Modular Specification of Frame Properties in JML

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

TR #01-03
April 2001

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, 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 langauges; 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;

Submitted for publication

Department of Computer Science
226 Atanasoff Hall
Iowa State University
Ames, Iowa 50011-1040, USA

Modular Specification of Frame Properties in JML

Peter Müller[1], Arnd Poetzsch-Heffter[1], and Gary T. Leavens[2]

[1] FernUniversität Hagen, 58084 Hagen, Germany,
{Peter.Mueller, Arnd.Poetzsch-Heffter}@Fernuni-Hagen.de,
[2] Iowa State University, Ames, Iowa, 50011-1040, USA
leavens@cs.iastate.edu

Abstract. 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 restricts aliasing, and by modularity requirements for dependencies. For concreteness, we adapt this technique to the Java Modeling Language, JML.

1 Introduction

In an interface specification language, a *frame property* describes what locations a method may modify, and, implicitly, what locations it may not modify [BMR95]. This is often specified using a *modifies clause* [GHG+93,Win87].

We address three problems for specification and verification of frame properties: (1) Information hiding—The concrete (e.g., private) fields of a class should be hidden from its clients, even in specifications, yet the frame properties of (public) specifications must somehow permit those locations to be modified. (2) Extended state—When a subclass overrides a method, it may need to modify additional fields it declared; yet the demands of behavioral subtyping (e.g., [LW94,DL96]) would seem to prohibit modification of these additional fields [Lei98]. (3) Modularity—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 context in which the methods will be used. However, in general one cannot know what locations might be found in a program that extends a given class or interface.

Leino's work [Lei95] solves problems (1) and (2) by introducing abstract fields with explicitly declared dependencies and a refined semantics of modifies clauses (see below). This paper explains part of Müller's thesis [Mül01], which builds on Leino's work and provides a modular sound solution to problem (3).

1.1 Related Work

When modeling objects as records containing possibly abstract locations, one needs a way to specify the correspondence between abstract and concrete locations. To do this, Leino introduced depends and represents clauses [Lei95,Lei98].

A represents clause says how an abstract location's value is determined from various concrete locations. To a first approximation, a depends clause says what concrete locations are used to determine the abstract location's value. More precisely, a dependency declaration allows dependees to be modified whenever the abstract location is named in a modifies clause. Thus, in JML, "`depends absloc <- concloc`" says that `concloc` can be modified whenever `absloc` is modifiable.

To support the specification of extended state, a subtype may declare that an inherited abstract field depends on the fields it declares. Such dependencies allow overriding methods in subclasses to modify their extended state.

Leino and Nelson distinguish *static dependencies*, of the form "`depends f <- g`", and *dynamic dependencies*, of the form "`depends f <- p.g`", in which abstract field `f` depends on field `g` of the *pivot object* `p`. Leino and Nelson handle static and dynamic dependencies in different ways, that is, by different desugaring of modifies clauses, and different modularity rules. Although Müller's thesis [Mül01] treats both cases uniformly, in this paper, to avoid introducing additional concepts, we also distinguish them.

Leino and Nelson use scope-dependent depends relations [Lei95], which lead to a scope-dependent meaning of modifies clauses. Soundness 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.

1.2 Approach

To solve the first two problems described above, we follow Leino and Nelson [Lei95,LN00], using abstract fields and explicitly declared dependencies. We explain the ideas by applying them to the Java Modeling Language (JML) [LBR01,LBR99], which allows the specifier to declare abstract fields by using the modifier "`model`". JML also allows one to declare dependencies, although it does not yet incorporate the restrictions we propose here.

Our solution to the modularity problem entails three steps: (1) We define a programming model that hierarchically structures the object store into so-called contexts and restricts references between contexts [MPH00,MPH01,Mül01].

(2) Dependency declarations generate a theory for dependencies declared in a given set of modules. This depends relation does not specify dependencies for extensions to the given set of modules. Because of this underspecification one can only prove properties about a module that hold in well-formed extensions. Thus modular soundness is 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.

(3) We impose three modularity requirements to restrict the permissible dependencies of abstract locations. These restrictions allow us to prove a modularity theorem that makes modular verification of frame properties in method implementations possible.

A detailed presentation of these ideas, including all formalizations and proofs, but not their application to JML, is found in [Mül01].

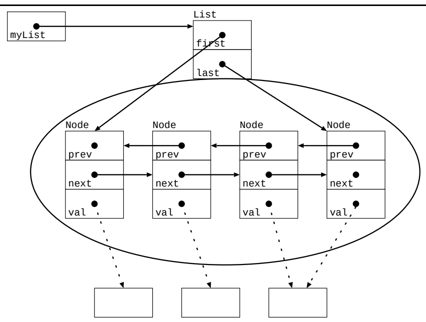

Fig. 1. Nodes in a context (the oval). The owner object sits atop the context it owns.

