F$ is also covered by *T:m*'s modifies-clause.

By simple programming logic, the conjunction of the two specifications above yields the pre-post pair stemming from the modifies-clause.

By the modularity lemma, the proof obligation for *T:m*'s modifies-clause can be reduced to the pre-post pair $(*)$. The restriction to accessible locations allows one to rely on a (besides encapsulation) complete specification of the depends-relation and therefore to prove the specification based on the information available in T's package.

7.4.5 Using Modifies-Clauses

In this subsection, we apply the techniques described above to the example of GUI components introduced in Sec. 7.2. We specify frame properties for interfaces and subtypes extending the state of their supertypes, and demonstrate the limitations of the techniques described so far.

Specification Example. Recall that method `reset` of interface `Component` sets the preferred size of a component to a default value. Besides that, it is assumed not to affect the object store:

```
public void reset()  modifies {this.prefSize};
```

† In this context, a constructor call behaves like a method invocation.

To illustrate the modification of extended state, we revisit class `Box`. `Box` stores its dimensions explicitly in the fields `width` and `height`. Thus, X.`prefSize` depends on the `width` and `height` locations of X:

```
depends X.prefSize on X.width, X.height;
```

These dependencies allow `Box:reset` to modify the new fields (that is, the extended state).

Using Imported Types. To discuss a more realistic example of OO-programming, we extend the above program by a class `TextComponent`. `TextComponent` makes use of an imported type `StringBuffer` which implements mutable strings. We assume that `StringBuffer` contains one abstract field `length` representing the length of the string. A text component is supposed to display a string in a fixed font size. Thus, the preferred size of a text component depends only on the length of the string. `reset` sets the string to the empty string:

```
class TextComponent implements Component {
   private StringBuffer text;
   public  rep X.defaultSize by [ 10, 10 ];
   depends X.prefSize on $(X.text).length;
   public  void reset() { text.replace(0, text.length(), ""); }
   public  TextComponent(StringBuffer s) { text = s; }
}
```

This example reveals an important problem: With the new dependencies, `prefSize` is no longer authentic: In `StringBuffer`, the `length` field of `StringBuffer` is accessible, but `prefSize` is not, because the `StringBuffer` package does not import `Component`. The example shows that the authenticity rule forbids abstract fields to depend on fields of imported types, and thus reasonable reuse, which is unbearable for OO-programming.

The authenticity requirement is needed to prevent undetected modifications of fields. For example, a type T might also import `StringBuffer`. If `TextComponent` and T objects share a `StringBuffer` object, a method of T could modify `prefSize` via the shared string undetectably. With techniques that prohibit certain patterns of object sharing, we can refine the authenticity requirement to allow for reuse patterns like the example above. Such techniques are described in the following section.

7.5 Universes: Effective Control of Sharing

In this section, we analyze the situations in which nonauthenticity leads to undetectable modification of fields. To prevent such situations, we enhance Java's type system by so-called *universes*, which can be used to restrict object sharing. Based on these restrictions, we refine the relation between modifies-clauses and program

extensions such that reuse is enabled. Furthermore, we discuss the resulting programming model.

7.5.1 Harmful Patterns of Sharing

The `TextComponent` example in the last section has demonstrated that the authenticity rule is too strong for modular program development. In particular, it entails two restrictions: (1) Abstract locations cannot depend on fields of imported types. Our solution to this problem is described in the sequel of this section. (2) Subclasses must not introduce new abstract locations that depend on inherited locations. If they did, inherited methods could modify these new locations without explicit permission. In the context of this article, the second restriction is a minor problem. Since its solution entails some technical subtleties, it is not presented here.

To reuse prefabricated packages, we definitely have to allow abstract fields to depend on fields of imported types. Imported methods must have the permission to modify the abstract fields by modifying locations they depend on (cf. the use of `StringBuffer:replace` in `TextComponent`). Therefore we have to refine the definition of authenticity and the meaning of modifies-clauses.

If both a nonauthentic location L and the locations it depends on are accessible in the caller of a method m, one can easily detect by verification that L might be modified by m. However, if we permit nonauthentic dependencies on imported fields in a general manner, m might modify a field without giving the caller a chance to detect this modification: Assume a type U which is imported by T and S. As illustrated in Fig. 7.1, T and S objects (depicted by boxes) can both reach a common U-location L (denoted by solid arrows). Nonauthentic locations A and B both depend on L (dashed arrows), that is, they are *codependent*. Let the modifies-clause of method *T:m* contain location A. Since A depends on L, *T:m* may modify L. However, this affects B although B is not contained in *T:m*'s modifies-clause.

