# Modular Specification of Frame Properties in JML

Peter Müller, Arnd Poetzsch-Heffter, and Gary T. Leavens

TR #02-02a
February 2002, Revised October 2002

**Keywords:** frame property, frame axiom, modifies clause, depends clause, pivot object, rep exposure, argument exposure, ownership type model, universe type system, data abstraction, aliasing, mutation, modularity, relevant location, specification, verification, Java language, JML language.

**2000 CR Categories:** D.2.1 [*Software Engineering*] Requirements /Specifications — languages, tools, JML; D.2.2 [*Software Engineering*] Design Tools and Techniques — Object-oriented design methods; D.2.3 [*Software Engineering*] Coding Tools and Techniques — Object-oriented programming; D.2.4 [*Software Engineering*] Software/Program Verification — Class invariants, correctness proofs, formal methods, programming by contract, reliability, tools, JML; D.2.7 [*Software Engineering*] Distribution, Maintenance, and Enhancement — Documentation; D.2.13 [*Software Engineering*] Reusable Software — Reusable libraries; D.3.2 [*Programming Languages*] Language Classifications — Object-oriented languages; D.3.3 [*Programming Languages*] Language Constructs and Features — Abstract data types, classes and objects, modules, packages, polymorphism, procedures, functions, and subroutines; F.3.1 [*Logics and Meanings of Programs*] Specifying and Verifying and Reasoning about Programs — Assertions, invariants, logics of programs, pre- and post-conditions, specification techniques;



An earlier version of this report is ISU TR #01-03, and was presented at the Workshop on Formal Techniques for Java Programs associated with ECOOP 2001. This paper is an expanded version of the earlier one.

Department of Computer Science
226 Atanasoff Hall
Iowa State University
Ames, Iowa 50011-1041, USA

# Modular specification of frame properties in JML

Peter Müller[1§], Arnd Poetzsch-Heffter[2¶], and Gary T. Leavens[*3‖]

[1] *Deutsche Bank AG, Frankfurt, Germany*
[2] *Universität Kaiserslautern, 67653 Kaiserslautern, Germany*
[3] *Iowa State University, Ames, Iowa, 50011-1040, USA*

**SUMMARY**

**We present a modular specification technique for frame properties. The technique uses modifies clauses and abstract fields with declared dependencies. Modularity is guaranteed by a programming model that enforces data abstraction by preventing representation and argument exposure, a semantics of modifies clauses that uses a notion of "relevant location," and by modularity rules for dependencies. For concreteness, we adapt this technique to the Java Modeling Language, JML.**

## 1. Introduction

In an interface specification language [31], a *frame property* describes what locations a method may modify, and, implicitly, what locations it may not modify [2]. This is often specified using a *modifies clause* in a method specification [10, 31].

Such modifies clauses can be written in the Java Modeling Language (JML) [16, 15], which we use for examples in this paper. We use JML because it is a modern interface specification language, that supports Leino's notion of declared dependencies [18, 19, 20]. Such dependencies connect abstract (specification-only) fields with a program's concrete fields. In the same way that interface specification languages in the Larch family [10, 31] are tailored to particular programming languages, JML is tailored to the specification of Java classes and interfaces, and hence is useful for mathematically precise documentation and verification [12]. However,

*Correspondence to: Dept. of Computer Science, Iowa State University, 229 Atanasoff Hall, Ames, Iowa, 50011-1040, USA
§E-mail: p.mueller@web.de
¶E-mail: poetzsch@informatik.uni-kl.de
‖E-mail: leavens@cs.iastate.edu
Contract/grant sponsor: US National Science Foundation; contract/grant number: CCR-9803843, CCR-0097907, and CCR-0113181

```
public abstract class List1 {

    protected Node first, last;

    /*@ public normal_behavior
      @   modifies first; @*/
    public void initializeFirst() {
        first = null;
    }

    /* ... */
}
```

Figure 1. A JML specification of the Java class `List1`, of doubly-linked lists. (However, this specification violates JML's visibility restrictions, as explained in Section 1.1.1.) In JML, annotations for specifications are written in special comments. In this example, the method specification is written as a comment of the form `/*@` $\cdots$ `@*/` that precedes the method header. At-signs (`@`) on the beginnings of lines within annotations are ignored.

to avoid forcing users to learn the Larch Shared Language, JML hides its mathematical models behind a facade of Java classes. This allows JML to mostly use Java's expression syntax for writing assertions, which makes it more practical than Larch-style languages, and more like Eiffel [24]. As with Eiffel, one can use JML specifications to do runtime assertion checking.

For example, consider the specification given in Figure 1. The modifies clause in the specification of the `initializeFirst` method says that the object's field `first` can be modified by execution of the method. That is, the values stored in this location may differ between the pre-state (at the beginning of the call) and the post-state. However, it is not required that the value actually changes; the only obligation is that other locations, such as the field `last` and the fields of other objects, cannot be modified by a call to `initializeFirst`.

This naive semantics of modifies clauses is worth emphasis, as it is the crux of the problems we address.

**Definition 1.1 (Naive semantics of modifies clauses)** *When a method is invoked in a state in which its precondition is satisfied, its execution may only change the locations listed in its modifies clause; no other locations may be modified by such an execution.*

The property that "nothing else changes" is useful in reasoning about calls to methods [2]. For example, the modifies clause of `initializeFirst` says that calls to that method leave the values of locations other than the `first` field of the receiver object unchanged. For example, let `myList` be an object of type `List1`, `another` a list object distinct from `myList`. In this case, if the first of the following two assertions holds, then after the method call, we expect that the second holds as well.

```
//@ assert myList.last != null && another.first != null ;
myList.initializeFirst();
//@ assert myList.last != null && another.first != null;
```

Of course for this kind of reasoning to be valid, one must verify that method implementations meet their specified modifies clauses (when their preconditions hold) in the sense that they leave unchanged all locations not mentioned in their modifies clauses.

### 1.1. Three Specification Problems and their Solutions

The naive semantics for modifies clauses leads to three problems in the specification of frame properties, which we discuss in this section. The solutions to these problems lead to the modularity problem that is the focus of the rest of this paper.

#### *1.1.1. The Information Hiding Problem and Abstract Locations as a Solution*

The first problem is an information hiding problem. The problem is that the representation (concrete fields) of a class should be hidden from its clients, even in specifications of frame properties. If this is not done, then when the representation changes (e.g., when fields are renamed), the specification would also have to be changed. In JML, client-visible specifications are given the visibility modifier `public`, hence in JML information hiding means that public specifications should only mention public attributes of objects [24]. So in JML terms, the information hiding problem is that public specifications cannot mention protected and private fields in modifies clauses. For example, in Figure 1, the public specification of `initializeFirst` mentions the field `first`, which is protected. Since this protected field should not be visible to clients, the specification does not satisfy JML's visibility rules.

Leino's work [18] solves this information hiding problem by introducing abstract fields. These abstract fields are similar to the standard notion of specification using abstract values for objects [11], but following Leino (and other specification languages) JML provides several abstract fields rather than a single monolithic abstract value for an object. This matches the Java programmer's intuition that an object's state is composed of several fields. So in the semantics used for specification and verification, abstract fields are considered to denote locations, just like concrete fields, although the locations of these abstract fields are not present in actual programs. For example, in the legal JML specification given in Figure 2, we declare `listValue` as an abstract field, by using the modifier `model`. The abstract field `listValue` is declared to be public, and so can be used in public specifications. This use of abstract fields solves the information hiding problem, since the concrete fields used in an implementation can be changed (e.g., they can be renamed), without changing the specification visible to clients.

The correspondence between the abstract field `listValue` and the concrete field `first` is given by the represents clause in Figure 2. Such a *represents clause* says how an abstract location's value is determined from other abstract and concrete locations. It thus plays the role of an abstraction function [11, 18]. For example, the represents clause in Figure 2 says how `listValue` is determined from the concrete location `first` and the abstract location `first.values`. (The abstract field `values` is declared in the type `Node`; see Figure 7.) The

```
//@ model import org.jmlspecs.models.*;
public abstract class List2 {

    protected Node first, last;
    //@ public model JMLObjectSequence listValue;
    /*@ protected represents listValue <-
      @    (first == null ? new JMLObjectSequence() : first.values);  @*/

    /*@ public normal_behavior
      @   modifies listValue;
      @   ensures listValue != null && listValue.isEmpty(); @*/
    public void initializeFirst() {
        first = null;
    }

    /* ... */
}
```

Figure 2. A JML specification of the Java class `List2`. In JML, annotations can also be written as comments on lines beginning with `//@`. The specification of `initializeFirst` is not technically satisfiable as it stands, as explained in the text.

represents clause in Figure 2 is given protected visibility because it involves protected fields. It is thus not considered visible to clients. (The type `JMLObjectSequence` is defined in the package listed in the model import declaration at the top of Figure 2. `JMLObjectSequence` is an example of a class that hides a mathematical model as mentioned above. Objects of this type are immutable finite sequences of objects.)

### *1.1.2. The Modification of Concrete Locations Problem and Dependencies as a Solution*

The problem with abstract locations is that they conflict with the naive semantics of the modifies clause. For example, in the class `List2` (shown in Figure 2), how can the code in the `initializeFirst` method modify the concrete field `first` when the method's modifies clause only allows the abstract field `listValue` to be modified? According to the naive semantics of the modifies clause, it cannot, and so the code is incorrect according to this semantics. It turns out that this is not merely a technicality, as it is unsound to simply allow modification of locations that are not visible to clients [17, 18, 19].

Leino introduced a second kind of declaration to solve the modification of concrete locations problem. These declarations are depends clauses [18, 20]. To a first approximation, a *depends clause* says what locations are used to determine an abstract location's value. More precisely,

a dependency declaration allows dependees to be modified whenever the dependent abstract location is allowed to be modified.

In JML, the dependency declarations are written in annotations as follows.

`depends` *absfield* `<-` *dependeefield*

(See Appendix A for more about JML's syntax.) This declares a dependency relationship between fields, namely that *absfield* depends on *dependeefield*, a relationship that holds at runtime between the corresponding locations. Dependency relationships are transitive, if *dependeefield* is declared to depend on *dependeefield2*, then *absfield* also depends on *dependeefield2*, and the set of dependees of *absfield* includes both *dependeefield* and *dependeefield2*.

We call the locations that an abstract location $L$ depends on "the dependees of $L$". With this terminology, we can state a more sophisticated semantics of the modifies clause.

**Definition 1.2 (Semantics of modifies clauses with dependencies)** *When a method is invoked in a state in which its precondition is satisfied, its execution may only change the locations listed in its modifies clause and their dependees; no other locations may be modified by such an execution.*

Using this semantics, the code for `initializeFirst` in `List2` will only be considered to be correct if one adds the following dependency declaration to Figure 2.

```
//@ protected depends listValue <- first;
```

With this declaration, the code for `initializeFirst` acquires the right to modify `first`, since it is a dependee of `listValue`.

Leino and Nelson [20] distinguish *static dependencies*, such as "`depends` $f$ `<-` $g$", from *dynamic dependencies*, of the form "`depends` $f$ `<-` $p.g$", in which abstract field $f$ depends on field $g$ of the *pivot field* $p$. All the dependency declarations shown so far have been static dependencies. However, in Figure 2, the represents clause for `List2`'s abstract field `listValue` uses `first.values`; consequently, a change of `first.values` might cause a modification of `listValue`. Hence we must add the following dynamic dependency declaration to Figure 2.

```
//@ protected depends listValue <- first.values;
```

In the above declaration, `first` is the pivot field.

Leino and Nelson handle static and dynamic dependencies in different ways, that is, they are treated differently in the semantics of modifies clauses and have to obey different modularity rules. Although Müller's thesis [26] treats both cases uniformly, for the sake of simplicity, in this paper we also distinguish them.

Dependency declarations can be made even if the abstract location's value is not determined by the concrete location's value. For example, a field that is a cache would be a dependee that would not be needed to determine an object's abstract value. This often occurs in subtypes, which is the subject of the next problem.

### *1.1.3. The Extended State Problem and Dependencies as a Solution*

The third specification problem arises from subtyping in object-oriented languages. In such languages, a subtype often extends the state of its supertypes, by adding additional fields. To deal with this extended state, the subtype's methods often need to modify these additional fields, yet the naive semantics of the modifies clause and the demands of behavioral subtyping (e.g., [22, 8]) would seem to prohibit their modification [19]. This is because the methods of a behavioral subtype must obey the specifications of any methods they override from their supertypes.† For example, if one specifies `LengthCachingList` as a subtype of `List2` as in Figure 3, then its `initializeFirst` method needs to modify `absLength` to preserve the invariant. However, the modifies clause of the supertype's method specification, which must be obeyed in a behavioral subtype, would prohibit `absLength` from being modified, since it is not one of the locations named by that specification.

