A verification methodology for model fields

K. Rustan M. Leino [1] and Peter Müller [2]

[1] Microsoft Research, leino@microsoft.com
[2] ETH Zürich, peter.mueller@inf.ethz.ch

Abstract. Model fields are specification-only fields that encode abstractions of the concrete state of a data structure. They allow specifications to describe the behavior of object-oriented programs without exposing implementation details.
This paper presents a sound verification methodology for model fields that handles object-oriented features, supports data abstraction, and can be applied to a variety of realistic programs. The key innovation of the methodology is a novel encoding of model fields, where updates of the concrete state do not automatically change the values of model fields. Model fields are updated only by a special pack statement. The methodology guarantees that the specified relation between a model field and the concrete state of an object holds whenever the object is valid, that is, is known to satisfy its invariant.
The methodology also improves on previous work in three significant ways: First, the formalization of model fields prevents unsoundness, even if an interface specification is inconsistent. Second, the methodology fully supports inheritance. Third, the methodology enables modular reasoning about frame properties without using explicit dependencies, which are not handled well by automatic theorem provers.

1 Introduction

The development of object-oriented programs makes use of mutable objects, aliasing, subtyping, and modularity. We are interested in verifying such programs. To do that, we need specifications with data abstraction and a systematic way (a methodology) of reasoning. Existing methodologies either do not address these characteristics of object-oriented programming or do not support data abstraction in a satisfactory way. In this paper, we present a methodology that addresses these problems and that can be applied to a wide variety of realistic programs.

Specifications that are visible to client code must be expressed in an implementation-independent way to support information hiding. This can be achieved by using data abstraction [11], that is, by mapping the concrete state of a data structure to an abstract value. A standard example is to map the state of a singly-linked list to a mathematical sequence. The behavior of the list class can be expressed in terms of the abstract value of list objects, that is, in terms of the sequence. A convenient way to support data abstraction in specification languages for object-oriented programs is by *model fields* [5, 14, 15, 19]. In contrast to ordinary (*concrete*) fields, a program cannot directly assign to model fields. Instead, model fields are specification-only fields whose values are determined by mappings from an object's concrete state.

Class $Rectangle$ in Fig. 1 illustrates how model fields are used in specifications. A $Rectangle$ object stores the coordinates of two opposite corners, as expressed by the

invariant. The model field $width$ is used to refer to the width of the rectangle in specifications. The value of $width$ is the difference between the x-coordinates of the corners, $x2$ and $x1$. This relation between the model field $width$ and the concrete fields $x1$ and $x2$—the so-called *constraint* for $width$—is expressed by the **constrained_by** part of the model field declaration. The declaration of the model field $height$ is analogous.

Method $ScaleH$ scales the rectangle horizontally by a given percentage. The ensures clause uses the model field $width$ to express the functionality of $ScaleH$ without referring to the concrete implementation. The modifies clause allows method $ScaleH$ to change the values of the fields $width$ and $x2$. The second requires clause as well as the unpack and pack statements are required by the methodology for object invariants we build on and will be explained in Sec. 4.

```
class Rectangle {
  int x1, y1, x2, y2 ;    // lower left and upper right corner
  invariant x1 ≤ x2 ∧ y1 ≤ y2 ;
  model int width constrained_by width = x2 − x1 ;
  model int height constrained_by height = y2 − y1 ;
  Rectangle() {
    x1 := 0 ;  y1 := 0 ;
    x2 := 1 ;  y2 := 1 ;
    pack this as Rectangle ;
  }
  void ScaleH(int factor)
    requires 0 ≤ factor ;
    requires inv = Rectangle  ∧ ¬committed ;
    modifies width, x2 ;
    ensures width = old(width) ∗ factor/100 ;
  {
    unpack this from Rectangle ;
    x2 := (x2 − x1) ∗ factor/100 + x1 ;
    pack this as Rectangle ;
  }
}
```

Fig. 1. A specification with model fields.

The basic concept of data abstraction is well-understood. However, existing verification techniques for model fields (including our own previous work) suffer from soundness, modularity, expressiveness, or practicality problems. Our contribution in this paper is a verification methodology that solves these problems. The key innovation is to treat model fields as if they were stored in the heap and updated automatically in a systematic way. This treatment reduces reasoning about model fields to simpler concepts.

We illustrate the problems any verification methodology for model fields has to address in the next section and explain our approach to their solution in Sec. 3. Our approach is based on the Boogie methodology for object invariants [1, 17], which we summarize in Sec. 4. We present the details of our methodology for model fields in Sec. 5. The rest of the paper discusses related work and offers some conclusions.

2 Problems

A verification methodology for model fields has to address two major issues: (a) the meaning of model fields and their constraints, and (b) the meaning of frame properties in the presence of model fields. We discuss these issues in the following.

2.1 Meaning of model fields

In existing methodologies [4, 15, 18–20], the meaning of a model field is defined by an *abstraction function* that maps a receiver object and a heap to the model field's value. This abstraction function is specified by a programmer-provided constraint. The problem with this meaning is that if a programmer specifies inconsistent constraints, then the abstraction functions are not well-defined, which lends itself to unsound reasoning. Inconsistent constraints occur in two situations.