2 The Programming Model

To achieve modularity, dependencies must be controlled. There are two problems, both of which involve aliasing: (1) *Representation exposure* occurs when objects inside the representation of an object X may be referenced by objects outside of X's representation. (2) *Dependencies on argument objects* occur when an object X's abstract value is determined by the abstract values of objects, called *argument objects*, outside X's representation. Both problems allow modification of an object's abstract value in ways that cannot be controlled by its implementation.

To prevent representation exposure, the object store is structured into a hierarchy of contexts. *Contexts* are disjoint groups of objects. There is a root context. All other contexts have an *owner object* in their parent context. Aliasing is controlled by the following invariant: Every reference chain from objects in the root context to an object in a context C passes through C's owner. Thus, an owner object can control access to objects in its context. This structure of the object store is called the *ownership model* [CPN98].

The ownership model is not sufficient to prevent dependencies on argument objects because it allows objects inside a context to reference argument objects in ancestor contexts. We refined the ownership model in two ways [MPH01,Mül01]: (a) references to argument objects are made explicit by marking them `readonly`, and (b) readonly references can point to any object, not only to objects in ancestor contexts. Access via readonly references is restricted to reading operations without side-effects. This *refined ownership model* is more general than the original one. In this refined model, we prevent dependencies on argument objects by forbidding dependencies via readonly references.

Figure 1 illustrates our *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). Consequently, abstract fields of the list must not depend on fields of these objects.

To enforce the refined ownership model's invariant, we use the universe type system [MPH01,Mül01]. Besides tagging types as readonly, this type system also distinguishes between references that remain inside a context and references to objects that belong to the *descendant context* owned by the `this`-object. References of the latter kind are tagged with the keyword `rep` [CPN98].

3 Specification of frame properties in JML

3.1 Data abstraction in JML

Data abstractions in JML are specified using abstract locations, i.e., model fields. For example, consider the specifications of `List` in Figure 2 and `Node` in Figure 3.

```
//@ model import edu.iastate.cs.jml.models.*;
public abstract class List {
    //@ public model non_null JMLObjectSequence listValue;
    protected /*% rep %*/ Node first, last;
    //@ protected depends listValue <- first, first.values, last;
    /*@ 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) {
            last = new /*% rep %*/ Node(null, null, o);
            first = last;
        } else {
            last.next = new /*% rep %*/ Node(null, last, o);
            last = last.next;
        }
    }

    /* ... */
}
```

Fig. 2. A JML specification of the Java class `List`, of doubly-linked lists.

The class `List` declares a public model field `listValue`, which describes the abstract value of a `List` object. In the class `Node`, the model field `values` forms part of the abstract value of `Node` objects. In JML, method specifications precede the method header, preconditions are introduced by the keyword `requires` and postconditions by the keyword `ensures`. For example, in the specification of `List`'s method `append`, the postcondition describes the abstract effect of `append` on the model field `listValue`.

```
//@ model import edu.iastate.cs.jml.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));  @*/

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

Fig. 3. The JML specification of the Java class `Node`.

3.2 Explicit dependencies in JML

Although Müller's thesis [Mül01] uses a quite general form of dependencies, we use a syntax for depends clauses like that in Leino's thesis [Lei95]. Besides simplicity, this syntax also permits the restrictions discussed in Section 4 to be statically checked easily. We leave extensions to this syntax as future work.

For example, in the class `List`, the model field `listValue` is represented by a sequence determined by `first` and `first.values`. Hence `listValue` is also declared to depend on these fields. Although the represents clause for `List` does not use the field `last`, that field is listed in the depends clause, to permit it to be modified whenever `listValue` is modifiable. Similarly, in class `Node`, the model field `values` depends on `next`, `next.values`, `prev`, and `val`.

3.3 Modifies Clauses in JML

An example of a modifies clause in JML appears in the specification of `List`'s `append` method. It says that the method may modify `listValue`.

The semantics of the modifies clause is that all relevant locations that either are named in the clause or on which such locations depend may be modified. A location is *relevant* to the execution of a non-static method m if it is either in the context that contains m's receiver or a descendant context of the one that contains m's receiver. 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 locations in descendant contexts. Since the field `first` in `List` is declared using the keyword `rep`, the object `myList.first` points to is in the context owned by `myList` (see Figure 1), which is thus a descendant context of the context that contains `myList`. Since the `next` fields of `Node` objects are not