Dependencies also solve this extended state problem: i.e, the problem of how to write specifications that allow modification of extended state in subtypes. For example, to solve the extended state problem for `LengthCachingList`, one would add the following declaration to Figure 3.

```
//@ protected depends listValue <- absLength;
```

This allows `absLength` to be modified whenever `listValue` is modifiable.

For correctness, one also needs to add the following depends declaration to Figure 3.

```
//@ protected depends absLength <- len;
```

This makes the concrete variable `len` a dependee of `listValue`, and as such, `len` can be modified by `initializeFirst`, according to the specification in `List2`.

### *1.1.4. Summary of the Specification Problems and their Solutions*

In sum, to solve these specification problems, JML follows Leino and Nelson [18, 20] by using abstract fields (declared using the modifier "`model`") and by explicitly declaring dependencies [16, 15].

Compared to the naive semantics of modifies clauses, the semantics of modifies clauses with dependencies requires some changes in reasoning [20]. For example, the modifies clause of `initializeFirst` now says that calls to that method leave the values of all locations other than the `listValue` field of the receiver object, and its dependees, unchanged. Similarly, for this kind of reasoning to be valid, one must verify that the implementation of this method only modifies `listValue` and its declared dependees. All other locations not mentioned must remain unchanged.

†Technically, the modifies clauses of overridden methods only have to be obeyed when the supertype's precondition holds [8].

```
public abstract class LengthCachingList extends List2 {

    protected int len = 0;
    //@ public model int absLength;
    //@ protected represents absLength <- len;

    //@ public invariant absLength == listValue.length();

    /*@ also
      @   public normal_behavior
      @     modifies absLength; @*/
    public void initializeFirst() {
        super.initializeFirst();
        len = 0;
        //@ assert absLength == 0;
    }

    /* ... */
}
```

Figure 3. A JML specification of the Java class `LengthCachingList`, of doubly-linked lists with abstract field `absLength`. The specification of `initializeFirst` is not technically satisfiable as it stands, as explained in the text.

### 1.2. The Modularity Problem

The problem we address in the rest of this paper is a modularity problem. This modularity problem arises when one adopts the solutions to the specification problems described above. For example, suppose we define an abstract datatype of sets, `Set2`, using `List2` as its representation, as shown in Figure 4. In the figure, `Set2` is a client of the abstraction `List2`; `Set2` has an abstract field `setValue` that is represented by an object `theList` of type `List2`. As `Set2`'s represents clause declares, the elements in a set's abstract field, `setValue`, are determined by the elements in `theList.listValue`. Hence, the depends clause of `Set2` says that `setValue` depends on both `theList` and `theList.listValue`.

The modularity problem is that the call to `initializeFirst` in `Set2`'s `emptyOut` method may change the value of the abstract location `setValue`. However, this location was not listed in the modifies clause of `initializeFirst` in Figure 2, and is not a dependee of the locations listed there. So the semantics of modifies clauses with dependencies presents us with two unsatisfactory choices.

```
//@ model import org.jmlspecs.models.*;
public abstract class Set2 {

    protected /*@ non_null @*/ List2 theList;
    //@ public model non_null JMLObjectSet setValue;
    /*@ protected represents setValue \such_that
      @   (\forall Object o; o != null;
      @       theList.listValue.has(o) <==> setValue.has(o));  @*/
    //@ protected depends setValue <- theList, theList.listValue;

    /*@ public normal_behavior
      @   modifies setValue;
      @   ensures setValue.isEmpty();  @*/
    public void emptyOut() {
        theList.initializeFirst();
    }

    /* ... */
}
```

Figure 4. The JML specification of the Java class `Set2`.

Intuitively, the code for `emptyOut` is correct. When called, `emptyOut` sets `theList` to be the empty list, using the method `initializeFirst`, and, by the represents clause, this makes `setValue` empty. However, the semantics of modifies clauses with dependencies says that, because `setValue` is not mentioned in the modifies clause of `initializeFirst`, and because the locations mentioned there do not depend on `setValue`, `setValue` must remain unchanged by the execution of `initializeFirst`. So, for example, if `setValue` is not empty when `emptyOut` is called, then it stays non-empty, in contradiction to its postcondition.

```
//@ assert !setValue.isEmpty();
theList.initializeFirst();
//@ assert !setValue.isEmpty();
```

But since the code is correct, however, this indicates that something is wrong with the semantics of modifies clauses with dependencies, because it leads to invalid conclusions.

If we try to fix the problem by recognizing that the call to `initializeFirst` does indeed change the value of `setValue` (according to `Set2`'s represents clause), then we have another problem, because the modifies clause in `initializeFirst` is no longer valid. That is, the

`initializeFirst` has suddenly become incorrect, since its implementation no longer satisfies its modifies clause when used in a program containing `Set2`.

These problems indicate that the semantics of modifies clauses with dependencies is not modular; in a *modular* reasoning system, conclusions would be valid in every well-formed program in which a class or interface was reused. A modular solution to the frame problem must allow one to precisely specify the frame properties of methods and to verify their implementations, without knowing the program context in which the methods will be used. The problem is that, in general, one cannot know what locations might be found in a program that extends or uses a given class or interface. Looked at another way, the problem is that either we need a different semantics, or we need to restrict dependencies, or both [26].

Leino and Nelson have addressed this problem using scope-dependent depends relations [18, 20], which lead to a scope-dependent meaning of modifies clauses. That is, the meaning of a modifies clause is determined by the declarations that are visible in a module. Soundness of reasoning with such a semantics is not immediate, because proofs for smaller scopes do not necessarily carry over to larger scopes; indeed, Leino and Nelson have not yet proved modular soundness of their technique for dynamic dependencies.

This paper builds on Leino's work and provides a modular sound solution to this problem.

### 1.3. Approach

Our solution to the modularity problem entails four steps:

1. We define a programming model that hierarchically structures the object store (i.e., the program's memory) into "contexts" (see Section 2). The model also restricts references between contexts [26, 28, 29].
2. We loosen the semantics for modifies clauses using underspecification; instead of talking about all locations, only the modification of "relevant" locations (and their dependees) is specified by the modifies clause (as explained in Section 3). The notion of relevant locations is defined using the hierarchical programming model.
3. We also underspecify the theory generated by the dependency declarations in a given set of modules; this theory is underspecified in the sense that it does not specify dependencies for extensions to the given set of modules (sketched in Subsections 4.1.1 and 4.4). Because of this underspecification one can only prove properties about dependencies in a module that hold in all well-formed extensions.
4. We impose modularity rules to restrict the permissible dependencies of abstract locations (see Subsection 4.3).

Taking these steps makes modular soundness much simpler to prove than with a scope-dependent semantics of the modifies clause. The restricted programming model guarantees that this weaker semantics is still strong enough to verify method invocations, as we explain below.

A detailed presentation of a more general version of these ideas, including all formalizations and proofs, but not their application to JML, is found in [26]. The goal of this paper is to convey the main ideas behind this work and to apply them to Java and JML. For simplicity, we focus on a subset of sequential Java and omit inner classes, static fields, and static methods.

Our technique can be extended to these features, however. We also simplify Java's package system, and extending our techniques to deal with packages as found in Java is future work.

The rest of this paper is organized as follows. Section 2 explains the programming model. Section 3 informally presents our refined semantics of the modifies clause that solves the modularity problem. Section 4 formalizes these ideas, and presents the technical restrictions on dependencies and the modular soundness theorem. Section 5 presents further discussion, including a discussion of related work, and Section 6 presents conclusions.

## 2. The Programming Model

This section introduces a restricted programming model. The concepts of this model are used to define the notion of relevant locations needed for our refined semantics of the modifies clause and for controlling dependencies.

Modularity can only be achieved if the dependencies are controlled. There are two problems, both of which involve aliasing [30]: (1) *Representation exposure* occurs when objects inside the representation of an object $X$ may be referenced by objects outside of $X$'s representation. (2) *Argument exposure* occurs when an object $X$'s abstract value[‡] is determined by the values of locations in objects, called *argument objects*, that are outside $X$'s representation. Both problems allow modification of an object's abstract value in ways that cannot be controlled by its implementation. For example, if a client of `Set2` has a reference to the list that is used in the representation of an object of type `Set2`, then the client can change the abstract value of the set without calling any of `Set2`'s methods. Similarly, if the abstract value of a set object depends on the abstract values of its elements, then a client could modify the set's abstract value by modifying its elements, again without calling any methods of the set type.

In the next two subsections, we explain how representation and argument exposure can be avoided by structuring the object store into contexts and marking references as readonly. In the third subsection, we show how this additional structure and the underlying invariant can be enforced by our universe type system.

### 2.1. Preventing Representation Exposure: Contexts and Owner Objects

To prevent representation exposure, memory is structured into a hierarchy of contexts. *Object contexts* or simply *contexts* are disjoint groups of objects. There is a root context. All other contexts have a *parent* context and an *owner object* in their parent context. (See Figure 5.) Aliasing is controlled by the following invariant for this "ownership model" [4, 5, 30].

**Definition 2.1 (Ownership model invariant)** *Every reference chain from objects in the root context to an object in a context $C$ different from the root context passes through $C$'s owner. For purposes of this invariant, local variables and method parameters are treated like locations of the corresponding* `this`*-object.*

[‡]The abstract value of an object is a record that contains all the abstract locations specified for that object.

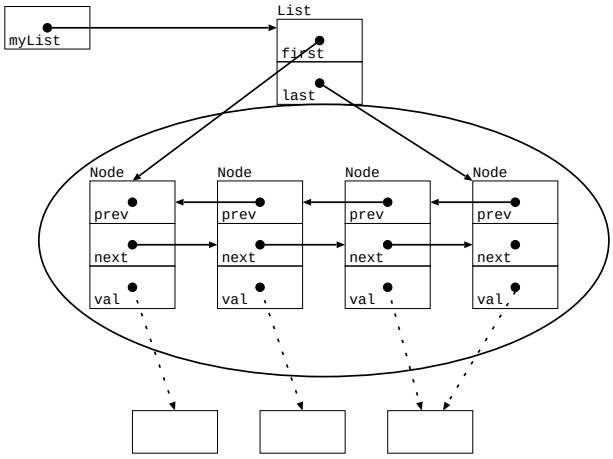


Figure 5. A parent context (outside the oval) and nodes within a context (the oval). The owner object sits atop the context it owns. Readonly references are indicated by dashed arrows.

The ownership model allows an owner object to control access to objects in its context, since this object acts as a gateway from the rest of the program to the objects in the context it owns. This directly prevents representation exposure. Ownership types [5] or universes [29] can be used to enforce the ownership model statically.

## 2.2. Preventing Argument Exposure: Readonly References and the Locality of Dependencies

The ownership model is not sufficient to prevent dependencies on argument objects: It allows objects inside a context to directly reference argument objects in parent or ancestor contexts. Thus, objects can transitively reference argument objects and locations can transitively depend on locations of argument objects in arbitrary contexts. On the other hand, if one uses contexts, then the ownership model would prevent a collection type, such as sets, from directly referencing objects that are not in ancestor contexts.

To solve these problems, we refine the ownership model [26, 29]. Fields may be declared as `readonly`. Locations that are instances of a readonly field are called *readonly locations*. They can hold references to any object independent of its context. References obtained from readonly locations are called *readonly references*. They can only be used for direct read access and invocation of operations without side-effects. In terms of the work of Boyland, Noble, and Retert [3, p. 18], readonly references are transitive in the sense that locations accessed via a readonly reference can only be read, not written, and any references read through them are also readonly.

In the *refined ownership model*, every reference leaving a context (except those in an owner object) has to be readonly. Thus references to argument objects are readonly. The refined model is more general than the original one in that it allows readonly references into contexts bypassing the owner. It is characterized by the following invariant:

**Definition 2.2 (Refined ownership model invariant)** *If object $X$ holds a reference to object $Y$, then either $X$ and $Y$ belong to the same context, or $X$ is the owner of $Y$'s context, or the reference is readonly. For purposes of this invariant, local variables and method parameters are treated like locations of the corresponding* `this`*-object.*

Figure 5 also illustrates the refined ownership model. The nodes of a linked list are contained in a context owned by the list header. The objects stored in the list are outside the context and are referenced readonly (dashed arrows).

Based on the refined ownership model, we can prevent dependencies on argument objects by forbidding dependencies via readonly references. We will explain this in Subsection 4.3. For the example in Figure 5 this means that abstract fields of the list must not depend on locations of the objects stored in the list.