First, constraints can be unsatisfiable. For instance, consider the abstraction function abs_{len} of an integer model field len of a class $List$. If the model field len is constrained by $len = len + 1$, then abs_{len} has to satisfy the unsatisfiable $abs_{len}(l, H) = abs_{len}(l, H) + 1$ for any $List$ object l and heap H. In practice, such ill-formed specifications are far less obvious than this example, because they typically involve strengthening of inherited constraints or cyclic dependencies among several model fields.

Second, abstraction functions of model fields are typically well-defined only for objects that satisfy their invariants. Applying an abstraction function to an object whose invariant is temporarily violated can lead to inconsistencies. For instance, consider a linked list implementation, where the invariant requires the list to be acyclic. In such an implementation, the model field len for the length of the list could be constrained by a conditional expression such as $len = (next = \mathbf{null})\ ?\ 1 : next.len + 1$, where $next$ is the field that stores the next node of the list. This, too, is inconsistent if applied to a cyclic list. For instance, if $l = l.next$ for a list l, then abs_{len} again has to satisfy the unsatisfiable $abs_{len}(l, H) = abs_{len}(l, H) + 1$.

Both kinds of inconsistent constraints can be avoided in carefully written specifications. However, specifications found in practice contain flaws. A verification methodology has to ensure that these flaws are detected during verification and do not lead to unsound reasoning.

2.2 Meaning of frame properties

The *frame properties* of a method limit the effects the method may have on the program state. This is crucial when reasoning about calls. Frame properties are typically specified using modifies clauses. Roughly speaking, a modifies clause lists the concrete and model locations a method is allowed to modify.

Any update of a field $x.f$ potentially affects each model field that depends on $x.f$. A model field $o.m$ *depends* on a field $x.f$ if the value of $x.f$ constrains the value of $o.m$. For instance, for a $Rectangle$ object o, $o.width$ depends on $o.x1$ and $o.x2$ because these fields are mentioned in the constraint for $width$.

When the meaning of a model field is given by an abstraction function, any modification of the heap, for instance, by an update of a field $x.f$, will have an *instant*

effect on all dependent model fields. That is, the value of these model fields is changed simultaneously with the update of $x.f$.

Instant effects lead to a modularity problem, which is illustrated by class $Legend$ in Fig. 2. $Legend$ objects display some text within a bounding box. The box is represented by a $Rectangle$ object. The **rep** modifier in the declaration of the field box is used to express ownership and will be explained in Sec. 4. The model field $maxChars$ yields the maximum number of characters that fit into one line in the box at a given font size.

The value of $maxChars$ depends on the width of the $Rectangle$ object box. Therefore, if method $ScaleH$ is executed on a $Rectangle$ object r, it potentially modifies $maxChars$ for any $Legend$ object that uses r. However, we cannot require $ScaleH$ to declare this potential modification in its modifies clause, since the implementor of $ScaleH$ need not be aware of class $Legend$ (in fact, $Legend$ might have been implemented long after $Rectangle$).

```
class Legend {
  rep Rectangle box ;
  int fontSize ;
  invariant box ≠ null ∧ fontSize > 0 ;
  model int maxChars constrained_by maxChars = box.width/fontSize ;
  void Reset()
    requires inv = Legend ∧ ¬committed ;
    modifies maxChars ;
  {
    unpack this from Legend ;
    box.ScaleH(0) ;
    pack this as Legend ;
  }
  // constructors and other methods omitted.
}
```

Fig. 2. A client of class $Rectangle$.

An analogous modularity problem occurs when model field constraints refer to inherited fields. For instance, if a subclass $MyRectangle$ of class $Rectangle$ declares a model field $area$ that depends on the inherited field $x2$, method $ScaleH$ potentially modifies $area$ without listing the field in its modifies clause.

A useful verification methodology for model fields must address the modularity problem of frame properties for aggregate objects (such as $Legend$ objects) and subclasses (such as $MyRectangle$).

3 Approach

In this section, we explain the general ideas that allow us to solve the problems described in the previous section. To focus on the essentials, we ignore subtyping in this overview, but we will include it in Sec. 5 when we explain our methodology in detail.

Our methodology for model fields builds on the Boogie methodology for object invariants [1, 17]. In the Boogie methodology, an object is either in a *valid* or a *mutable*

state. Only when in a valid state, an object is guaranteed to satisfy its invariant, and only fields of objects in a mutable state can be assigned. The transition from valid to mutable and back is performed by two special statements, **unpack** and **pack**.

Principles. Our methodology is based on the following three principles:

1. *Validity principle:* The declared constraint for a model field m constrains the value of $o.m$ only if the object o is valid, that is, if o's invariant is known to hold.
2. *Decoupling principle:* The change of the value of a model field is decoupled from the updates of the fields it depends on. Instead of applying an abstraction function to obtain the value of a model field $o.m$, a stored value is used. The stored value of $o.m$ is not updated instantly when a dependee field is modified, but only at the point when o is being packed.
3. *Mutable dependent principle:* If a model field $o.m$ depends on a field $x.f$, then the dependent object o must be mutable whenever x is mutable.

The validity principle defines the meaning of model fields and avoids inconsistencies due to temporarily broken invariants. Inconsistencies due to unsatisfiable constraints (e.g., $len = len + 1$) are avoided by assertions, as we explain in Sec. 5.2.