```
//@ model import edu.iastate.cs.jml.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); }
    }
}
```

Fig. 4. The JML specification of the Java class `Set`.

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 fields of these nodes are also relevant locations. That is, the call may modify `myList.first`, `myList.first.values`, `myList.last`, and all the fields of the nodes reachable from `myList.first` via the `next` field.

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 4. `Set`'s model 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 other relevant 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 5). 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 model field `setValue`, which it does when it modifies the abstract value of `theList`? The semantics allows this because it underspecifies the locations that `append` can modify, since it only describes the modification of relevant locations, and `setValue` is not relevant for the call `theList.append(o)`. The reason for this is that a context's owner is not contained in the context it owns, and `theList` is in the context owned by the receiver in `Set`'s `insert`

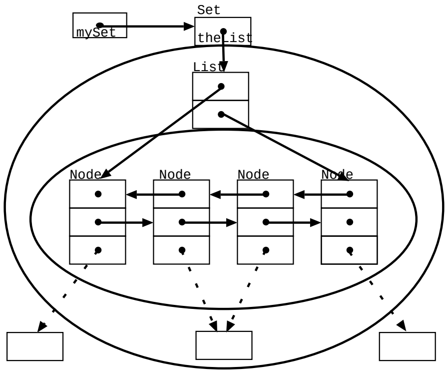

Fig. 5. Object Structure for a `Set` object.

method (see Figure 5). Hence in `Set`'s `insert` method, `this.setValue` is not a relevant location for the call to `theList.append(o)`.

Responsibility for verifying frame properties is divided. A method's implementor is responsible for the locations relevant to its executions, as specified in its modifies clause, and the method's caller is responsible for other locations. For example, `append`'s implementor is responsible for verifying the frame properties in its modifies clause. When verifying the call to `append` in `Set`'s `insert` method, one uses `append`'s modifies clause and `Set`'s depends clauses to reason about modification of `Set`'s fields `theList` and `setValue`.

4 Modularity and Dependencies

To achieve modularity, we impose three requirements on dependencies:

Locality Requirement: Abstractions of an object X can only depend on locations in the context that contains X or its descendants. That is, they may depend on locations in X's representation, but not on argument objects.

Authenticity Requirement: The declaration of an abstract location L in a context C must be visible in every scope that contains a method m that could—if invoked on a target object in C—modify L. Thus the verifier of m can determine all relevant locations that m might modify.

Visibility Requirement: Whenever two locations are declared in a scope S, the dependencies in S must allow one to determine whether these locations depend on each other or not.

We enforce these requirements by statically checking the following rules for single depends clauses of the form "`depends` f `<-` g" or "`depends` f `<-` $p.g$".

Locality Rule: For dynamic dependencies, the pivot field must not hold a readonly reference; that is, p must not be of a `readonly` type.

Authenticity Rule: For static dependencies and for dynamic dependencies where the pivot field is not of a `rep` type, f must be declared in the scope of g. For dynamic dependencies where the pivot field is of a `rep` type, f must be declared in the scope of the owner type of p. In most implementations such as in our examples, the *owner type* of a field is its declaration type (see [Mül01] for a precise definition).

Visibility Rule: Static and dynamic dependencies where the pivot field is not of a `rep` type must be declared in the scope of g. Dynamic dependencies where the pivot field is of a `rep` type must be declared in the scope of p's owner type.

To verify frame properties of a method m, one has to prove that m leaves all relevant locations that are not covered by m's modifies clause unchanged. This proof obligation can be shown for those locations that are declared in the scope of m by referring to their representations and dependencies. For all other relevant locations, the locality and authenticity requirements guarantee that they are not modified by m, as stated by the following modularity theorem:

A method m can only modify relevant locations that are declared in m's scope.

A sketch of this theorem's proof is contained in the appendix. A formalization of the theorem and the full proof can be found in [Mül01]. The modularity theorem's proof shows that the modularity requirements in combination with the universe programming model are strong enough to enable modular verification of frame properties. Similar requirements are used in [LN00].

5 Conclusions

We extended the Java Modeling Language by constructs to specify frame properties in a modular way. The extension is 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 have to be declared readonly. The ownership model is enforced by the universe type system. It provides the basis to refine the semantics of the modifies clause and to define context conditions that guarantee the modularity of specification and verification of frame properties.