Flexible alias protection [30] features an `arg` mode that is similar to the `readonly` mode of our refined ownership model. An arg mode reference transitively restricts access to fields of an object that are not modifiable. This has a similar effect as our model. The difference is that we focus on specification dependencies, whereas they focus on implementation dependencies.

### 2.3. The Universe Type System

The refined ownership model structures the objects into contexts and distinguishes between readwrite and readonly references. In Sections 3 and 4, we show how the additional structure and the properties expressed by the refined ownership model invariant are used to define the semantics of modifies clauses and to enable modular verification. Here, we explain the technique by which we establish and enforce the refined ownership model and its invariant. We use an extended typing discipline, the so-called *universe type system* [26, 29]. A program that is type correct with respect to the universe type system is guaranteed to maintain the refined ownership model invariant in all execution states. Thus, the invariant can be used as a background property for program verification.

The universe type system provides for each context a universe of types, i.e. an object of class T in one context is considered to be of a different type from an object of class T in another context. It is beyond the scope of this paper to explain this idea in detail. Instead, we focus on the notions that are needed in the rest of this paper. The universe type system distinguishes three kinds of reference (examples of which are given below):

- Ordinary references are readwrite references that do not cross context boundaries. They are declared and used like object references in Java.[§]

[§]If a Java program does not use contexts, then it has no context boundaries. Thus existing Java programs do not violate the restrictions of the universe type system.

- Rep-references are readwrite references to objects in the context owned by the current `this`-object. To distinguish them from context-local references, we prefix the ordinary type with the type modifier `rep` (see [5]), which stands for "representation". It indicates that the objects forming the representation of an owner $X$ are contained and encapsulated in the *descendant context* owned by $X$.
- Readonly references: As explained above, readonly references may cross context boundaries, but only enable read access. Readonly references are declared by using the type modifier `readonly`.

Fields, local variables, and parameters can be declared with an ordinary, a `rep`-type, or a `readonly`-type. Pivot fields whose pivot is a `rep`-type are called *rep-pivot fields*. As we will illustrate, objects in descendant contexts are created as objects of a `rep`-type.

For example, consider the specifications of `List` in Figure 6 and `Node` in Figure 7. The `rep` annotations in Figure 6 specify that pivot fields `first` and `last` refer to objects in a descendant context owned by the list object. These annotations are also added to the `new` expressions in the body of the `append` method in Figure 6, which are required by the type system to match against the declarations of the fields `first` and `last`. There are no `rep` annotations in Figure 7, so the nodes are all in the same context. The `readonly` annotations in the `append` method of Figure 6 and in `Node`'s constructor in Figure 7 are necessary to mark argument objects. The type system requires these to be specified to allow such objects to be assigned to `Node`'s `val` field, which is also annotated as `readonly`.

## 3. Informal Semantics of Modifies Clauses in JML

In this section we describe a modular semantics of modifies clauses using the concepts from the refined ownership model given in the previous section.¶

We first define the notion of a relevant location, which is the key concept in the semantics.

**Definition 3.1 (Relevant location)** *A location, $L$, is* relevant to the execution of a method $m$ with receiver object $X$ *iff $L$ is either in the context $C$ containing $X$ or in a descendant context of $C$.*

For example, if `myList` is an object of type `List`, then for the call `myList.append(o)` the relevant locations are those in the context that contains `myList`, and those in descendant contexts of `myList`. Since the field `first` in `List` is declared using the keyword `rep`, the object that `myList.first` points to is in the context owned by `myList` (see Figure 5), which is thus a descendant context of the context that contains `myList`. Since the `next` fields of `Node` objects are not declared using `rep`, the objects reachable via `next` are all in the same context. It follows that all the nodes are in the context owned by `myList`, and hence that the locations of these nodes are also relevant.

¶JML actually uses a stricter interpretation of the modifies clause than that presented here [16], but the differences are not important for this paper.

```
//@ model import org.jmlspecs.models.*;
public abstract class List {
    //@ public model non_null JMLObjectSequence listValue;
    protected /*% rep %*/ Node first, last;
    /*@ protected depends listValue <- last, last.next,
      @                              first, first.values;
      @ protected represents listValue <-
      @    (first == null ? new JMLObjectSequence() : first.values);  @*/

    /*@ public normal_behavior
      @   requires o != null;
      @   modifies listValue;
      @   ensures listValue.equals(\old(listValue.insertBack(o)));  @*/
    public void append(/*% readonly %*/ Object o) {
        if (last==null) {
            first = new /*% rep %*/ Node(null, null, o);
            last = first;
        } else {
            last.next = new /*% rep %*/ Node(null, last, o);
            last = last.next;
        }
    }

    /*@ public normal_behavior
      @   modifies listValue;
      @   ensures listValue != null && listValue.isEmpty(); @*/
    public void initializeFirst() {
        first = null;
    }

    /* ... */
}
```

Figure 6. A JML specification of the Java class `List`, of doubly-linked lists. Annotations of the form `/*%` $\cdots$ `%*/` enclose constructs that are new with this paper and not yet part of JML.

```
//@ model import org.jmlspecs.models.*;
public class Node {
    //@ public model non_null JMLObjectSequence values;
    public Node next, prev;
    public /*% readonly %*/ Object val;
    //@ public depends values <- next, next.values, prev, val;
    /*@ public represents values <-
      @   (next == null ? new JMLObjectSequence(val)
      @                 : next.values.insertFront(val));  @*/

    public Node(Node nextp, Node prevp, /*% readonly %*/ Object valp) {
        next = nextp;  prev = prevp;  val  = valp;
    }
}
```

Figure 7. The JML specification of the Java class `Node`. The public fields in this class are acceptable because its objects can be completely hidden within lists, due to the universe type system.

Using this concept, we can now give a modular semantics to modifies clauses.

**Definition 3.2 (Modular semantics of modifies clauses)** *When a method is invoked in a state in which its precondition is satisfied, among the relevant locations for the call its execution may only change the locations listed in its modifies clause and their dependees; no other relevant locations may be modified by such an execution.*

For example, the call `myList.append(o)`, may modify the locations `myList.first`, `myList.first.values`, `myList.last`, and all the fields of the nodes reachable from `myList.first` via the `next` field. This is because the modifies clause of `append` explicitly names `myList.listValue`, which depends on these other locations, all of which are relevant to the call.

Note that this definition says nothing about the locations that are not relevant to a call. This is the sense in which the semantics is underspecified, which is key to making the semantics modular.

To explore the modularity consequences of this semantics, consider an extended program, in which the type `List` is used to implement the type `Set`, specified in Figure 8. (This specification adds `rep` and `readonly` annotations to the specification `Set2` of Figure 4, as well as an additional method.) `Set`'s abstract field `setValue` depends on its concrete field `theList` and `theList.listValue`. Since the specification of `Set`'s `insert` method lists `setValue` in its modifies clause, a call such as `mySet.insert(o)` may modify `mySet.setValue` and all the locations on which it depends. Since `theList` is declared using `rep`, it is in the context owned by `mySet`, and so is in a descendant context of the one containing `mySet` (see Figure 9).

```
//@ model import org.jmlspecs.models.*;
public abstract class Set {
    //@ public model non_null JMLObjectSet setValue;
    protected /*% rep %*/ /*@ non_null @*/ List theList;
    //@ protected depends setValue <- theList, theList.listValue;
    /*@ protected represents setValue \such_that
      @   (\forall Object o; o != null;
      @       theList.listValue.has(o) <==> setValue.has(o));  @*/

    /*@ public normal_behavior
      @   requires o != null;
      @   modifies setValue;
      @   ensures setValue.has(o);  @*/
    public void insert(/*% readonly %*/ Object o) {
        if (!theList.contains(o)) { theList.append(o); }
    }
}
```

Figure 8. The JML specification of the Java class `Set`.

Therefore `mySet.theList` is a relevant location, and since it is also a dependee, it can be modified. Similarly, `mySet.theList.listValue`, `mySet.theList.first`, and the fields of the nodes are relevant, and so these dependees can be modified.

The modularity of the semantics is shown by the call `theList.append(o)` in `Set`'s `insert` method. How does the semantics allow `List`'s `append` method to modify the set's abstract field `setValue`, which it does when it modifies the abstract value of `theList`? The semantics allows this because a modifies clause only describes the modification of relevant locations, and `setValue` is not relevant for the call `theList.append(o)`. The reason for this can be seen in Figure 9. In that figure, the locations relevant to the call are all inside the context that owns `theList`, which is the outer oval in the figure. The set object itself, i.e., the object pointed to by `mySet` is not in this context, and hence locations in that object are not relevant for the call. Thus, although the figure does not show the abstract field `setValue`, since that field is thought of as part of this set object, it is also not relevant for the call in question.

Responsibility for verifying frame properties is divided. A method's implementor is responsible for the locations specified in its modifies clause that are relevant to its executions, and the method's caller is responsible for other locations. For example, `append`'s implementor is responsible for verifying that, of the relevant locations, only `listValue` and its dependees are modified. When verifying the call to `append` in `Set`'s `insert` method, one uses `append`'s

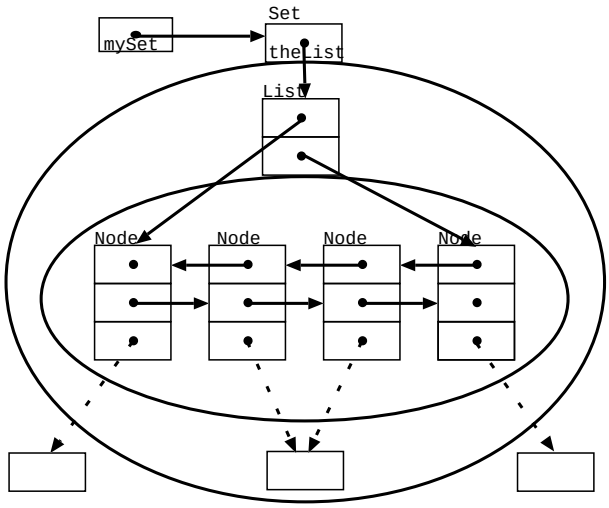


Figure 9. Three contexts, the outer of which contains a `Set` object that owns a context containing a `List` object.

modifies clause and `Set`'s depends clauses to reason about modification of `Set`'s fields `theList` and `setValue`.

## 4. Modularity and Dependencies

In this section, we argue that our specification approach enables modular sound verification of frame properties. Modular verification means that a method implementation $m$ can be proved correct with respect to its specification based on the knowledge available in the module of $m$. In particular, verification can be done without knowing all program contexts in which $m$ will be used. Modular soundness means that the proof of $m$ remains valid in all well-formed program contexts extending the module of $m$.

To understand the scenarios that threaten modular soundness, let us consider a module $M$ containing the implementation of a method $m$ and a module $M_{ext}$ that imports $M$ and uses $m$. We assume that the modifies clause allows $m$ only to modify the field $g$ of the receiver object and that $m$ is proved correct based on the knowledge available in $M$. Essentially, there are three *problematic scenarios* that must be avoided to achieve modular soundness.

1. *New fields*: Let $M_{ext}$ contain the declaration of an abstract field $f$ and dependency declarations such that a location $X.f$ might depend on a location $Y.g$. Thus, modifying $Y.g$ can change $X.f$. Assuming that $X.f$ can be relevant for a call to $m$, it has to be contained in the modifies clause of $m$. However, $f$ is not available in $M$, so the modifies

clause cannot refer to it. In particular, the proof of $m$ might no longer be valid in $M_{ext}$. (See Subsection 1.2 for an example for this scenario.)

2. *New dependencies*: Let's assume that a location $X.h$ does not depend on a location $Y.g$ according the dependency declarations of $M$ or modules imported by $M$. However, in $M_{ext}$ dependencies might be added such that $X.h$ becomes dependent on $Y.g$ in $M_{ext}$. Thus, a modification of $Y.g$ could modify $X.h$ which is not allowed according to $m$'s modifies clause.
3. *Hidden dependencies*: Let $f$ and $g$ be visible in $M$, but the dependency between them not be visible to the developer of $m$. Then, the developer might wrongly conclude that location $X.f$ does not depend on $Y.g$.

In the following, we explain how our specification and verification framework for frame properties addresses these problems. The basic ideas are:

- An underspecified *depends relation* ensures that one can only prove properties about a set of modules' dependencies that hold in all well-formed extensions (defined below) of this set of modules.
- *Modularity rules* restrict the declaration of fields and dependencies in modules that import other modules.

A module that is type correct w.r.t. the universe type system and that satisfies the modularity rules is called *well-formed*.