The decoupling principle solves the modularity problems of frame properties. Consider a method M that updates a field $x.f$. Because of decoupling, this update does not have an instant effect on a dependent model field $o.m$. That is, $o.m$ remains unchanged and, therefore, need not be mentioned in M's modifies clause (as long as M does not pack o).

The mutable dependent principle and the validity principle are prerequisites for decoupling to be sound. Consider a model field $o.m$ that depends on a field $x.f$. Updating $x.f$ potentially causes $o.m$ not to satisfy its constraint any more. However, the Boogie methodology requires that x be mutable when $x.f$ is updated. Therefore, the mutable dependent principle implies that o is also mutable, and the validity principle allows $o.m$ not to satisfy its constraint.

There are several ways to enforce the mutable dependent principle. The one we use in this paper is to organize objects in an ownership hierarchy [17]. A model field $o.m$ is allowed to depend on fields of an object x only if x is (transitively) owned by o. The Boogie methodology guarantees that the (transitive) owner objects of a mutable object are themselves mutable.

Example. To illustrate how these principles work, we revisit the $Legend$ example (Fig. 2). Let l be a $Legend$ object and let r be the $Rectangle$ object stored in $l.box$. The modifier **rep** in the declaration of box indicates that l owns r. Thus, the Boogie methodology guarantees that l is mutable whenever r is mutable. Since the model field $l.maxChars$ depends on $r.width$, this ownership relation is required by the mutable dependent principle.

Consider the execution of method $ScaleH$ invoked from $Legend$'s method $Reset$. The first statement of $ScaleH$ unpacks the receiver object (that is, r) to permit updates of its fields. By the decoupling principle, the subsequent update of $x2$ does not change the value of the model field $r.width$, even though $width$ depends on $x2$. In the state

after the update, the value of $r.width$ in general does not satisfy the specified constraint because the concrete state has changed, but the value of the model field has not (yet) been adapted. This discrepancy is permitted by the validity principle since r is mutable.

The value of $r.width$ is brought up to date when r is packed. Again, by the decoupling principle, this update does not instantly affect the value of $l.maxChars$. Consequently, this model field does not have to be mentioned in $ScaleH$'s modifies clause, which shows the modularity of the approach.

Updating $r.width$ potentially causes $l.maxChars$ not to satisfy its constraint. However, since l is mutable, this discrepancy is permitted by the validity principle. It will be resolved when l is packed in method $Reset$.

4 Background: the Boogie methodology for object invariants

In this section, we summarize those parts of the Boogie methodology for object invariants [1] that are needed in the rest of this paper. The motivation for the design and the technical details are presented in our earlier paper [17].

Explicit representation of when invariants have to hold. To handle temporary violations of object invariants and reentrant method calls, the Boogie methodology represents explicitly in every object's state whether the object invariant is required to hold or allowed to be violated. For this purpose, it introduces for every object a concrete field inv that ranges over class names. If $o.inv <: T$ for a T object o (where $<:$ denotes the subtype relation), then o's invariants declared in class T and its superclasses must hold and we say o is *valid for* T. If o is not valid for T then the invariant of o declared in T are allowed to be temporarily violated and we say o is *mutable* for T.

The inv field can be used in method specifications, but cannot be assigned directly by the program. Instead, the Boogie methodology provides two special statements: $\mathbf{unpack}\ o\ \mathbf{from}\ T$ and $\mathbf{pack}\ o\ \mathbf{as}\ T$ change $o.inv$ from T to T's direct superclass and back, respectively. Before setting inv to T, the pack statement checks that the object invariant declared in class T holds for o.

Since the update of a field $o.f$ potentially breaks the invariant of o, $o.f$ is allowed to be assigned only at times when o is mutable for the class F that declares f. To enforce this policy, each update of $o.f$ is guarded by an assertion $F \lneq: o.inv$. This assertion is crucial for the soundness of our methodology, see Sec. 5.4.

Ownership. The Boogie methodology handles aggregate objects by guaranteeing that the validity of an object implies the validity of its component objects. Providing this guarantee requires some form of *aliasing control*, a discipline on the use of object references. The Boogie methodology uses the notion of *ownership* for aliasing control, associating with every object a unique owner object. That is, an aggregate object is the owner of its component objects. Objects outside the aggregate are allowed to reference component objects, but these references are only of limited use.

To encode ownership, the Boogie methodology introduces two additional concrete fields for every object: a field $owner$ that ranges over pairs $\langle o, T \rangle$, where o is the owner object and T is a superclass of the dynamic type of o at which the ownership

is established, and a boolean field $committed$. Like inv, these fields can be used in method specifications, but cannot directly be assigned by the program. The owner of an object is set when the object is created. Because it would be a distraction in this paper, we omit a program statement for changing the $owner$ field (but see [17]).

Let p be an object that is owned by $\langle o, T\rangle$. The fact that p is committed (that is, $p.committed = true$) expresses that p is valid for its dynamic type, and o is valid for T. The $committed$ field is used to implement a protocol that enforces that an owner object is unpacked before the owned object is unpacked. Packing is done in the reverse order. More precisely, this protocol ensures that the owner object o is mutable for the owner type T whenever p is mutable.