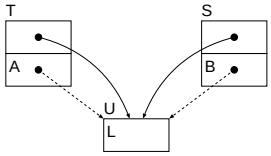

Fig. 7.1. Codependency.

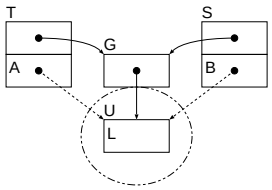

Fig. 7.2. Guards.

To prevent undetected modification of locations, objects can be placed into a so-called *guarded universe* or *universe* for short (denoted by the dashed circle in

Fig. 7.2). Each universe has a guard type G which guarantees that objects outside its universe can modify objects within only by methods of the guard. Thus, guards provide centralized access to guarded objects and can therefore be used to detect the kind of harmful codependencies described above. The next subsection describes universes and guards.

7.5.2 Universes

In the following, we present a type system and formal model for universes. Besides the sharing model enforced by universes, we describe related extensions to Java.

Type System and Formal Model. To restrict sharing, we divide the object store into several universes. We assume there is one instance of each reference type for each universe of a program. Two instances of type T in two different universes U and V are regarded as different types T_U and T_V. Thus, it is not possible to assign objects of type T_U to variables or fields of type T_V and vice versa. However, different instances of one type share a common implementation.

Besides the designated root universe which contains the whole object store, each universe U is associated with a type, the so-called *owner type* of U.† For simplicity, each class is the owner of at most one universe.

Universes form a hierarchic structure: There is one instance of every type for the root universe. Each instance can in turn be the owner of another universe and so on. Every owner of a universe enclosing universe U (including U itself) is called a guard of U. The guard of U that belongs to universe V is called *the guard of U w.r.t. V*. Universes are formalized by the following refinement of the data model described in Sec. 7.2:

data type

Universe = *root()*
| *mkuniv(CTypId)*

Type = *booleanT()*
| *intT()*
| *nullT()*
| *it(ITypId, Universe)*
| *ct(CTypId, Universe)*

univ : *Type* → *Universe*
univ(it(ID, U)) = *U*
univ(ct(ID, U)) = *U*

univ yields the universe a reference type belongs to (as opposed to *mkuniv* which yields the universe owned by a class). To be subtypes, two types have to belong to the same universe: $S \preceq T \Rightarrow univ(S) = univ(T)$. We denote the property that a universe U directly encloses universe V by $V \lhd U$. The reflexive, transitive closure of $\lhd$ is denoted by $\trianglelefteq$.

Accessibility and Notation. Universes are private to their owner types: A type T_U of universe U is accessible only in other types of U and the owner type of U.

† Technically, it would be possible to associate universes with owner packages instead of types. That would grant more flexibility but leads to a more complex formal model for universes.

In particular, T_U must not be used in signatures of nonprivate methods or as range types of nonprivate fields in U's owner type.

Since each type T has access to at most two universes (*univ(T)* and *mkuniv(T)*), we can use unprimed and primed type identifiers to refer to types of the enclosing and the owned universe, respectively. Note that an unprimed type of a universe U is identical to the primed type in the universe directly enclosing U. Since Java does not provide references to values, there is no need for primed versions of primitive types (for example, `int`). An occurrence of a primed type T' in a type S refers to the implementation of T in the universe owned by S. That is, all unprimed reference types in T's implementation are primed types w.r.t. S (there is no correspondence for primed types of T in S, since those types are not accessible in S).

In the following refined version of `TextComponent`, we store `StringBuffer` objects in a separate universe owned by `TextComponent`. Since primed type names must not occur in signatures of nonprivate methods/constructors, the constructor takes an unprimed `StringBuffer` object (see below for the constructor body):

```
class TextComponent implements Component {
   private StringBuffer' text;
   ...
   public TextComponent(StringBuffer s) {...}
}
```

Sharing Model. Universes guarantee the following invariant for every execution state: *If object X reaches object Y by a chain of references, then the universe of the type of X is equal to or encloses the universe of Y's type.* The invariant holds because (1) objects of type T_U (belonging to universe U) can be created only by types of U and U's owner type, and (2) a reference to an object of T_U cannot be passed to an object of a type not belonging to U except to U's owner type. Both properties are a direct consequence of the type and accessibility rules described above. For brevity, we omit a formal proof of this property.