The rest of this section is structured as follows. Subsection 4.1 explains the module system and provides the basic definitions for the depends relation. Subsection 4.2 formulates *modularity properties* that ensure modular soundness. In modules that have the modularity properties the problematic scenarios sketched above cannot occur. The remaining subsections develop the modularity rules, illustrate them by an example, and show that modules that are well-formed w.r.t. these rules have the modularity properties, that is, enable modular verification.

### 4.1. Module System and Depends Relation

To make the following presentation more precise, we introduce a simple module system and define what it means for one location to depend on another.

#### *4.1.1. A Simple Module System*

To talk about modular soundness and verification, one needs a module system. The module system has to define for each class $C$ the precise program context that is used for the verification of $C$. Unfortunately, the package mechanism of Java provides a bad basis for this purpose. Java packages are essentially a naming mechanism. They are not (necessarily) sealed — a package can be extended by new types in new compilation units at any time. Thus, if a compilation unit, $CU$, imports classes from a package $P$, the set of classes available in the program at runtime from $P$ might change after verification of $CU$. Additional classes can in particular introduce more dependencies with problematic consequences for modular soundness (see below).

To avoid these soundness problems, and to have a sufficiently precise notion of program contexts, we introduce a simple module system that directly provides the needed notions. A *module* is just a uniquely named finite set of classes and interfaces. A module can import a list of other modules. Cyclic imports are not allowed. Modules also provide namespace control; that is, two classes or interfaces that have the same simple name, but which are declared in different modules are considered to be different. We leave as future work an adaptation of this module system that would deal with Java's existing package system.

Each class or interface belongs to exactly one module. Classes and interfaces declared in a module $M$ can refer to declarations in $M$ and all modules imported by $M$. More precisely, any declaration $D$ in a class or interface $C$ declared in module $M$ is *available* in $M$ and in all modules that directly or transitively import $M$. A declaration $D$ is *visible* in a class $C$, if and only if it is available in the module of $C$ and the rules for access modifiers of Java and JML allow its use in $C$.

In the following, we are mainly interested in the visibility of field and dependency declarations. For modular reasoning, it is important to know which declarations are available in a module, because only the declarations that are available can be used in proofs.

By the *range of a module*[‖] $M$, we refer to the set of all classes and interfaces that are contained in $M$ or that are directly or indirectly imported by $M$. We say that a module $N$ is an *extension* of $M$, if it directly or indirectly imports $M$. If $N$ extends $M$, the range of $N$ is a superset of the range of $M$.

For simplicity, we say that a *location* $X.f$ is *declared/available/visible* in a class $C$, a module $M$, or a range $R$, iff its field named $f$ is declared/available/visible in $C$, $M$, or $R$, respectively.

### *4.1.2. The Depends Relation*

The dependency declarations on fields induce a dependency relation on locations. Static dependency declarations of the form "`depends` $f$ `<-` $g$", where $f$ and $g$ are fields of a type $T$ lead to dependencies where locations of the form $X.f$ depend on locations $X.g$, where $X$ is an object of type $T$. Dynamic dependency declarations of the form "`depends` $f$ `<-` $p.g$" lead to dependencies between locations of different objects: If $X.p = Y$ in an execution state $S$, then location $X.f$ depends on $Y.g$ in $S$. That is, whereas dependency of fields is a static property, dependency of locations is in general a dynamic property. The reflexive, transitive closure of this relation on locations is called the *depends relation*. We use the dependency declarations in modules to axiomatize the depends relation (see Subsection 4.4).

For a given module $M$, the depends relation is underspecified in two respects: (1) It is only specified for the locations available in $M$, and not for locations corresponding to fields that are declared in modules extending $M$. (2) It is only specified w.r.t. the dependency declarations available in $M$, and not for dependency declarations contained in modules extending $M$. The first point is a consequence of modular programming and we have to cope with it. The problems

[‖] We use this term instead of saying "the scope of $M$" as it is e.g. done in [20], because in the programming language literature "scope" often refers to the part of a program text where an identifier is visible.

resulting from the second point will be dealt with by restricting the dependencies that might be added in modules importing $M$ (see Subsection 4.3).

### 4.2. Modularity Properties

To prove that the implementation of a method $m$ in module $M$ satisfies its modifies clause, we need to know about the relevant locations that $m$ might modify. For modular reasoning, this knowledge has to be available in $M$ and may not change in modules extending $M$. The availability of sufficient knowledge in $M$ is necessary to prove $m$'s modifies clause is satisfied in $M$. The claim for stable knowledge is necessary for modular soundness.

The proof of $m$ done in $M$ guarantees that relevant locations available in $M$ are: either (i) covered by $m$'s modifies clause according to the dependency declarations available in $M$ or (ii) remain unchanged under execution of $m$. We say that a method $m$'s modifies clause, $mc$, *covers a location* $X.g$ iff the field $g$ is listed in $mc$ for an object that denotes $X$ in $m$'s pre-state, or is a dependee of such a location in $M$. To show that the locations remain unchanged one has to prove that they do not depend on concrete locations modified in $m$ and do not depend on locations listed in modifies clauses of methods that are called in $m$.

As demonstrated by the problematic scenarios above, in a naive approach it is not possible to use the proof of $m$ done in $M$ to infer that all relevant locations in any extension of $M$ are covered by $m$'s modifies clauses or remain unchanged. However, if the verification framework satisfies the following three *modularity properties*, then such a proof generalizes to all extensions of $M$.

**Property 4.1 (Field visibility)** *Whenever an execution of a method $m$ modifies an abstract location $X.f$, at least one of the following three cases applies:*
*1. $X.f$ is not relevant for the execution of $m$,*
*2. The declaration of $f$ is visible in the module where $m$ is declared, or*
*3. $m$ modifies $X.f$ by invocation of a method $n$, and $X.f$ is covered by $n$'s modifies clause.*

Property 4.1 guarantees that fields (and the corresponding locations) are visible whenever they are needed to prove frame properties. Note that, in case 3, we are relying behavioral subtyping to ensure that methods that override $n$ also satisfy $n$'s modifies clause.

**Property 4.2 (Local completeness)** *The depends relation of module $M$ is* locally complete *in $M$. That is, for all locations $X.f$ and $Y.g$ such that the fields $f$ and $g$ are available in $M$ and in all states $S$, it is specified whether $X.f$ depends on $Y.g$ in $S$ or not.*

Property 4.2 is needed to show that two locations do *not* depend on each other and especially to show that a method does *not* modify a given location. Further explanation of this property will be given in Subsection 4.4.

**Property 4.3 (Consistency)** *The depends relation of module $M$ is* consistent *with the depends relation in all extensions of $M$. That is, in every state, $X.f$ depends on $Y.g$ in $M$ if and only if $X.f$ depends on $Y.g$ in all extensions of $M$.*

Property 4.3 guarantees that proofs based on the formalization of the depends relation in $M$ remain valid in extensions of $M$.

Properties 4.2 and 4.3 allow us to formalize that two locations available in $M$ are independent in $M$ and all extensions of $M$ iff there are no dependency declarations available in $M$ that make them dependent. They guarantee that knowledge available in $M$ remains valid in extensions of $M$. And they provide sufficient knowledge to show that locations not covered by the modifies clause of a method $m$ do not depend on locations modified by $m$. This is a critical part of proving method implementations correct. Especially, verification frameworks that have the above properties are modular sound.

**Theorem 4.4 (Modular soundness)** *A verification framework that satisfies Properties 4.1 and 4.3 is modular sound. That is, if there is a proof done in module $M$ that method $m$ satisfies its modifies clause, then $m$ also satisfies its modifies clause in every well-formed module $M_{ext}$ that extends $M$.*

According to the modular semantics of modifies clauses (see Definition 3.2), we have to show that whenever an execution of $m$ in the program context given by $M_{ext}$ modifies a location $X.f$, then either $X.f$ is not relevant or $X.f$ is covered by $m$'s modifies clause according to the dependency declarations available in module $M$.** According to Property 4.1, there are three cases.

**Case 1:** *$X.f$ is not relevant.* Thus according to the modular semantics of modifies clauses, nothing has to be shown.

**Case 2:** *$X.f$ is visible in $M$.* Thus, since $m$ satisfies its modifies clause in $M$, $X.f$ is covered by $m$'s modifies clause according to the dependency declarations available in $M$. Since the dependency declarations available in $M$ are also available in $M_{ext}$, Property 4.3 guarantees that $X.f$ is also covered by $m$'s modifies clauses according to the dependency declarations available in $M_{ext}$.

**Case 3:** *$m$ modifies $X.f$ by invocation of a method $n$, and $X.f$ is covered by $n$'s modifies clause.* In this case, $n$ and the locations listed in its modifies clause are visible in $M$ and are, according to the proof for $m$ done in $M$, covered by $m$'s modifies clause. Consequently, $X.f$ is covered by $m$'s modifies clauses according to the dependency declarations available in $M$. Again, Property 4.3 allows us to derive that $X.f$ is also covered by $m$'s modifies according to the dependency declarations available in $M_{ext}$. (Note that in this case, $X.f$ does not need to be available in $M$.) □

The above theorem shows that the modularity properties are sufficient for modular verification and guarantee modular soundness. In particular, these properties rule out the problematic scenarios sketched at the beginning of this section as explained in the following.

In the first scenario (*New Fields*), we know that the location $X.f$ is relevant to the execution of $m$ and that $X.f$ is not visible in $M$. Thus, Cases 1 and 2 of Property 4.1 do not apply and we can conclude that Case 3 applies: $X.f$ is modified by an invocation of a method $n$ and is covered by $n$'s modifies clause. This case is identical to Case 3 of the proof of the soundness theorem.

**This informal argument simplifies matters in two respects. First, we do not consider the properties given in preconditions. Second, we use the term "a modifies clause covers a location" which is a bit sloppy; to be precise, coverage has to be taken w.r.t. the method's pre-state.

The second scenario (*New Dependencies*) cannot happen, because it contradicts Property 4.3: If $X.h$ does not depend on $Y.g$ according to the depends clauses in $M$, then it does not depend on $Y.g$ in extensions of $M$. That is, adding dependencies in the way assumed in the second scenario is not possible in verification and specification frameworks that satisfy the modularity properties.

The third scenario (*Hidden Dependencies*) cannot happen, because we use a formal approach in which it is only allowed to infer properties from the formalization of the depends relation, but not by informal meta reasoning. Nevertheless, the third scenario illustrates an interesting aspect: What happens if there are dependency declarations available in $M$ that are not visible? In that case, certain axioms are also hidden and cannot be used in proofs with the consequence that one might neither be able to prove that $X.f$ depends on $Y.g$ nor that it does not. Thus, some proofs cannot be accomplished, which means that the proof techinque is incomplete.

So far, we have shown that the above modularity properties ensure modular soundness. The following subsections explain the concepts and restrictions we use in our verification framework to establish the modularity properties.

### 4.3. Authenticity

Authenticity is concerned with how to achieve Property 4.1 for relevant locations that are modified through field updates. Recall that property 4.1 ensures that the verifier of a method $m$ can prove that all relevant locations that $m$ might modify are covered by $m$'s modifies clause.

This property is trivial for locations that are not relevant for $m$ (Case 1 of Property 4.1). For locations modified by method invocations (Case 3), we assume the following property for all method invocations:

**Property 4.5 (Caller coverage)** *All locations that are covered by the modifies clause of the invoked method are also covered by the modifies clause of the invoking method.*

Although this property is not technically necessary [26], it simplifies the treatment of method invocations significantly. We assume this property to hold since, in general, it has to be proved for every method invocation during verification of a method body anyway.

This leaves case 2 of Property 4.1; that is how to reason about relevant locations that are modified through field updates. Assume that a method $m$ declared in module $M$ updates a location $Y.g$. To ensure Property 4.1, we must guarantee that all abstract locations that are relevant for the execution of $m$ and that might be affected by the update of $Y.g$ (that is, depend on $Y.g$) are visible in $M$. This requirement is called *authenticity* in Leino's and Müller's work [20, 26].

In a naive approach, we could achieve authenticity by requiring that all dependents of $Y.g$ are visible wherever $g$ is. Such a requirement works for static dependencies and certain visibility modifiers, but rules out important implementation patterns such as the `List2-Set2` example in Subsection 1.2, where `setValue` can certainly not be visible in the module that declares `listValue` (see Figures 2 and 4).