In connection with the fact that field updates are allowed only for mutable objects, this protocol guarantees that the following two program invariants hold in each reachable execution state of a program: If an object o is valid for a class T, then the object invariants declared in T hold for o and all objects owned by $\langle o, T\rangle$ are committed (see J1 below). Committed objects are valid for their dynamic type (see J2 below). (Here and throughout the paper, quantifications over object references range over non-null references to allocated objects.)

J1: $(\forall\, o, T \bullet\ o.inv <: T \Rightarrow Inv_T(o) \wedge (\forall\, \mathbf{object}\ p \bullet\ p.owner = \langle o, T\rangle \Rightarrow p.committed\ ))$

J2: $(\forall\, o \bullet\ o.committed \Rightarrow o.inv = \mathbf{typeof}(o)\ )$

The protocol is implemented by the unpack and pack statements. The act of packing an object o for a class T also commits the objects owned by $\langle o, T\rangle$ by setting their $committed$ fields to $true$. This operation requires these owned objects to be previously uncommitted and valid for their dynamic types. Unpacking an object o from a class T requires o to be uncommitted and sets the $committed$ field of the objects owned by $\langle o, T\rangle$ to $false$. We formalize these statements by the pseudo code shown in Fig. 3. $Inv_T(o)$ denotes the expression that says that o satisfies the object invariant declared in class T, $\mathbf{typeof}(o)$ is the dynamic type of object o, and $Super(T)$ denotes the direct superclass of T.

```
unpack o from T  ≡
  assert o ≠ null ∧ o.inv = T ∧ ¬o.committed ;
  o.inv := Super(T) ;
  #foreach object p such that p.owner = ⟨o, T⟩ { p.committed := false }
pack o as T  ≡
  assert o ≠ null ∧ o.inv = Super(T) ;
  assert (∀ object p • p.owner = ⟨o, T⟩ ⇒ p.inv = typeof(p) ∧ ¬p.committed ) ;
  assert Inv_T(o) ;
  #foreach object p such that p.owner = ⟨o, T⟩ { p.committed := true }
  o.inv := T
```

Fig. 3. Pseudo code for **unpack** and **pack**.

To simplify the specification of aggregate objects, we allow the use of the modifier **rep**. Applied to the declaration of a field f in class T, it gives rise to the *implicit object invariant* $f \neq \mathbf{null} \Rightarrow f.owner = \langle \mathbf{this}, T\rangle$. This keyword also allows us to prescribe syntactic checking of admissible model fields, as we shall see in Sec. 5.1.

Static verification. The proof rules of the Boogie methodology are formulated in terms of assertions, which cause the program execution to abort if evaluated to $false$. Assertions appear in the following places: (a) before method calls for the requires clauses of the called method, (b) at the end of a method body for the method's ensures and modifies clauses, (c) in the pseudo code for $\mathbf{unpack}$ and $\mathbf{pack}$, and (d) before field updates. Proving the correctness of a program amounts to statically verifying that the program does not abort due to a violated assertion. To do that, each assertion is turned into a proof obligation. One can then use an appropriate program logic to show that the assertions hold. All of the proof obligations can be generated and shown modularly. That is, a class C can be verified based on the specifications of the classes used by C, but without knowing the complete program in which C will be used.

For the proof, one may assume that the program invariants J1 and J2 hold. This assumption is justified by a soundness theorem for the Boogie methodology presented in earlier work [17, 21].

5 Model fields

In this section, we present the technical details of our methodology. We define which model field declarations are admissible, present a novel encoding of model fields that builds on the validity principle and enables decoupling, discuss how frame properties are specified and proved in our methodology, and prove soundness. In the following, we assume a programming language similar to the sequential subset of Java.

5.1 Declaration of model fields

The declaration of a model field m has the following form:

$$\mathbf{model}\ T\ m\ \mathbf{constrained\_by}\ E\ ;$$

where T is the type of the model field. The expression E specifies a constraint for $\mathbf{this}.m$. It is a boolean expression of the programming language, which is also allowed to mention model fields. For simplicity, we disallow method calls in model field constraints, but an extension is possible.

A model field constraint may specify a unique value for the model field, as for instance shown in Fig. 1. It is also allowed to underspecify the value of a model field, which is useful to express abstraction relations and to constrain model fields in abstract classes and interfaces. For instance, an abstract superclass $Shape$ of $Rectangle$ might constrain $width$ by $0 \leqslant width$.

A subclass can strengthen the constraint for an inherited model field by giving a declaration of the above form that repeats the name of the model field and supplies a further constraint. The *effective constraint* of a model field m in type T is the conjunction of the constraints for m in T and T's supertypes.

The mutable dependent principle (see Sec. 3) limits what fields can be mentioned in the constraint for a model field. The admissible model fields are summarized by the following definition.

Definition 1. *A model field m declared in type T is admissible if the constraint given in m's declaration typechecks according to the rules of the programming language and if each of the field access expressions in the constraint has one of the following forms:*

1. $\mathbf{this}.m$
2. $\mathbf{this}.f$*, where* f *is a concrete field*
3. $\mathbf{this}.p.f$*, where* p *is a concrete rep field and* f *is a model or concrete field*

The fields f and p must not be one of the predefined fields inv and $committed$ (but we allow f to be $owner$).