The structure of the depends-clauses enforces that a location $X.F$ can only depend on $Y.E$ in state OS if X reaches Y in OS, and therefore, the universe of the type of X is equal to or encloses the universe of Y's type.

Language Extensions. To support universes, the Java type system has to be extended as described above.

Clone Operation. Since the universe type system does not allow one to pass references across universe boundaries (except to objects of owner types), we provide a special clone operation to still be able to exchange data across universe boundaries. This operation (1) clones a whole object structure (that is, performs a deep copy) and (2) moves the new structure from one universe to another. We provide an *upclone* and a *downclone* operation to clone objects and transfer the result from the primed to the unprimed universe respectively vice versa. These operations are

denoted like method invocations (for example, `x.upclone()`). The static type of a clone operation is determined from the static type of its argument (by priming or unpriming).

In `TextComponent`, the constructor has to clone the `StringBuffer` parameter and move it to the primed universe: `text = s.downclone();`

Translation into Java. Although universes require some changes to the programming language, they are rather a specification technique than a language modification. Programs using universes can easily be translated into ordinary Java programs with identical behavior. To do that, we replace every primed type by its unprimed version and the clone operations by an invocation of the method `clone` with an appropriate cast. The `clone` method has to guarantee that its result is completely disjoint from its argument which can be enforced by a suitable specification for `Object:clone`. The result of the translation is an ordinary Java program which at runtime adheres to the partition of object stores enforced by universes.

7.5.3 A Weaker Definition of Authenticity

Universes have been introduced to enable reuse without permitting undetected modification of locations: Instead of simply importing a type T, a client imports T into a separate universe. Since the client is the guard of that universe, it controls access to T. To support this reuse pattern, the authenticity rule and the meaning of modifies-clauses have to be refined.

Authenticity. Locations outside a universe U must be allowed to depend on locations inside. With the above definition of authenticity this is not permitted since fields outside a universe are not accessible in types inside. An object of universe V can modify locations of a universe U enclosed by V only via invoking a method m of U's guard w.r.t. V. To meet its specification, m has to guarantee in particular that V-locations are not modified undetectedly. Therefore, every location of V depending on a U-location must be accessible in the guard of U w.r.t. V.† This is achieved by the following refined authenticity rule. *If $X.F$ depends on $Y.E$ and U and V are the universes containing the types of X and Y then: (1) if $V = U$, then $X.F$ must be accessible in the domain type of E, and (2) if $V \triangleleft U$, then $X.F$ must be accessible in the guard of V w.r.t. U.* Requirement 1 corresponds to the old authenticity requirement. Requirement 2 enforces the accessibility in guards described above. Both requirements can be checked syntactically.

† Recall that the sharing model of universes guarantees that $V \trianglelefteq U$ (cf. Subsec. 7.5.2).

Negation of the Depends-Relation. To avoid contradictions, the axioms generated for the negation of the depends-relation have to be adapted to the weaker form of authenticity:

$$\forall X, Y, OS : univ(\tau(X)) = univ(\tau(Y)) \wedge \neg\rho(X, Y, OS, R(F, E)) \Rightarrow \neg(X.F \xrightarrow{OS} Y.E)$$

The new visibility rule only has to hold for pairs of locations belonging to the same universe. With the new rules, it is again guaranteed that program extensions do not introduce axioms contradicting the specification of imported packages (by the argument described in Subsec. 7.4.2).

Modifies-Clauses. Methods of a type inside a universe U must be allowed to change abstract locations outside by modifying locations they depend on, even if the locations outside are not covered by the modifies-clause. Therefore, we have to exclude locations of outer universes from the proof obligation stemming from the modifies-clause of *T:m* (cf. Subsec. 7.4.3):

`pre` $\$ = OS \wedge this = P_0 \wedge \bigwedge_{i=1}^{n} p_i = P_i$
`post` $alive(X, OS) \wedge univ(dtype(F)) \trianglelefteq univ(T) \Rightarrow$
$X.F \in \delta(\mu(T{:}m, P_0, \ldots, P_n, OS), OS) \vee OS(X.F) = \$(X.F)$

The adaption of the modularity lemma (cf. Subsec. 7.4.4) to the refined semantics of modifies is straightforward. The adapted proof makes use of the sharing properties enforced by universes (see Subsec. 7.5.2). In particular, it is important that methods of a type in universe U can only be invoked by types of U or U's owner type. If a type invokes a method of the same universe, the proof is as in Subsec. 7.4.4. Otherwise, an owner type invokes a method of the universe it owns. In that case, the new definition of authenticity guarantees that all abstract locations that might be affected by the method invocation are either declared in the modifies-clause of or accessible to the caller. Thus, we have traded flexible reuse for a slightly weaker semantics of modifies-clauses.