This problem is analogous to the problem that lead to the modular semantics of modifies clauses (see Definition 3.2): Abstract locations of classes such as `Set2` are not available in classes of imported modules that are used as underlying implementation (such as `List2`). To solve this problem, we enforce (1) that such abstract locations are not relevant for executions of methods of the imported classes and (2) that the *callers* of methods of imported classes are able to determine the effects of the method executions on such locations (rather than the executed method that performs the actual update). This approach allows us to achieve authenticity and still enable verification of implementations such as the `List2`-`Set2` example. Thus we aim to establish Property 4.1 by the following four requirements:

1. *Locality*: We prevent dependencies on locations of argument objects (see Subsection 2.2) to restrict the contexts to which dependees of a location can belong.
2. *Ownership*: We exploit the fact that the refined ownership model (see Definition 2.2) permits access to contexts only through owner objects.
3. *Visibility*: We impose a restriction on depends clauses that guarantees that abstract locations are available in all methods that could modify them and for which they are relevant.
4. *Accessibility*: We require abstract fields to be public, so that abstract fields are visible wherever they are available.

In the following we explain how these requirements can be enforced by statically checkable rules and how they allow one to determine the effects of a field update on relevant locations.

### *4.3.1. Locality*

As explained in Subsection 2.2, locality of dependencies means that locations do not depend on argument objects. We can achieve locality by forbidding dependencies via readonly references:

**Definition 4.6 (Locality rule)** *Pivot fields must not be readonly.*

This rule and the invariant of the refined ownership model guarantee that a location in a context $C$ only depends on locations in $C$ or $C$'s descendants: According to the forms of depends clauses we allow in this paper, $X.f$ can only depend on $Y.g$ if $X$ and $Y$ are the same object or if there is a reference chain from $X$ to $Y$. $X$ and $Y$ are the same object when there is a static dependency; dynamic dependencies involve reference chains. Such a reference chain can only leave the context that owns $X$ if it contains a readonly reference.

### *4.3.2. Ownership*

According to the invariant of the refined ownership model (see Definition 2.2), each context except the root context has an owner object that controls access to the objects in the context it owns. In analogy to owner objects, the universe type system introduces the notion of an owner type.

The *owner type* of a field is a class that is visible in every module that contains code that directly accesses the field. In most programs, such as in our examples, the owner type of a

field is its declaration type. In other cases, it is the most general super class of the declaration type that contains a field of a `rep` type. The universe type system guarantees the following properties (see [26] for a precise definition of owner types).

**Property 4.7 (Owner type property)**
*1. There is an owner type for each field.*
*2. All fields of `rep` types that could hold references to objects in the same context have the same owner type.*

For instance, in class `Set` (see Figure 8), the owner type of `theList` is `Set`, which is also the type of the owner object of the context `theList`-references point to.

#### *4.3.3. Accessibility*

To achieve that abstract fields are visible wherever they are available, we require that abstract fields be public.

**Definition 4.8 (Access rule)** *Abstract fields have to be public.*

The access rule is a simple way to guarantee the following transitivity property for abstract fields $f$ and $g$, and a program point $P$: If $f$ is visible at the declaration of $g$, and $g$ is visible at $P$, then $f$ is visible at $P$. This would for instance not be the case, if $f$ was private, $g$ was public, $f$ and $g$ were declared in the same class $C$, and $P$ was outside $C$.

More sophisticated rules that allow the hiding of abstract fields can be used to achieve transitivity. However, the given rule suffices for our purposes without violating information hiding: The declaration of an abstract field only reveals its name, not its representation.

#### *4.3.4. Visibility*

According to the refined ownership model, a method $m$ executed in context $C$ can update concrete locations of objects in $C$ and immediate descendants of $C$.

If $m$ updates a location $Y.g$ in $C$, locality guarantees that all relevant locations that might be affected by these updates are also in $C$. Thus, we can achieve authenticity by requiring that for each dependency where the dependent and the dependee belong to the same context, the dependent must be visible wherever the dependee is. In this case, the dependees have to be visible where $Y.g$ is, especially in $m$.

If $m$ updates a location $Y.g$ in an immediate descendant of context $C$, we need a weaker requirement to enable implementation patterns such as the `List2-Set2` example. In this case, we can exploit the fact that the owner type of a field is visible in every module that contains code that directly accesses the field. Thus, it suffices to require that all dependents of $Y.g$ in $C$ are visible in $g$'s owner type and, thus, in $m$.

These two requirements can be statically checked using the following *visibility rule.*

**Definition 4.9 (Visibility Rule)** *Let $D$ be a dependency declaration of the form "`depends f <- g`" or "`depends f <- p.g`". If $D$ is static or if its pivot field is not of a `rep` type, then*

*D must be visible in the module where g is declared. If D is dynamic and its pivot field is of a* `rep` *type, then D must be visible in the module that contains the declaration of p's owner type.*

For authenticity, we do not need the visibility rule, but a consequence of it and the access rule:

**Lemma 4.10 (Authenticity)** *Let D be a dependency declaration of the form "*`depends` *f* `<-` *g" or "*`depends` *f* `<-` *p.g". If D is static or if its pivot field is not of a* `rep` *type, then the* declaration of *f must be visible in the module where g is declared. If D is dynamic and its pivot field is of a* `rep` *type, then the* declaration of *f must be visible in the module that contains the declaration of p's owner type.*

The lemma holds since (1) the field of the dependent in a depends clause is visible wherever the depends clause is and (2) the access rule guarantees transitivity of visibility for abstract fields.

In summary, the locality, access, and visibility rules act together to allow one to determine all relevant abstractions that might be affected by the modification of a concrete location $Y.g$ by the execution of a method $m$. The locality rule enforces that the abstractions that might depend on $Y.g$ belong to certain contexts. Most of these abstractions are not relevant for the execution of $m$. For the remaining abstractions, the access and visibility rules require that the involved depends clauses be visible in the module that contains $m$'s declaration. Therefore, the verifier can refer to the dependencies declared in the range of that module to reason about the modification. That is, the rules guarantee that authenticity and, thus, Property 4.1 holds in all well-formed programs. We will prove this in Subsection 4.7.

### 4.4. Locally Complete Depends Relations

Our approach to establishing Property 4.2 in our verification framework is to (almost) completely specify the depends relation for all locations available in a module $M$. We axiomatize both the depends relation and its negation as follows. (See [26] for a full formalization of the depends relation and its negation.)

As explained in Subsection 4.1, the depends relation for a module $M$ is specified by

- an appropriate axiom for each depends clause available in $M$ that relates the dependent and the dependee,
- a reflexivity axiom,
- a transitivity axiom.

According to Property 4.2, the axiomatization of the depends relation and its negation for a module $M$ should be complete for all fields available in $M$. However, such a complete axiomatization rules out a common implementation pattern that is illustrated in Figures 10 and 11.

Assume that `List`, `AbstractSet`, and `Set3` are declared in different modules. By following Property 4.2, `Set3` would not be allowed to introduce the depends clause `setValue <- theList, theList.listValue` for the following reason: There are modules in which `setValue`

```
//@ model import org.jmlspecs.models.*;
public abstract class AbstractSet {
    //@ public model non_null JMLObjectSet setValue;
}
```

Figure 10. The JML specification of the Java class `AbstractSet`. This abstract class declares only the abstract field `setValue`.

```
//@ model import org.jmlspecs.models.*;
public abstract class Set3 extends AbstractSet {
    protected /*% rep %*/ /*@ non_null @*/ List theList;
    //@ protected depends setValue <- theList, theList.listValue;
    /*@ protected represents setValue \such_that
      @   (\forall Object o; o != null;
      @       theList.listValue.has(o) <==> setValue.has(o));  @*/
}
```

Figure 11. The JML specification of the Java class `Set3`. Besides the extends clause and the missing declaration of `setValue`, `Set3` is an extract of class `Set`.

and `listValue` are available, but the class `Set3` is not (think of a module that imports the modules of `List` and `AbstractSet` and nothing else). If we would conclude from such modules that a `setValue`-location cannot depend on a `listValue`-location because no such dependency is declared, adding class `Set3` to the program would lead to an inconsistency since it introduces such a dependency.

To avoid this inconsistency and still enable the implementation pattern used in the example, we slightly underspecify the negation of the depends relation, that is, we use the following *weak local completeness property* in our verification framework:

**Property 4.11 (Weak local completeness)** *Let $X.f$ and $Y.g$ be locations with fields available in $M$ and let $S$ be any state. If no set of dependencies that would declare a dependency of $X.f$ on $Y.g$ contains a depends clause with a rep pivot, the owner of which is not available in $M$, then it is specified in $M$ whether $X.f$ depends on $Y.g$ in $S$ or not. Otherwise, $M$ does not need to specify whether $X.f$ depends on $Y.g$.*

Based on this weaker property, the negation of the depends relation for a module $M$ is axiomatized as follows: For each pair of locations $X.f$ and $Y.g$ available in $M$, an axiom is

generated that states that $X.f$ does not depend on $Y.g$ if the following two requirements are met.

1. The dependency cannot be derived from the depends clauses available in $M$, reflexivity, and transitivity.
2. Let $D$ be an arbitrary set of depends clauses that declare a dependency of $X.f$ on $Y.g$. For each depends clause in $D$ with a rep pivot $p$, the owner of $p$ is available in $M$.

The first requirement directly describes the negation of the depends relation. The second requirement slightly weakens the axiomatization, that is, it leaves parts of the negation of the depends relation underspecified: If $X$ and $Y$ belong to the same context, the second property always holds since no depends clause with a rep pivot can be involved. However, if $X$ belongs to an ancestor of the context to which $Y$ belongs, extensions of $M$ may introduce a dependency of $X.f$ on $Y.g$, thereby enabling the implementation pattern described above.

The weak local completeness property is still strong enough for modular verification: Because of our hierarchical programming model, verification is in general only concerned with dependencies between locations that belong to the same context or where the dependent belongs to the ancestor context of the dependee with the owner type of the dependee available. Both cases are not affected by the slightly underspecified depends relation.

In particular, the strong local completeness property is not necessary to achieve modular soundness. Therefore, the proof of Theorem 4.4 is also valid for verification frameworks with the weak local completeness property.

### 4.5. Consistent Depends Relations

Achieving consistency (Property 4.3) is more difficult than local completeness. The basic idea is to restrict the placement of dependency declarations in such a way that modules extending $M$ must not add further dependencies between fields available in $M$ (besides the exception discussed above). Note that it is not sufficient to simply forbid dependency declarations in extensions of $M$ that directly declare a dependency between fields available in $M$. Assuming that $f$ and $h$ are fields available in $M$ that are independent, the two dependency declarations "`depends` $f$ `<-` $g$" and "`depends` $g$ `<-` $h$" in a module extending $M$ could generate an inconsistency.

In [20], Leino and Nelson used the so-called the *visibility requirement* to avoid inconsistencies for static dependencies: A declaration "`depends` $f$ `<-` $g$" has to be visible in every range in which $f$ and $g$ are. For dynamic dependencies they use additional requirements.

The universe type system with its notion of owner types allows us to use a uniform rule for both static and dynamic dependencies. Our axiomatization of the depends relation and its negation together with the visibility rule (see Definition 4.9) guarantees that the axioms introduced for an extension $M_{ext}$ of a module $M$ are consistent with the axiomatization for $M$.

To illustrate how the visibility rule works, we argue that $M_{ext}$ cannot declare dependencies that contradict the axiomatization of the negation of the depends relation in $M$. A more comprehensive proof sketch of Property 4.3 can be found in Subsection 4.7.

Consider two locations $X.f$ and $Y.g$ available in $M$ that are independent. To introduce a dependency of $X.f$ on $Y.g$, $M_{ext}$ has to declare at least one depends clause where the dependee $Z.h$ is available in $M$ ($Z.h$ could be $Y.g$ or a dependent of $Y.g$). The visibility rule guarantees that this depends clause can neither be a static dependency nor a dynamic dependency where the pivot is not of a rep type, because in these cases the depends clause would have to be available in $M$. Therefore, the new depends clause has to be a dynamic dependency with rep pivot $p$ where the owner type of $p$ is available in $M_{ext}$, but not in $M$. Otherwise,the visibility rule would again ensure that the depends clause is available in $M$. However, if the dependency is established by such a depends clause, it does not contradict the axiomatization of the negation of the depends relation in $M$ since this case is explicitly excluded by the second requirement for the negation axioms (see above). That is, extensions lead to a refinement of the axiomatization of the depends relation and its negation.

As explained in detail in [26], this approach has been extended to reflect the different levels of visibility (public, protected, default, private). The more access rights a class has, the more parts of the axiomatization can be used within its proof. For instance, if only the public declarations of a class $A$ are visible in a class $B$, the proofs for $B$ may only use the public parts of the theory formalizing $A$. This way visibility is reflected in the formalization.

### 4.6. Example

Before we prove that the modularity rules presented above guarantee the modularity properties 4.1, 4.11, and 4.3, we illustrate how these rules work by revisiting the `List-Set`-example from Section 3. We assume that the classes `List` and `Node` are contained in a module List, which is imported by the module Set of class `Set`.