Field accesses of Form 2 occur when the constraint for a model field refers to concrete fields of the same object, for instance, in the constraint for $width$ (Fig. 1). The standard type rules require f to be declared in T or a superclass of T. That is, the constraint is allowed to refer to inherited fields. The requirement that f be concrete is not strictly necessary, but simplifies the formalization; dependencies between different model fields of the same object could be permitted as long as they are not cyclic.

Field accesses of Form 3 are used for aggregate objects, for instance, in the constraint for $maxChars$ (Fig. 2). The requirement that p be a rep field together with the implicit object invariant for rep fields guarantees that the object referenced by $\mathbf{this}.p$ is owned by $\langle \mathbf{this}, T \rangle$ when $\mathbf{this}$ is valid for T. It is imposed to adhere to the mutable dependent principle. The field p is allowed to be an inherited field.

5.2 Encoding and automatic updates of model fields

Following the validity principle explained in Sec. 3, our methodology guarantees that a model field $o.m$ satisfies the effective constraint for m in a class T if o is valid for T. That is, the following property is a program invariant:

$$\text{J3: } (\forall\, o, T, m \bullet\ o.inv <: T \ \Rightarrow\ R_m^T(o, o.m)\ )$$

$R_m^T(o, r)$ denotes the effective constraint for m in T, where $\mathbf{this}.m$ is replaced by r and $\mathbf{this}$ is then replaced by o. For instance, $R_{width}^{Rectangle}(o, r)$ denotes $r = o.x2 - o.x1$.

To achieve decoupling, we store the value of a model field in the heap as if it were an extra field of the class. Whenever a model field is read, that is, whenever a specification refers to a model field, the stored value is used. With the value of a model field being stored in the heap, any update of the values of a model field's dependees may cause the stored value to become out-of-date. We arrange for the stored value to be updated automatically, but we do so only at select times—eagerly updating the stored value whenever a dependee is changed would not just be inefficient and clumsy, but it would also retain the instant effect problems of using abstraction functions, that is the modularity problems of frame properties.

Specifically, we include an automatic update of a model field in the pack operation by inserting the following statements between the second and third assert statement of the pseudo code for $\mathbf{pack}\ o\ \mathbf{as}\ T$ (see Fig. 3):

```
#foreach m such that m is declared in or inherited by T {
  if ¬R_m^T(o, o.m) then
    assert ( ∃ r • R_m^T(o, r) ) ;
    o.m := choose r such that R_m^T(o, r)
  end
}
```

The automatic updates nondeterministically assign to $o.m$ any value of m's declared type that satisfies the effective constraint for m in T. If no such value exists, the assert statement will cause program execution to abort. This assertion allows us to detect unsatifiable constraints such as the $len = len + 1$ example from Sec. 2. The guard $\neg R_m^T(o, o.m)$ simply avoids updates that are not necessary.

5.3 Frame properties

As explained in Sec. 2.2, methodologies for model fields based on abstraction functions lead to difficult problems for the verification of frame properties. In our methodology, a model field behaves essentially like a concrete field that is updated automatically by pack statements. Therefore, model fields do not introduce additional complexity for the verification of frame properties. In particular, the semantics of modifies clauses used in the Boogie methodology [1] works also in the presence of model fields.

The modifies clause of a method M lists access expressions that, evaluated in the method's pre-state, give a set of locations that the method is allowed to modify. We denote this set by $mod(M)$. In addition to the locations in $mod(M)$, method M is allowed to modify fields of objects allocated during the execution of M as well as fields of objects that are committed in M's pre-state. The latter policy lets the method modify the internal representation of valid aggregate objects without explicitly mentioning these fields in the modifies clause, which enables information hiding. Clients of an aggregate object should not access the internal representation directly. Therefore, they do not have to know whether or not these fields are modified by the method. In summary, M is allowed to modify a field $o.f$ if at least one of the following conditions applies:

1. $o.f$ is contained in $mod(M)$
2. o is not allocated in the pre-state of M
3. o is committed in the pre-state of M

Note that this interpretation of modifies clauses sometimes requires hidden fields to be mentioned in modifies clauses. For instance, the modifies clause of method $ScaleH$ of class $Rectangle$ (Fig. 1) has to mention the concrete field $x2$ because **this** is allocated and uncommitted in the pre-state of the method. We do not address this information hiding problem in this paper, because existing solutions such as static data groups [16] or more coarse-grained wildcards [1] can be combined with our methodology.

In our example, method $ScaleH$ potentially modifies $x2$ and $width$. Both modifications are permitted by Case 1 because $x2$ and $width$ are mentioned in the modifies clause. We show that $height$ is not modified as follows. By program invariant J3, we have $height = y2 - y1$ in the pre-state of the method. Since $ScaleH$ does not assign

to $y1$ and $y2$, this property still holds before the pack statement. Therefore, $height$ is not updated when $\mathbf{this}$ is being packed.

Method $Reset$ of class $Legend$ (Fig. 2) potentially modifies fields of the $Rectangle$ object box by the call $box.ScaleH(0)$ as well as $maxChars$ by packing $\mathbf{this}$. Since box is a rep field and $\mathbf{this}$ is valid for $Legend$ in the pre-state of $Reset$, program invariant J1 and the implicit object invariant for rep fields imply that the object referenced by $\mathbf{this}.box$ is owned by $\langle \mathbf{this}, Legend \rangle$ and committed in the pre-state of $Reset$. Therefore, modification of its fields is permitted by Case 3. The modification of $maxChars$ is permitted by Case 1.