Example. In the text component example, the dependency

```
depends X.prefSize on $(X.text).length;
```

is now authentic because (1) `prefSize` is accessible in `TextComponent`, which is the domain type of `text`, and (2) the `length` field belongs to the primed universe of `TextComponent` (since the range type of `text` is `StringBuffer'`). That is, `TextComponent` is the guard of `length` w.r.t. the universe of `prefSize`. Therefore, `TextComponent` has a legal specification.

To modularly verify `TextComponent`, one has to guarantee that objects of type `StringBuffer'` cannot be shared among several text components. Otherwise, `reset` would modify the `prefSize` field of all text components sharing a `StringBuffer`. Such fine-grained sharing control can be achieved by appropriate type invariants, which are not dealt with in this article.

7.5.4 Programming Model

Universes allow for modular verification by restricting sharing of object structures. In this subsection, we describe the influence of universes on the programming model and discuss the limitations of the approach.

Authenticity requires using universes when a location depends on a location declared in an imported type. This requirement reveals three aspects of universes: (1) Universes are only required when there actually are dependencies. For example, container data structures such as lists of objects usually don't depend on locations of contained objects and can therefore be implemented conventionally. (2) Universes are not required when both the source and target location of a dependency are declared in the same package. In particular, this allows one to implement (mutually) recursive types. Therefore, universes provide far more flexibility than a part-of relation. (3) In addition to the situations where universes must be used to achieve authenticity, they are helpful to ease verification of many implementation patterns.

Authenticity and the use of universes, as described above, impose two limitations on programs: (1) With the universe model presented in this article, it is not possible to share universes and their locations among objects of different types (since universes are private to types). However, the accessibility of universes can be extended to packages (we omitted this generalization to obtain an easier formal model). This allows for sharing of universes among objects of types declared in the same package, and provides enough flexibility for most implementation patterns. (2) The clone operation is required to exchange information across universe boundaries. Besides the runtime overhead to copy objects, information about object identities is lost by cloning.

In summary, the authenticity requirement and universes do not enforce a major shift in the programming model: Universes are only required in designated situations and do still allow for many sharing patterns.

7.6 Related Work

Combinations of formal techniques and OO-concepts have been investigated w.r.t. different development and abstraction levels ranging from requirement and design specification languages (cf. for example [DvK92, CDD+89]) to executable assertions extending OO-programming languages ([Mey92]). Our approach lies in the middle of this range. Like the former frameworks it uses declarative techniques† to formalize pre- and postconditions, and aims to support complete specifications of functional properties. However the foci are different. Design frameworks concentrate on development steps (for example, refinement techniques, [LH92]). We want to specify and verify interfaces of components implemented in existing programming languages. Consequently, we have to deal with the less abstract semantics of pro-

† We use first-order logic in contrast to the model-based approach of VDM and Z.

grams, in particular with sharing which is essential in practice, but often neglected in design languages.

Our specification technique builds on the two-tiered approach introduced by Larch (cf. [GH93]). To be suitable for verification, we provide explicit abstraction. The formal semantics of interface specifications used in this article is presented in [PH97]. A similar semantics is used in the Larch/C++ language (cf. [Lea96]). The programming logic on which our specification technique is based is presented in [PHM98, PHM99]. The rest of this section discusses research related to the specific contributions of this article.

Frame Properties. [Lei95] provides a basis for our work by introducing explicit dependencies and downward-closures for modifies-clauses. In his approach, the translation of a modifies-clause into a pre-post pair depends on the context in which the translation takes place. Therefore, it is difficult to handle this semantics by Hoare-style programming logics. Furthermore, Leino's soundness proof does not yet cover dynamic dependencies.

An alternative approach to the extended state problem is presented in [Lei98]. Instead of abstract fields, so-called data groups are used to represent a set of concrete fields. Like abstract fields, data groups can be mentioned in modifies-clauses and provide support for information hiding and modification of extended state. They are a natural way to reflect a programmer's intention. In contrast to abstract fields, data groups do not have a value. This allows one to drop the authenticity requirement. But on the other hand, data groups cannot be used to specify functional behavior in terms of abstract values which is crucial for verification of OO-programs.