In a program that consists only of modules List and Set, `List`'s `append` method is correct since (1) all locations of `List` and `Node` objects modified by the method are covered by the modifies clause, and (2) all locations of `Set` objects modified by the method are not relevant for the method execution. These properties follow mainly from the depends relation and the types of `List`'s and `Set`'s fields. For instance, from the depends clause for `setValue`, the visibility rule, and the fact that `Set`'s `theList` field is of a rep-type, we can conclude that only the `setValue` location of the `Set` object that owns the `this` object can be modified by the execution of `append`. This object is not relevant for the method execution. Similarly, we can show that only the `listValue` of the `this` object is modified by `append` since the rep-types of `List`'s `first` and `last` fields indicate that `Node` objects are not shared between `List` objects. Thus, different `listValue` locations depend on disjoint `Node` locations. We assume that the typing of variables and fields is implicitly given in every pre- and postcondition and that we can refer to these properties in proofs (see [26] for a complete formalization).

However, the above argument does not hold if we import the module List by a module that provides the alternative implementation of sets given in Figure 12.

In contrast to `Set`, `SetAlt` in Figure 12 does not use the `rep` tag in the declaration of the field `theList`. Hence, `SetAlt` does not store the list in a descendant context (see Figure 13). Consequently, `List.append` might modify relevant `SetAlt` locations that are not covered by its modifies clause, because all these locations are in the same context.

```
//@ model import org.jmlspecs.models.*;
public abstract class SetAlt {
    //@ public model non_null JMLObjectSet setValue;
    protected /*@ non_null @*/ List theList;
    //@ protected depends setValue <- theList, theList.listValue;
    // ... rest identical to class Set
}
```

Figure 12. The JML specification of the Java class `SetAlt`.

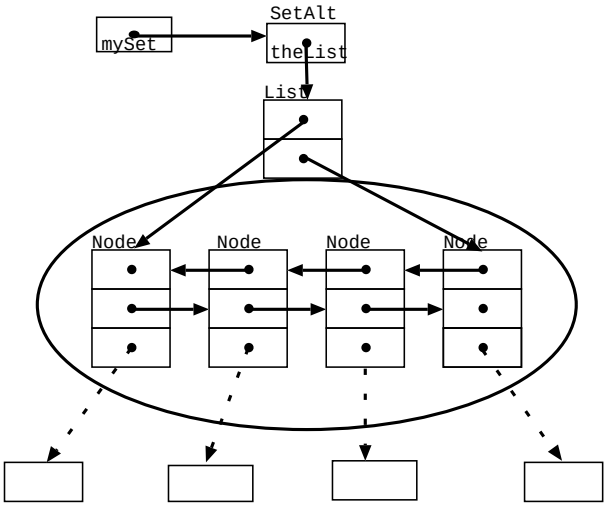


Figure 13. Object Structure for a `SetAlt` object.

In summary, the example illustrates the first problematic scenario from the beginning of this section, and we would expect our modularity rules to forbid the implementation of `SetAlt` since it invalidates the correctness of `List.append`.

Our first example, with the types `List` and `Set`, is well-formed. It is easy to check that the abstract fields and dependencies declared in Figures 6, 7, and 8 satisfy the modularity rules. In particular, the `setValue` location of a `Set` object is not relevant for executions of `append` on the corresponding `List` object.

1. *Locality rule:* Neither of the pivot fields, `first` of class `List`, `next` of class `Node`, and `theList` of class `Set`, are readonly. Finally, it should be noted that the dependency of `values` on `val` does not contradict the locality rule. The rule would only prohibit a dependence of `values` on a field of `val`.
2. *Access rule:* The abstract fields `values`, `listValue`, and `setValue` are declared public.
3. *Visibility rule:* Of course, the depends clause of class `Node` is visible in `Node`, where all fields mentioned in the depends clause are declared. The depends clause of class `List` is visible in the module List, which contains the declarations of `first` and `last` as well as their owner type, `List`. The depends clause of class `Set` is visible in the module Set, which contains the declarations of `theList` and its owner type, `Set`.

The situation is different for the second example with `List` and `SetAlt`. Since `theList` is not of a rep type, the depends clause

```
protected depends setValue <- theList, theList.listValue;
```

must be visible where the field `listValue` of the dependee is declared. That is, it must be visible in class `List` (rather than in `theList`'s owner type `SetAlt` as in the `List-Set` example), which is not the case. Therefore, the `List-SetAlt` example violates the visibility rule and thus is not well-formed. Consequently, the `setValue` locations of `SetAlt` objects are relevant for the corresponding executions of `append`, which leads to the unsoundness illustrated previously.

Ruling out the `List-SetAlt` example is also well justified from programming methodology: A set object that uses a list object to store its elements should make sure that other objects cannot manipulate the list in unexpected way and thereby, for instance, break the set's invariant. Consequently, the list object should not be exposed, that is, it should be owned by the set object. This is the case for `Set`, but not for `SetAlt`.

### 4.7. Modularity Theorem

In this subsection, we present the central modularity theorem that, together with Theorem 4.4, shows how the universe type system and the modularity rules enable modular sound verification:

**Theorem 4.12 (Modularity)** *The universe type system and the locality, access, and visibility rules guarantee the modularity properties, that is, field visibility (Property 4.1), weak local completeness (Property 4.11), and consistency (Property 4.3).*

We sketch the proof of the three modularity properties in turn. A formalization of the proof can be found in [26].

#### *4.7.1. Proof of Field Visibility*

This proof part runs by induction over the structure of the body of a method. That is, the induction base consists of the primitive statements, whereas all compound statements and method invocations are handled in the induction step. Here, we present the cases for field updates (the base case) and method invocations (the inductive case). These are the most

interesting cases since they directly deal with the modification of locations. All other cases are rather straightforward or necessary to handle the technical problems of recursion.

*Field Updates*

Let $m$ be executed in context $C$ (i.e., the receiver is in $C$). Suppose $m$ updates a concrete location $Y.g$ of some object $Y$. Then the universe type system guarantees that $Y$ is in $C$ or in one of $C$'s immediate descendants. Consider an abstract location $X.f$ that is relevant for $m$. If $X.f$ does not depend on $Y.g$, $X.f$ is not affected by updates of $Y.g$. Otherwise, we show that the declaration of $f$ is visible in the module where $m$ is declared:

**Case 1:** *$Y$ is in $C$.* Since $X.f$ is relevant for $m$, by the locality rule $X$ is in $C$. Thus, $X$ and $Y$ are in the same context, which implies that the dependencies between $X.f$ and $Y.g$ do not involve any rep-pivot fields. Thus, authenticity (Lemma 4.3) ensures that the declaration of $f$ is visible in the module where $g$ is declared. Since $g$ is visible in $m$, the declaration of $f$ is visible in the module where $m$ is declared.

**Case 2:** *$Y$ is in an immediately-descendant context $D$ of $C$.* Due to locality, $X$ is in $D$ or in $C$. The former case is analogous to Case 1, since both $Y$ and $X$ are in the same context. In the latter case a dynamic dependency must be involved with a pivot field $p$ of a `rep` type. In this case the owner type of $p$ is visible in the module where $m$ is declared (by the universe type system), and the declaration of $f$ is visible in the module of $p$'s owner type (by authenticity, Lemma 4.3). Thus, the declaration of $f$ is visible in the module where $m$ is declared.

*Method Invocations*

Let $m$ be executed with a receiver object $Z$ belonging to context $C$ and let $D$ be the context owned by $Z$; that is, $D$ is an immediate descendant of $C$. If $m$ invokes a method $Y.n$, we can assume inductively that the modularity theorem holds for $Y.n$ (for mutually recursive methods, one has to use a more refined proof technique, see [26]). The universe type system guarantees that either $Y.n$ has no side effects, or that $Y$ is in $C$ or in $D$. Let $X.f$ be an abstract location modified by $Y.n$. If $X.f$ is not relevant for the call $Z.m$ or covered by $n$'s modifies clause, the modularity property for $Z.m$ follows directly since Case 1 or Case 3 of Property 4.1 applies trivially. Otherwise, $X.f$ is relevant for $Z.m$ and not covered by $n$'s modifies clause. We show by case distinction that $f$ is visible in the module where $m$ is declared, that is, that Case 2 of Property 4.1 applies:

**Case 1:** *$Y$ is in $C$.* Because $Y$ and $Z$ belong to the same context and because $X.f$ is relevant for $Z.m$, it is also relevant for $Y.n$. As $X.f$ is relevant for $Y.n$ and not covered by $n$'s modifies clause, we know by the induction hypothesis that the declaration of $f$ is visible in the module where $n$ is declared. Since $n$ must be declared in the range of the module where $m$ is declared, the declaration of $f$ is also visible in the module where $m$ is declared (by transitivity of abstract field visibility).

**Case 2:** *$Y$ is in $D$.* The proof for this case depends on the context to which $X$ belongs:

**Case 2a:** *$X$ is in $D$ or a descendant of $D$.* Thus, $X.f$ is relevant for $Y.n$ and the reasoning is the same as for Case 1.

**Case 2b:** *$X$ is in a descendant context of $C$, different from $D$ and its descendants.* The locality rule ensures that $X.f$ cannot depend on locations in $D$ or $D$'s descendants. Due to the universe type system, $n$ can only modify concrete locations in $D$ or $D$'s descendants. Consequently, $n$ cannot modify a dependee of $X.f$ and, thus leaves $X.f$ unchanged. This is in contradiction to the assumption that $X.f$ is modified by $Y.n$, that is, this case does not occur.

**Case 2c:** *$X$ is in $C$.* If $X.f$ does not depend on a location in $D$, $X.f$ cannot be modified by the invocation of $n$. Otherwise, if $X.f$ depends on a location in $D$, there must be a chain of dependencies between $X.f$ and $Y.g$ that contains exactly one dynamic dependency `depends` $f'$ `<-` $p.f''$ where $p$ is of a `rep` type (locality rule). Authenticity (Lemma 4.3) guarantees that:

(i) the declaration of $f$ is visible in the module where $f'$ is declared (only static dependencies and dynamic dependencies without `rep` pivot fields are involved in the dependency chain from $f$ to $f'$), and
(ii) the declaration of $f'$ is visible in the module where $p$'s owner type is declared.

Now from the second property of owner types (Property 4.7), we know that

(iii) $p$'s owner type is visible in the module where $m$ is declared.

Thus, authenticity (Lemma 4.3) together with items (i)–(iii) implies that the declaration of $f$ is visible in the module where $m$ is declared and, thus, Case 2 of Property 4.1 applies.

*4.7.2. Proof of Weak Local Completeness*

Let $X.f$ and $Y.g$ be two locations with fields available in module $M$ and $S$ any state. According to weak local completeness (see Definition 4.11) we have to show that one of the following three properties holds in well-formed modules:

1. $X.f$ depends on $Y.g$, and this dependency can be derived from the axioms generated for the depends clauses available in $M$, or
2. $X.f$ does not depend on $Y.g$, and this independence can be derived from the axioms generated for the negation of the depends relation for $M$, or
3. any set of depends clauses that declare a dependency of $X.f$ on $Y.g$ contains a depends clause with a rep pivot $p$ where the owner type of $p$ is not available in $M$.

**Case 1:** *$X.f$ depends on $Y.g$.* From the definition of the depends relation, we know that there is a set of depends clauses $d_1, \ldots, d_n (n \geq 0)$ such that (1) each depends clause $d_i (1 \leq i \leq n)$ has the form `depends` $h_i$ `<-` $h_{i+1}$ or `depends` $h_i$ `<-` $p_i.h_{i+1}$; (2) $h_1$ is $f$; (3) $h_{n+1}$ is $g$.

**Case 1a:** If all $d_i$ are static dependencies, dynamic dependencies where the pivot is not of a rep type, or dynamic dependencies with a rep pivot the owner type of which is available in $M$, then we can conclude by repeated application of the visibility rule and the authenticity lemma that each $d_i$ is available in $M$. Consequently, the axioms generated for the depends relation for $M$ suffice to show that $X.f$ depends on $Y.g$, that is, the first of the above properties holds.

**Case 1b:** There is at least one $d_j$ with a rep pivot $p_j$ the owner type of which is not available in $M$. This means that dynamically there is an object $Z$ such that $Z.p_j.h_{j+1}$ is on the chain

of dependencies from $X.f$ to $Y.g$. Since pivots must not be of readonly types (locality rule), we know that there is a chain of readwrite references from $X$ to $Y$ that passes through $Z$, and since $p_j$ is of a rep type, $Y$ belongs to the context owned by $Z$ or one of its descendants.

Now we show by contradiction that the third of the above properties holds: Assume that there is a set $D$ of depends clauses that declare a dependency of $X.f$ on $Y.g$ and that the owner type of every rep pivot of a depends clause in $D$ is available in $M$.