5.4 Soundness

As explained in Sec. 4, soundness of our methodology means that it is justified to assume certain program invariants when proving the assertions introduced by the methodology. Program invariants J1 and J2 are guaranteed by the Boogie methodology. To ensure that the proofs of these program invariants remain valid, we disallow model fields in object invariants. Our methodology is sound without this restriction, but we do not have the space to present the required soundness proof here, nor does the proof give additional insights. We now proceed with the proof of program invariant J3.

The proof runs by induction over the sequence of states of an execution of a program **P**. The induction base is trivial since there are no allocated objects in the initial program state.

For the induction step, we assume that the program invariant holds before the next statement s to be executed, and show that s preserves it by proving that the following property holds after the execution of s for any object o, type T, and model field m.

$$o.inv <: T \Rightarrow R_m^T(o, o.m) \tag{1}$$

We continue by case distinction on s. Only the statements that manipulate fields of objects are interesting; we omit all other cases for brevity.

Concrete field update. Let f be a concrete field declared in a class F and consider the effect of an update $x.f := e$. We show that if $R_m^T(o, o.m)$ contains an access expression that denotes $x.f$, then o is sufficiently unpacked: $T \not<: o.inv$ (that is, the left-hand side of implication 1 is $false$). We follow the cases of Def. 1.

Form 1: Since f is a concrete field, $R_m^T(o, o.m)$ does not refer to $x.f$ by access expressions of this form.

Form 2: $R_m^T(o, o.m)$ refers to $o.f$ and $x = o$. The precondition of the field update requires $F \not<: o.inv$. Since T is a subclass of F (otherwise the expression $o.f$ would not typecheck), we get $T \not<: o.inv$.

Form 3: $R_m^T(o, o.m)$ refers to $o.p.f$, where p is a rep field declared in a (not necessarily proper) superclass S of T, and $o.p = x$. From the precondition of the update of $x.f$ and from J2, we know that x is not committed. If o were valid for S, then J1, and the fact that p is a rep field, which translates into an implicit object invariant, gives us $o.p.owner = \langle o, S \rangle$, and therefore $o.p.committed$—a contradiction, so we conclude that o is mutable for S: $S \not<: o.inv$. Since T is a subclass of S, we have $T <: S \not<: o.inv$.

Unpack. Consider the statement $\mathbf{unpack}\ x\ \mathbf{from}\ S$. This statement changes the inv field of x as well as the $committed$ fields of objects directly owned by x, but nothing else. Since model fields must not refer to inv or $committed$ fields (see Def. 1), the value of $R_m^T(o, o.m)$ cannot be changed by the unpack statement.

If $x = o$, the value of $o.inv$ after the statement is the direct superclass of S. Thus, the value of $o.inv <: T$ might only be changed from $true$ to $false$. That is, Property 1 still holds after the unpack statement.

Pack. Consider the statement $\mathbf{pack}\ x\ \mathbf{as}\ S$. The only concrete fields that are changed by a pack statement are inv and $committed$. Since model fields must not refer to these fields, these updates do not have an effect on $R_m^T(o, o.m)$.

One way the program can abort is if the implicit object invariants for $\mathbf{rep}$ modifiers (see Sec. 4) do not hold in the pre-state of a pack statement. This behavior is independent of the automatic updates and the checking of the model field constraints. Therefore, we may assume in the rest of the proof that these invariants hold.

For $x \neq o$, we can prove, analogously to the case for Form 3 of concrete field updates, that the update of a model field $x.f$ preserves $R_m^T(o, o.m)$. Also, $o.inv$ is not changed by the pack statement. Consequently, the pack statement preserves Property 1.

For $x = o$, Property 1 holds trivially if $T \not<: S$ because the pack statement sets $o.inv$ to S. For $S <: T$, we have to consider two cases:

1. $R_m^S(o, o.m)$ holds before the pack statement. In this case, $o.m$ is not updated. Since effective constraints include the constraints of supertypes, the implication $R_m^S(o, o.m) \Rightarrow R_m^T(o, o.m)$ holds.
2. $R_m^S(o, o.m)$ does not hold before the pack statement. By the assert statement, we know that $R_m^S(o, o.m)$ is satisfiable, that is, there is a value to choose for the update of $o.m$. Consequently, the update establishes $R_m^S(o, o.m)$. Again, since effective constraints include the constraints of supertypes, we have $R_m^S(o, o.m) \Rightarrow R_m^T(o, o.m)$.

The automatic update of a model field $o.m$ establishes $R_m^T(o, o.m)$. It remains to show that the subsequent update of any other model field $o.n$ does not invalidate $R_m^T(o, o.m)$. This property follows from the fact that during the automatic updates, $R_m^T(o, o.m)$ does not depend on $o.n$. By the definition of admissible model fields (Def. 1), $R_m^T(o, o.m)$ can only mention three forms of field access expressions. Forms 1 and 2 cannot refer to $o.n$, since n is a model field distinct from m.