The JML extensions are based on a more general framework that was developed for modular verification of Java programs [Mül01]. 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.

Although our technique can express common implementation patterns such as containers with iterators and mutually recursive types [Mül01], 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 [Mül01]. We leave such extensions for future work.

Acknowledgments

The work of Leavens was supported in part by the US NSF under grant CCR-9803843, and was done while Leavens was visiting the University of Iowa.

References

[BMR95] Alex Borgida, John Mylopoulos, and Rayomnd Reiter. On the frame problem in procedure specifications. *IEEE Transactions on Software Engineering*, 21(10):785–798, October 1995.

[CNP01] D. G. Clarke, J. Noble, and J. M. Potter. Simple ownership types for containment. In *European Conference on Object-Oriented Programming, ECOOP 2001*, Lecture Notes in Computer Science. Springer-Verlag, 2001. (to appear).

[CPN98] 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.

[DEJ+00] Sophia Drossopoulou, Susan Eisenbach, Bart Jacobs, Gary T. Leavens, Peter Müller, and Arnd Poetzsch-Heffter. Formal techniques for Java programs. In Jacques Malenfant, Sabine Moisan, and Ana Moreira, editors, *Object-Oriented Technology. ECOOP 2000 Workshop Reader*, volume 1964 of *Lecture Notes in Computer Science*, pages 41–54. Springer-Verlag, 2000.

[DL96] Krishna Kishore Dhara and Gary T. Leavens. Forcing behavioral subtyping through specification inheritance. In *Proceedings of the 18th International Conference on Software Engineering, Berlin, Germany*, pages 258–267. IEEE Computer Society Press, March 1996. A corrected version is Iowa State University, Dept. of Computer Science TR #95-20c.

[GHG+93] John V. Guttag, James J. Horning, S.J. Garland, K.D. Jones, A. Modet, and J.M. Wing. *Larch: Languages and Tools for Formal Specification*. Springer-Verlag, New York, NY, 1993.

[LBR99] Gary T. Leavens, Albert L. Baker, and Clyde Ruby. JML: A notation for detailed design. In Haim Kilov, Bernhard Rumpe, and Ian Simmonds, editors, *Behavioral Specifications of Businesses and Systems*, pages 175–188. Kluwer Academic Publishers, Boston, 1999.

[LBR01] Gary T. Leavens, Albert L. Baker, and Clyde Ruby. Preliminary design of JML: A behavioral interface specification language for Java. Technical Report 98-06m, Iowa State University, Department of Computer Science, February 2001. See `www.cs.iastate.edu/~leavens/JML.html`.

[Lei95] K. Rustan M. Leino. *Toward Reliable Modular Programs*. PhD thesis, California Institute of Technology, 1995. Available as Technical Report Caltech-CS-TR-95-03.

[Lei98] K. Rustan 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, October 1998.

[LH94] K. Lano and H. Haughton, editors. *Object-Oriented Specification Case Studies*. The Object-Oriented Series. Prentice Hall, New York, NY, 1994.

[LN00] K. Rustan M. Leino and Greg Nelson. Data abstraction and information hiding. Technical Report 160, Compaq Systems Research Center, 130 Lytton Avenue Palo Alto, CA 94301, 2000.

[LW94] Barbara Liskov and Jeannette Wing. A behavioral notion of subtyping. *ACM Transactions on Programming Languages and Systems*, 16(6):1811–1841, November 1994.

[MPH00] Peter Müller and Arnd Poetzsch-Heffter. A type system for controlling representation exposure in Java. Published in [DEJ$^+$00]., 2000.

[MPH01] P. Müller and A. Poetzsch-Heffter. Universes: A type system for alias and dependency control. Technical Report 279, FernUniversität Hagen, 2001.

[Mül01] Peter Müller. *Modular Specification and Verification of Object-Oriented programs.* PhD thesis, FernUniversität Hagen, Germany, March 2001.

[SBC92] Susan Stepney, Rosalind Barden, and David Cooper, editors. *Object Orientation in Z.* Workshops in Computing. Springer-Verlag, Cambridge CB2 1LQ, UK, 1992.

[Win87] Jeannette M. Wing. Writing Larch interface language specifications. *ACM Transactions on Programming Languages and Systems*, 9(1):1–24, January 1987.

A Sketch of the Modularity Theorem's Proof

In the following, we sketch the field update case of the proof of the modularity theorem from Section 4. The proof for method invocations is similar.

Proof Sketch. Let m be executed in context C (i.e., the receiver is in C). If m updates $Y.g$, the universe type system guarantees that Y is in C or in one of C's immediate descendents. 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 f is declared in m's scope:

Case 1: Y is in C. If $X.f$ is relevant for m, then by the the locality rule X is in C. Thus, X and Y are in the same context, and the authenticity rule ensures that f is declared in g's scope. Since g is accessible in m, f is in m's scope.

Case 2: Y is in an immediately-descendent context D of C. Due to locality, X is in D or in C. The former case is analogous to Case 1. In the latter case: a dynamic dependency must be involved with a pivot field p of a `rep` type, the owner type of p is in the scope of m (by the universe type system), and f is declared in the scope of p's owner type (by the authenticity rule). Thus, f is declared in m's scope.