(i) If $D$ does not contain a depends clause with a rep pivot at all, $X.f$ can only depend on $Y.g$ if $X$ and $Y$ belong to the same context $C$. In this case, $Z$ would also belong to $C$ (locality rule), which leads to a contradiction since $Y$ belongs to the context owned by $Z$ or one of its descendants.

(ii) There are depends clauses with rep pivots in $D$, and the owner types of these rep pivots are available in $M$. From the refined ownership model invariant (see Definition 2.2), we know that each reference chain from $X$ to $Y$ passes through $Z$. Thus, there is a location $Z.p$ on the chain of dependencies from $X.f$ to $Y.g$, and $D$ contains a depends clause with rep pivot $p$. By the second owner type property (see Property 4.7), we conclude that $p$ and $p_j$ have the same owner type $T$. According to the assumption of Case 1b, $T$ is not available in $M$ which contradicts the assumption of Case (ii).

**Case 2:** *$X.f$ does not depend on $Y.g$.* $M$ does not contain depends clauses that allow one to derive that $X.f$ depends on $Y.g$ (otherwise $X.f$ would depend on $Y.g$). Thus, the first requirement of the axioms for the negation of the depends relation is met (see Subsection 4.4). Let $D$ be any possible set of depends clauses that would declare a dependency of $X.f$ on $Y.g$.

**Case 2a:** If for each depends clause in $D$ with a rep pivot $p$ the owner of $p$ is available in $M$, the second requirement is also met and we can use the axioms for the negation of the depends relation for $M$ to show that $X.f$ does not depend on $Y.g$. Thus, the second of the above properties holds.

**Case 2b:** $D$ contains a depends clause with a rep pivot the owner of which is not available in $M$. In analogy to Case 1b, we can prove that this is the case for every set of depends clauses that would declare a dependency of $X.f$ on $Y.g$. Thus, the third of the above properties holds.

### *4.7.3. Proof of Consistency*

To prove that our axiomatization of the depends relation and its negation is consistent, we essentially have to show that whenever a module $M_{ext}$ extends a module $M$

1. the axioms for the depends relation for $M_{ext}$ do not contradict the axioms for the negation of the depends relation for $M$.
2. the axioms for the negation of the depends relation for $M_{ext}$ do not contradict the axioms for the depends relation for $M$.

**Case 1:** Consider two locations $X.f$ and $Y.g$ available in $M$ such that the depends clauses available in $M$ do not declare $X.f$ to depend on $Y.g$. That is, there is an axiom that specifies that $X.f$ does not depend on $Y.g$ if the second requirement for negation axioms is met (see Subsection 4.4). We show that the axiomatization of the depends relation in $M_{ext}$ (i) does not specify $X.f$ to depend on $Y.g$ or (ii) specifies that $X.f$ depends on $Y.g$ in situations in which the second requirement for negation axioms is not met.

If the axiomatization of the depends relation for $M_{ext}$ does not allow one to derive that $X.f$ depends on $Y.g$, property (i) trivially holds. Otherwise, it specifies that $X.f$ depends on $Y.g$. Like in Case 1 of the local completeness proof, we know from the definition of the depends relation that there is a set of depends clauses $d_1, \ldots, d_n (n \geq 0)$ available in $M_{ext}$ such that (1) each depends clause $d_i (1 \leq i \leq n)$ has the form `depends` $h_i$ `<-` $h_{i+1}$ or `depends` $h_i$ `<-` $p_i.h_{i+1}$; (2) $h_1$ is $f$; (3) $h_{n+1}$ is $g$.

**Case 1a:** If all $d_i$ are static dependencies, dynamic dependencies where the pivot is not of a rep type, or dynamic dependencies with a rep pivot the owner of which is available in $M$, then we can conclude by the visibility rule that each $d_i$ is available in $M$. Consequently, the dependency of $X.f$ on $Y.g$ can be derived from the axiomatization of the depends relation for $M$, which contradicts the assumption of Case 1.

**Case 1b:** There is at least one $d_j$ with a rep pivot $p_j$ the owner of which is not available in $M$. Therefore, the second requirement for negation axioms is not met, that is, property (ii) holds.

**Case 2:** Assume that a dependency of $X.f$ on $Y.g$ can be derived from the depends clauses available in $M$, reflexivity, and transitivity. We show that the axiomatization of the negation of the depends relation for $M_{ext}$ does not specify the independence of $X.f$ and $Y.g$, that is, does not lead to an inconsistency.

Each depends clause that is available in $M$ is also available in $M_{ext}$. Therefore, the dependency of $X.f$ on $Y.g$ can also be derived from the depends clauses available in $M_{ext}$, reflexivity, and transitivity. According to the first requirement for axioms for the negation of the depends relation (see Subsection 4.4), this implies that the axiomatization for $M_{ext}$ does not specify the independence of $X.f$ and $Y.g$.

□

## 5. Discussion and Related Work

In this article, we have presented a technique for the modular specification of frame properties. This technique is the first solution to the modularity problem that has been proved sound for both static and dynamic dependencies. It thus solves the three problems described in the introduction (the information hiding, modification of concrete locations, and extended state problems) in a modular way. This has been accomplished by developing and combining:

- A programming model that restricts references to support data abstraction. This programming model can be statically checked by type systems such as the universe type system. It is the basis for both the semantics of modifies clauses and the modularity requirements.
- A semantics for modifies clauses that is independent of the extensions written in using modules, which allows us to treat static and dynamic dependencies in a uniform way, which in turn simplifies formalizations and reasoning.
- Modularity requirements based on explicit dependencies for abstract fields. These requirements guarantee modular soundness and, together with our semantics of modifies clauses, they solve the modularity problem described in Section 1.2.

In this section, we first discuss the expressiveness and limitations of our technique and then compare it to related work.

### 5.1. Expressiveness and Limitations

Our solution to the modular frame problem is achieved by restricting references through the refined ownership model and the admissible dependencies through modularity requirements. However, these restrictions still provide enough flexibility for many common programming patterns as we explain in the rest of this subsection.

#### *5.1.1. The Programming Model*

The refined ownership model allows one to realize dynamic components with encapsulated representations. By enforcing the refined ownership model invariant, whole object structures can be protected from unwanted modifications. Contexts provide encapsulation at the object level, which cannot be directly expressed by access modes in Java.

On the other hand, the refined ownership model is flexible enough to express many common implementation patterns including binary methods (such as `equals`), iterators, several objects sharing one representation, mutual recursive types, etc. (see [26] for examples of these patterns).

However, the programming model is still too restrictive for certain programming patterns. For instance, the refined ownership model does not allow objects to migrate from one context to another. Such patterns are for instance used in initialization methods (e.g., the initialization method of a lexer could take an input stream as parameter [6]). Possible work-arounds are cloning of object structures (thereby losing object identities) or readonly references (thereby losing the ability to modify objects). A promising approach to overcome these shortcomings is the combination of contexts with unique variables. A unique variable guarantees that the referenced object is not aliased at all and can therefore safely be moved to another context [25].

#### *5.1.2. Dependencies*

The set of admissible dependencies of a location are restricted by the expressiveness of our depends clauses and the modularity requirements. We discuss the limitations of both in the following.

*5.1.2.1. Depends Clauses.* In this article, we use a rather restricted form of depends clauses for simplicity. That allows us to check all modularity requirements statically, but requires recursive data structures to be handled by recursive depends-clauses (i.e., depends-clauses where the fields of the dependent and the dependee are identical (e.g., in Fig. 7 the clause `depends values <- next.values`).

In [26], we show that the approach presented in this paper also works for more general depends clauses, which allow almost arbitrary predicates to specify the dependees of an

abstraction. The only restriction is that the field names of the dependees must be specified in the depends clauses.

*5.1.2.2. Locality Rule.* The locality rule is fairly natural: Usually, abstractions of dynamic components abstract from the states of their interface and representation objects. As long as these objects are reachable from an interface object via read-write references, such abstractions meet the locality requirement. Objects that are only reachable via readonly references can be seen as *arguments* of a dynamic component (e.g., the elements in a container). It seems widely accepted that abstractions of a dynamic component must not depend on the states of its arguments (see e.g., the arg mode in [5]).

*5.1.2.3. Authenticity.* Of all modularity requirements, authenticity entails the most onerous restrictions:

(1) Authenticity forces programmers to use `rep` types whenever a type declaration declares a dependency where the field of the dependee is declared in an imported module. In such situations, all restrictions of the universe type system (see above) apply.

(2) Because of authenticity, it is not possible for a location $L$ declared in class $C$ to depend on a location $K$ of the same object if $K$'s declaration is inherited by $C$ and contained in a different module. Otherwise, $K$ and $L$ would belong to the same context, but $L$'s declaration was not visible for the declaration of $K$. Therefore, authenticity does not fully support inheritance. This problem occurs also in different approaches [17] and is not solved yet. In many class libraries, such as the Java API, super- and subclass are often declared in what would be the same module in our technique, and could be handled by our technique; but this would not help users of such a framework to make subclasses outside these modules. However, we need to refine our module system to deal with Java's package concept before this can be studied in detail.

*5.1.3. Summary*

In sum, the main application of abstract fields is the specification of abstract values for dynamic components. In most cases, the dependencies of such abstractions meet all rules of our technique and are not affected by the limitations discussed above. In particular, the dependees are reachable from the interface objects via read-write references, and the representations of the dynamic components are encapsulated in contexts. Therefore, our modularity rules are weak enough to handle such abstractions and, hence, support a wide range of Java programs.

### 5.2. Related Work

In the following, we compare our solution to the frame problem to related work. We focus on the modularity problem, but begin with a more general discussion.

The frame problem was first described in the context of artificial intelligence [23]. Borgida, Mylopoulos, and Reiter [2] give a survey of work on the frame problem in design specifications, however, they do not discuss issues related to object-oriented programming, such as the information hiding, modification of concrete locations, extended state, and modular verification

problems. Their proposal would organize the permission to modify variables around variables instead of around methods. However, the semantics of such specifications in terms of proof obligations is unclear and its capability to support modular verification has not been addressed.

Several specification languages include frame axioms organized around methods. For example VDM-SL [1] says what locations a method may read and write. However, VDM-SL does not solve the information hiding problem and does not support subtyping, so does not address the modularity problem we address in this paper.

Leino's work was motivated in large part by the use of abstract locations in interface specification languages of the Larch family [10]. Larch interface specification languages have modifies clauses that give frame axioms. Since these abstract locations are client-visible, they address the information hiding problem. However, they do not typically address the modification of concrete locations and the extended state problems [19]. Larch/C++ [14] did adopt Leino's dependency declarations [18], which solve these two problems. However, Larch/C++ does not enforce modularity requirements on dependency declarations and thus does not support modular verification of frame properties.

The most closely related work is that of Leino and Nelson [18, 20], which provides the basis for our work. In particular we have adopted the notion of explicitly declared dependencies from Leino and Nelson's work. As described in our paper's introduction; this solves the information hiding, modification of concrete location, and extended state problems. In contrast to our work, Leino and Nelson adopt a semantics that is not based on a notion of relevant locations and an underspecified depends-relation; instead, they use a scope-dependent semantics, where so-called *residues* are used to represent dependees that are not visible in a given scope. Such a semantics is not straightforward to handle in Hoare-style programming logics since the translation of a modifies clause into a pre-post pair depends on the scope in which this translation takes place. Furthermore, modular soundness of the scope-dependent semantics is more difficult to prove. Leino and Nelson have proved modular soundness of their approach for static dependencies; however, they do not claim to have all requirements for dynamic dependencies and have not yet proved soundness for dynamic dependencies in their technique.

Leino and Nelson use different modularity rules for static and dynamic dependencies. We used Leino's and Nelson's rules for static dependencies and generalized them to dynamic dependencies. The generalization relies on the concept of relevant locations which is made possible by the refined ownership model. Most requirements in Leino's and Nelson's work have a direct correspondence in our work, and vice versa. In particular, their modularity requirements enforce data abstraction by controlling aliasing in ways that are similar to those of the refined ownership model. However, the universe type system and our semantics of modifies clauses allow us to formulate the modularity rules in a more syntactic way which simplifies static checking. The disjoint ranges requirement and the swinging pivot restriction needed in [20] are essentially based on properties of the semantics. Concerning the expressiveness of the programming model, the two approaches are not comparable in a strict sense. Certain restrictions imposed by the universe type system are more severe than the analogous requirements of [20]. On the other hand, the two different forms of dynamic dependencies supported by the presented approach provide additional flexibility.