Form 3 could refer to $o.n$ if there was a rep field p declared in T and $o.p = o$. However, for any such p, we show that $o.p \neq o$:

(i) By the implicit invariant for rep fields, we have $o.p.owner = \langle o, T \rangle$;
(ii) By (i) and the second assert statement of $\mathbf{pack}$, we have $o.p.inv = \mathbf{typeof}(o.p)$;
(iii) By the first assert statement of $\mathbf{pack}$, we have $o.inv = S$, where S is a proper superclass of T;
(iv) By type safety, we have $\mathbf{typeof}(o) <: T$ (otherwise, $\mathbf{pack}\ o\ \mathbf{as}\ T$ would not type check);
(v) By (iii) and (iv), we have $o.inv \neq \mathbf{typeof}(o)$;
(vi) By (ii) and (v), we have $o.p \neq o$. □

6 Related Work

JML [5, 14] requires model fields to satisfy their constraints even for objects whose invariants are temporarily violated. Therefore, programmers are supposed to provide constraints that are satisfiable in all execution states. In our methodology, constraints express properties of valid objects, which makes specifications more concise. JML and ESC/Java2 [6] allow strengthening of constraints for inherited model fields, but do not enforce consistency. This can lead to unsoundness.

Breunesse and Poll [4] address the soundness problem due to unsatisfiable constraints. They propose two solutions. Like ours, their first solution requires verifiers to provide a witness to ensure that the constraint for a model field is satisfiable. However, their desugaring of model fields does not support recursive constraints, which are often useful to handle recursive data structures. Our methodology supports recursive constraints, provided that the pivot field in the recursive model field access is a rep field. Breunesse and Poll's second solution transforms model fields into parameterless pure methods (that is, methods without side effects). However, they do not show how to specify and prove frame properties in this solution.

The work closest to ours is the earlier work by Müller *et al.* [19, 20]. Like the methodology presented here, that work uses ownership to solve the modularity problem of frame properties for aggregate objects. Ownership is expressed and enforced by the Universe type system [9], which is more restrictive than the ownership encoding of the Boogie methodology. Müller *et al.*'s work encodes model fields as abstraction functions, which leads to the instant effect problem described earlier. Our methodology avoids this problem by the decoupling principle.

Leino and Nelson [15, 18] require programmers to declare explicitly which fields a model field constraint is allowed to depend on. They use these explicit dependencies for three purposes: (a) to permit methods to modify certain model fields of aggregate objects without mentioning these model fields explicitly in the modifies clause. A method is allowed to modify model fields that depend on a field listed in the modifies clause. This solves the modularity problem of frame properties for aggregate objects. (b) as an abstraction mechanism to permit methods to modify the components of aggregate objects without declaring these modifications explicitly. A method is allowed to modify all dependee fields of a model field listed in the modifies clause. (c) to determine whether the modification of a field potentially affects a model field.

Explicit dependencies are not well suited for automatic program verifiers such as ESC/Java [6, 10] and Boogie [2] because automatic theorem provers such as Simplify [7] cannot easily determine how often the recursive predicate for the (transitive) depends relation should be unfolded [8]. Our methodology avoids explicit dependencies as follows: (a) Due to the decoupling principle, model fields of aggregate objects do not change instantly when their dependees are modified. Avoiding these instant changes solves the modularity problem of frame properties. (b) We allow methods to modify fields of committed objects without mentioning these fields in the modifies clause. If an aggregate object is valid, its components are committed. (c) Again due to the decoupling principle, the modification of a field never changes the value of a model field. Whether a dependent model field $o.m$ will be updated by the next $\mathbf{pack}\ o\ \mathbf{as}\ T$ statement can be determined using the constraint for m in T.

Both Müller *et al.*'s and Leino and Nelson's work [15, 18] need a strong authenticity requirement for soundness. This requirement prevents model fields from depending on inherited fields (such as $MyRectangle$ in Sec. 2.2) and, therefore, limits the support for inheritance. Moreover, they do not allow classes to strengthen inherited constraints. By freeing model fields of mutable objects from the obligation to satisfy their constraints (validity principle) and by (un-)packing objects for each superclass of their dynamic type individually, these restrictions are not necessary in our methodology.

Other recent work use abstraction functions for model fields by exploring different encodings of the programming logic [13, 12].

In this paper, we have used ownership to adhere to the mutable dependent principle. There are extensions of the Boogie methodology that use alternatives to ownership. For example, our visibility-based approach [17] adheres to the mutable dependent principle. The *update guards* of Barnett and Naumann [3] adhere to a slightly weaker mutable dependent principle, which we could have used here instead. These extensions allow model fields to depend on non-owned state, which is useful in some implementations.

Separation logic uses new logical connectives to express that a predicate depends only on certain objects in the heap. It has been used successfully to modularly verify an invariant of a single class with a single instance [22]. Parkinson and Bierman [23] extend separation logic to an object-oriented language, introducing abstract predicate families to encapsulate an object's state. They do not show, however, how to express abstractions of aggregate objects such as the $maxChars$ field in Fig. 2. Our methodology treats model fields like ordinary fields with automatic updates, which are both handled by separation logic. Therefore, we hope that this contribution will help to improve the support for data abstraction in separation logic.

7 Conclusions

We have presented a sound and modular verification methodology for reasoning about model fields. Since our methodology supports subtyping, aggregate objects, and recursive object structures, it can be applied to realistic programs. Our methodology is significantly simpler and more expressive than previous approaches. These improvements are achieved by not making any guarantees about model fields of mutable objects (validity principle), by inserting automatic updates of model fields (decoupling principle), and by imposing an ownership structure (mutable dependent principle).