Type Systems for Sharing Control. [CPN98] and [Alm97] present type systems (so-called ownership types and balloon types) that can be used to partition the object store (into contexts and balloons, respectively) and provide flexible alias control. In contrast to universes, contexts and balloons are owned by objects instead of types which allows for a more fine-grained sharing control. However, our intention was to clarify the requirements for modular verification. Therefore, we wanted to use as little syntactic support as possible to put the focus on the essential semantics. Although type invariants can be used to express sharing properties, elaborated type systems can be used to dramatically decrease the verification effort. Therefore, they are a good supplement to the techniques presented in this article.

7.7 Conclusions

We presented modular specification and verification techniques for OO-software components. To enable modularity, specifications have to support abstraction from encapsulated implementation aspects, remain valid under composition, and be sufficiently expressive to verify properties of composed programs from the specifications

of their components. Concerning functional behavior of methods, behavioral subtyping is a suitable technique to enforce modularity of specifications and proofs.

Information hiding, the extended state problem, and program extensions make modular specification of frame properties delicate. Based on abstract fields with explicit dependencies, we enhanced the modifies-clause technique to meet these modularity requirements. Our semantics enables one to verify a method w.r.t. its modifies-clause locally in the context of the method. Authenticity guarantees that locations declared outside this context are not modified by the method.

Since authenticity is too strong to allow for effective reuse, we introduced a type system providing universes. By preventing certain patterns of sharing and codependencies of locations, universes allow us to weaken the authenticity requirement and enable reuse. The price for these benefits is that object structures have to be cloned to move data from one universe to another. Furthermore, objects usually cannot be shared among data structures implemented in different packages.

In a nutshell, the described techniques allow for modular specification and verification of functional behavior and frame properties. They can be extended to cover type invariants. Therefore, they provide a basis for formal treatment of object-oriented software components.

Acknowledgments

We'd like to express our gratitude to the anonymous reviewers who provided extensive and very valuable comments. Thanks also to Gary T. Leavens for his comments on an earlier version of this article.

Bibliography

[Alm97] Almeida, P. S. Balloon types: Controlling sharing of state in data types. In Akşit, M. and Matsuoka, S., editors, *ECOOP '97: Object-Oriented Programming*, volume 1241 of *Lecture Notes in Computer Science*, pages 32–59. Springer-Verlag, 1997.

[CDD+89] Carrington, D., Duke, D., Duck, R., King, P., and Rose, G. *Object-Z: an Object Oriented Extension to Z.* North-Holland, 1989.

[CPN98] Clarke, D. G., Potter, J. M., and Noble, J. Ownership types for flexible alias protection. In *Proceedings of Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, volume 33(10) of *ACM SIGPLAN Notices*, October 1998.

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

[DvK92] Durr, E. H. and van Katwijk, J. VDM++: A formal specification language for object-oriented design. In *TOOLS Europe '92*, pages 63–77, 1992.

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

[Lea96] Leavens, G. T. An overview of Larch/C++: Behavioral specifications for C++ modules. In Kilov, H. and Harvey, W., editors, *Specification of*

Behavioral Semantics in Object-Oriented Information Modeling, chapter 8, pages 121–142. Kluwer Academic Publishers, Boston, 1996.

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

[Lei98] Leino, K. R. M. Data groups: Specifying the modification of extended state. In *Proceedings of the 1998 ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA '98)*, volume 33(10) of *ACM SIGPLAN Notices*, pages 144–153, October 1998.

[LH92] Lano, K. and Haughton, H. Reasoning and refinement in object-oriented specification languages. In Madsen, O. L., editor, *ECOOP '92 European Conference on Object-Oriented Programming*, volume 615 of *Lecture Notes in Computer Science*, pages 78–97. Springer-Verlag, 1992.

[LN97] Leino, K. R. M. and Nelson, G. Abstraction and specification revisited. The manuscript KRML 71 can be obtained from the authors, 1997.

[Mey92] Meyer, B. *Eiffel: The Language.* Prentice Hall, 1992.

[PH97] Poetzsch-Heffter, A. Specification and verification of object-oriented programs. Habilitation thesis, Technical University of Munich, January 1997. URL: `www.informatik.fernuni-hagen.de/pi5/publications.html`.

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

[PHM99] Poetzsch-Heffter, A. and Müller, P. A programming logic for sequential Java. In Swierstra, S. D., editor, *Programming Languages and Systems (ESOP '99)*, volume 1576 of *Lecture Notes in Computer Science*, pages 162–176. Springer-Verlag, 1999.