Leino's OOPSLA '98 paper [19] focused on the extended state problem for object-oriented programs. It described a simpler solution to both the information hiding and extended state

problems, which used the concept of a data group. In terms of Leino's earlier work [18] and the solution presented in the present paper, a data group can be modeled by an abstract location whose value contains no information. Since this location contains no information, there is no need to use represents clauses to explain how its value is obtained from the values of other locations. This allows one to relax the authenticity rule. But on the other hand, since data groups have no value, they cannot be used to specify functional behavior in terms of abstract values, which is crucial for verification of OO-programs. When one declares that a location is in a data group, this can be modeled by a dependency declaration, which says that the abstract location depends on the new location; thus, membership in a data group allows the locations in the data group to be modified whenever the data group's name is mentioned in a modifies clause.

In an extension to earlier work on data groups, Leino, Poetzsch-Heffter, and Zhou [21] developed two restrictions, *pivot uniqueness* and *owner exclusion*, that guarantee modular soundness for static and dynamic data groups. Pivot uniqueness confines sharing of objects that are referenced by pivot fields. Owner exclusion is a precondition to procedure calls restricting the use of pivot values as parameters. This work is similar to our approach in that:

- both are based on a kind of pivot uniqueness (although in [21] it is enforced by restrictions on the programming languages, whereas we use the universe type system),
- both use a form of authenticity,
- both work with an underspecified depends relation.

Our approach differs in the techniques used to restrict access to representation objects (owner exclusion vs. universes combined with locality of dependencies).

In our work, the extended state problem is solved by allowing subtypes to introduce additional dependencies for inherited abstract locations. The same technique is used in Larch/C++ [13], the Extended Static Checking project [7, 20], and in [27].

Müller's thesis [26] provides a complete formalization of the technique presented here. It treats static and dynamic dependencies in a uniform way, which simplifies specifications and the modularity requirements, in addition to simplifying the theory. The thesis explains how our approach enables modular verification of frame properties and shows how modular soundness can be achieved by conservative theory extension. The modular specification and verification technique is generalized to type invariants. Moreover, Müller presents a formalization of the universe type system and proves type safety.

The refined ownership model is a basis we build on rather than a contribution of this paper. Therefore, we do not discuss work on aliasing here. The reader is referred to [26] for a comprehensive discussion of such work.

## 6. Conclusions

We extended Java's type system and refined the semantics of frames in JML to allow modular verification of frame properties. The extensions to Java's type system are based on a refined ownership model: the programmer can hierarchically structure the object store into contexts

to which only designated owner objects have direct access. All other references crossing context boundaries into non-descendant contexts have to be declared as readonly. The refined ownership model is enforced by the universe type system. It provides the basis for the modular semantics of modifies clauses and the modularity rules for dependencies.

The key idea behind the modular semantics of modifies clauses is the notion of a relevant location. Locations are relevant for a method when they are in the context containing the receiver object, or a descendent context. A modifies clause only restricts changes to relevant locations. Thus it becomes possible to rigorously enforce the semantics of modifies clauses and yet still write layered abstractions, as in our `Set-List` example.

The refined semantics of frames in JML is based on a more general theory that was developed for modular verification of Java programs [26]. In that work, these ideas are also applied to the modular treatment of class invariants, by considering invariants to be boolean-valued abstract fields. Thus these ideas also lead to modular specification and verification of invariants. We expect that this technique can also be extended to history constraints [22] in JML.

Although our technique can express common implementation patterns such as containers with iterators and mutually recursive types [26], some extensions might be useful in practice. For instance, unique variables would allow objects to migrate from one context to another, and less restrictive modularity rules would provide better support for inheritance [26]. We leave such extensions for future work.

**Acknowledgments**

Thanks to the program committee of the Formal Techniques for Java Programs 2001 workshop, and to Curtis Clifton, Yoonsik Cheon, and Clyde Ruby for comments on earlier versions of this paper. We especially thank Curt for a second reading of the paper.

Thanks to Rustan Leino for many very fruitful discussions about the semantics of frame conditions and dependencies, especially during Arnd Poetzsch-Heffter's visit to Compaq SRC in the Summer of 2001. Thanks to Rustan and to the anonymous referees for comments on an earlier draft of this paper.

The work of Leavens was supported in part by the US NSF under grants CCR-9803843, CCR-0097907, and CCR-0113181; part of this work was done while Leavens was a visiting professor at the University of Iowa.

## APPENDIX A. Depends Clause Syntax in JML

Although Müller's thesis [26] uses a quite general form of dependencies, we use a syntax for depends clauses like that in Leino's thesis [18]. A version of this syntax, adapted to JML, is shown in Figure A1. Besides simplicity, this syntax also permits the modularity rules discussed in Section 4 to be statically checked easily. We leave extensions to this syntax as future work.

## REFERENCES

⟨*depends-decl*⟩ ::= `depends` ⟨*store-ref*⟩ `<-` ⟨*store-ref-list*⟩ `;`
| ⟨*p-modifier*⟩ `depends` ⟨*store-ref*⟩ `<-` ⟨*store-ref-list*⟩ `;`
⟨*p-modifier*⟩ ::= `public` | `protected` | `private`
⟨*store-ref*⟩ ::= ⟨*store-ref-name*⟩ | ⟨*store-ref-name*⟩ ⟨*store-ref-suffix*⟩
⟨*store-ref-name*⟩ ::= ⟨*identifier*⟩ | `this .` ⟨*identifier*⟩ | `super .` ⟨*identifier*⟩
⟨*store-ref-suffix*⟩ ::= `.` ⟨*identifier*⟩ | `[` ⟨*spec-expression*⟩ `]`
⟨*store-ref-list*⟩ ::= ⟨*store-ref*⟩ | ⟨*store-ref-list*⟩ `,` ⟨*store-ref*⟩

Figure A1. Simplified grammar for JML's `depends` clause.


1. D. Andrews. *A Theory and Practice of Program Development.* FACIT. Springer-Verlag, London, UK, 1997.
2. A. Borgida, J. Mylopoulos, and R. Reiter. On the frame problem in procedure specifications. *IEEE Transactions on Software Engineering*, 21(10):785–798, Oct. 1995.
3. J. Boyland, J. Noble, and W. Retert. Capabilities for sharing. In J. L. Knudsen, editor, *ECOOP 2001 — Object-Oriented Programming: 15th European Conference, Budapest, Hungary*, volume 2072 of *Lecture Notes in Computer Science*, pages 1–27, Berlin, June 2001. Springer-Verlag.
4. D. G. Clarke, J. Noble, and J. M. Potter. Simple ownership types for object containment. In J. L. Knudsen, editor, *ECOOP 2001 — Object-Oriented Programming: 15th European Conference, Budapest, Hungary*, volume 2072 of *Lecture Notes in Computer Science*, pages 53–76, Berlin, June 2001. Springer-Verlag.
5. D. G. Clarke, J. M. Potter, and J. Noble. 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.
6. D. L. Detlefs, K. R. M. Leino, and G. Nelson. Wrestling with rep exposure. SRC Research Report 156, Compaq Systems Research Center, 130 Lytton Ave., Palo Alto, Dec 1998.
7. D. L. Detlefs, K. R. M. Leino, G. Nelson, and J. B. Saxe. Extended static checking. SRC Research Report 159, Compaq Systems Research Center, 130 Lytton Ave., Palo Alto, Dec 1998.
8. 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, Mar. 1996. A corrected version is Iowa State University, Dept. of Computer Science TR #95-20c.
9. S. Drossopoulou, S. Eisenbach, B. Jacobs, G. T. Leavens, P. Müller, and A. Poetzsch-Heffter. Formal techniques for Java programs. In J. Malenfant, S. Moisan, and A. Moreira, editors, *Object-Oriented Technology. ECOOP 2000 Workshop Reader*, volume 1964 of *Lecture Notes in Computer Science*, pages 41–54. Springer-Verlag, 2000.
10. J. V. Guttag, J. J. Horning, S. Garland, K. Jones, A. Modet, and J. Wing. *Larch: Languages and Tools for Formal Specification.* Springer-Verlag, New York, NY, 1993.
11. C. A. R. Hoare. Proofs of correctness of data representation. *Acta Informatica*, 1:271–281, 1972.
12. B. Jacobs and E. Poll. A logic for the java modeling language jml. Technical Report CSI-R0018, University of Nijmegen, Computing Science Institute, Teornooiveld 1, 655 Nijmegen, The Netherlands, Nov. 2000.
13. G. T. Leavens. An overview of Larch/C++: Behavioral specifications for C++ modules. In H. Kilov and W. Harvey, editors, *Specification of Behavioral Semantics in Object-Oriented Information Modeling*, chapter 8, pages 121–142. Kluwer Academic Publishers, Boston, 1996. An extended version is TR #96-01d, Department of Computer Science, Iowa State University, Ames, Iowa, 50011.
14. G. T. Leavens. Larch/C++ Reference Manual. Version 5.14. Available in ftp://ftp.cs.iastate.edu/pub/larchc++/lcpp.ps.gz or on the World Wide Web at the URL http://www.cs.iastate.edu/~leavens/larchc++.html, Oct. 1997.

15. G. T. Leavens, A. L. Baker, and C. Ruby. JML: A notation for detailed design. In H. Kilov, B. Rumpe, and I. Simmonds, editors, *Behavioral Specifications of Businesses and Systems*, pages 175–188. Kluwer Academic Publishers, Boston, 1999.
16. G. T. Leavens, A. L. Baker, and C. Ruby. Preliminary design of JML: A behavioral interface specification language for Java. Technical Report 98-06q, Iowa State University, Department of Computer Science, June 2002. See `www.jmlspecs.org`.
17. K. R. M. Leino. A myth in the modular specification of programs. Technical Report KRML 63, Digital Equipment Corporation, Systems Research Center, 130 Lytton Avenue Palo Alto, CA 94301, Nov. 1995. Obtain from the author, at rustan@pa.dec.com.
18. K. R. M. Leino. *Toward Reliable Modular Programs*. PhD thesis, California Institute of Technology, 1995. Available as Technical Report Caltech-CS-TR-95-03.
19. K. R. M. Leino. Data groups: Specifying the modification of extended state. In *OOPSLA '98 Conference Proceedings*, volume 33(10) of *ACM SIGPLAN Notices*, pages 144–153. ACM, Oct. 1998.
20. K. R. M. Leino and G. Nelson. Data abstraction and information hiding. Technical Report 160, Compaq Systems Research Center, 130 Lytton Avenue Palo Alto, CA 94301, 2000.
21. K. R. M. Leino, A. Poetzsch-Heffter, and Y. Zhou. Using data groups to specify and check side effects. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation*, volume 37(5), pages 246–257, June 2002.
22. B. Liskov and J. Wing. A behavioral notion of subtyping. *ACM Trans. Prog. Lang. Syst.*, 16(6):1811–1841, Nov. 1994.
23. J. McCarthy and P. J. Hayes. Some philosophical problems from the standpoint of artificial intelligence. In B. Melzter and D. Michie, editors, *Machine Intelligence 4*, volume 4, pages 463–502. Edinburgh University Press, 1969.
24. B. Meyer. *Object-oriented Software Construction*. Prentice Hall, New York, NY, second edition, 1997.
25. N. Minsky. Towards alias-free pointers. In P. Cointe, editor, *ECOOP '96 European Conference on Object-Oriented Programming*, volume 1098 of *Lecture Notes in Computer Science*, pages 189–209. Springer-Verlag, 1996.
26. P. Müller. *Modular Specification and Verification of Object-Oriented programs*, volume 2262 of *Lecture Notes in Computer Science*. Springer-Verlag, 2002. The author's PhD Thesis. Available from http://www.informatik.fernuni-hagen.de/import/pi5/publications.html.
27. P. Müller and A. Poetzsch-Heffter. Modular specification and verification techniques for object-oriented software components. In G. T. Leavens and M. Sitaraman, editors, *Foundations of Component-Based Systems*, chapter 7, pages 137–159. Cambridge University Press, 2000.
28. P. Müller and A. Poetzsch-Heffter. A type system for controlling representation exposure in Java. Published in [9]., 2000.
29. P. Müller and A. Poetzsch-Heffter. Universes: A type system for alias and dependency control. Technical Report 279, FernUniversität Hagen, 2001.
30. J. Noble, J. Vitek, and J. Potter. Flexible alias protection. In E. Jul, editor, *ECOOP '98 – Object-Oriented Programming, 12th European Conference, Brussels, Belgium*, volume 1445 of *Lecture Notes in Computer Science*, pages 158–185. Springer-Verlag, July 1998.
31. J. M. Wing. Writing Larch interface language specifications. *ACM Trans. Prog. Lang. Syst.*, 9(1):1–24, Jan. 1987.