Model fields are used to express abstractions of the concrete states of objects. However, in our encoding, we have fully reduced the concept of a model field to other concepts that are well-understood and well-behaved, namely fields with (automatic) updates. Therefore, our treatment of model fields can be readily adopted by a variety of programming logics.

As future work, we plan to implement our methodology as part of the .NET program checker Boogie, which is part of the Spec# programming system [2].

Acknowledgments. We are grateful to Mike Barnett, David Cok, Ádám Darvas, Sophia Drossopoulou, Manuel Fähndrich, Bart Jacobs, Yannis Kassios, Gary Leavens, David Naumann, Arnd Poetzsch-Heffter, Wolfram Schulte, and the anonymous reviewers for helpful discussions and suggestions.

References

1. Mike Barnett, Robert DeLine, Manuel Fähndrich, K. Rustan M. Leino, and Wolfram Schulte. Verification of object-oriented programs with invariants. *JOT*, 3(6), 2004.
2. Mike Barnett, K. Rustan M. Leino, and Wolfram Schulte. The Spec# programming system: An overview. In *CASSIS*, volume 3362 of *LNCS*, pages 49–69. Springer-Verlag, 2004.
3. Mike Barnett and David Naumann. Friends need a bit more: Maintaining invariants over shared state. In *MPC*, LNCS. Springer-Verlag, 2004.
4. Cees-Bart Breunesse and Erik Poll. Verifying JML specifications with model fields. In *Formal Techniques for Java-like Programs*, pages 51–60, 2003. Tech. Rep. 408, ETH Zurich.
5. Yoonsik Cheon, Gary T. Leavens, Murali Sitaraman, and Stephen Edwards. Model variables: cleanly supporting abstraction in design by contract. *Software—Practice & Experience*, 35(6):583–599, 2005.
6. David Cok and Joseph R. Kiniry. ESC/Java2: Uniting ESC/Java and JML. In Gilles Barthe, Lilian Burdy, Marieke Huisman, Jean-Louis Lanet, and Traian Muntean, editors, *CASSIS*, volume 3362 of *LNCS*, pages 108–128. Springer-Verlag, 2004.
7. David Detlefs, Greg Nelson, and James B. Saxe. Simplify: A theorem prover for program checking. Tech. Rep. HPL-2003-148, HP Labs, July 2003.
8. David L. Detlefs, K. Rustan M. Leino, Greg Nelson, and James B. Saxe. Extended static checking. Research Report 159, Compaq Systems Research Center, December 1998.
9. Werner Dietl and Peter Müller. Universes: Lightweight ownership for JML. *JOT*, 4(8), 2005.
10. Cormac Flanagan, K. Rustan M. Leino, Mark Lillibridge, Greg Nelson, James B. Saxe, and Raymie Stata. Extended static checking for Java. In *PLDI*, volume 37, number 5 in *SIGPLAN Notices*, pages 234–245. ACM, 2002.
11. C. A. R. Hoare. Proofs of correctness of data representation. *Acta Inf.*, 1:271–281, 1972.
12. Bart Jacobs and Frank Piessens. Verifying programs using inspector methods for state abstraction. Tech. Rep. CW 432, Dept. of Comp. Sci., K. U. Leuven, December 2005.
13. Yannis T. Kassios. Dynamic frames: Support for framing, dependencies and sharing without restrictions. Tech. Rep. CSRG-528, U. of Toronto, Comp. Sys. Research Group, July 2005.
14. Gary T. Leavens, Albert L. Baker, and Clyde Ruby. Preliminary design of JML: A behavioral interface specification language for Java. Technical Report 98-06-rev28, Iowa State University, Department of Computer Science, 2003. See www.jmlspecs.org.
15. K. Rustan M. Leino. *Toward Reliable Modular Programs*. PhD thesis, California Institute of Technology, 1995.
16. K. Rustan M. Leino. Data groups: Specifying the modification of extended state. In *OOPSLA*, volume 33, number 10 in *SIGPLAN Notices*, pages 144–153. ACM, 1998.
17. K. Rustan M. Leino and Peter Müller. Object invariants in dynamic contexts. In Martin Odersky, editor, *ECOOP*, volume 3086 of *LNCS*, pages 491–516. Springer-Verlag, 2004.
18. K. Rustan M. Leino and Greg Nelson. Data abstraction and information hiding. *ACM Transactions on Programming Languages and Systems*, 24(5):491–553, September 2002.
19. Peter Müller. *Modular Specification and Verification of Object-Oriented Programs*, volume 2262 of *LNCS*. Springer-Verlag, 2002.
20. Peter Müller, Arnd Poetzsch-Heffter, and Gary T. Leavens. Modular specification of frame properties in JML. *Concurrency & Computation: Practice & Experience*, 15:117–154, 2003.
21. David Naumann and Mike Barnett. Towards imperative modules: Reasoning about invariants and sharing of mutable state. In *LICS*, pages 313–323. IEEE, 2004.
22. Peter W. O'Hearn, Hongseok Yang, and John C. Reynolds. Separation and information hiding. In *POPL*, pages 268–280. ACM, 2004.
23. Matthew Parkinson and Gavin Bierman. Separation logic and abstraction. In *POPL*, pages 247–258. ACM, 2005.