A refinement methodology for object-oriented programs

Asma Tafat, Sylvain Boulmé, Claude Marché

A Refinement Methodology for Object-Oriented Programs *

Asma Tafat[1], Sylvain Boulmé[2], and Claude Marché[3,1]

[1] Lab. de Recherche en Informatique, Univ Paris-Sud, CNRS, Orsay, F-91405
[2] Institut Polytechnique de Grenoble, VERIMAG, Gières, F-38610
[3] INRIA Saclay - Île-de-France, F-91893

Abstract. Refinement is a well-known approach for developing correct-by-construction software. It has been very successful for producing high quality code e.g., as implemented in the B tool. Yet, such refinement techniques are restricted in the sense that they forbid aliasing (and more generally sharing of data-structures), which often happens in usual programming languages.
We propose a sound approach for refinement in presence of aliases. Suitable abstractions of programs are defined by algebraic data types and the so-called model fields. These are related to concrete program data using coupling invariants. The soundness of the approach relies on methodologies for (1) controlling aliases and (2) checking side-effects, both in a modular way.

1 Introduction

Design-by-contract is a methodology for specifying programs (in particular object-oriented ones) by attaching pre- and post-conditions to functions, methods and such. In recent years, significant progress has been made in the field of deductive verification of programs, which aims at building mathematical proofs that such a program satisfies its contracts. Some widely used programming languages, like JAVA, C# or C have been equipped with formal specification languages and tools for deductive verification, e.g., JML [11] for Java, Spec# [6] for C#, ACSL [7] for C. The assertions written in the contracts are close to the syntax of the underlying programming language, and directly express properties of the variables of the program. However, for codes of large size the need for data abstractions arises, both for writing advanced specifications and for hiding implementation details.

Leavens et al. [18] have listed some specification and verification challenges for sequential object-oriented programs that still have to be addressed. One of these issues deals with data abstraction in specification, and more specifically the specification of modeling types. The task to be done is summed up as follows: *Develop a technique for formally specifying modeling types in a way that is useful for verification.*

This paper proposes to solve this problem using a refinement approach. Our proposal has strong connections with the notion of program refinement of the B method [1]

* This work is partly supported by INRIA Collaborative Research Action (ARC) "CeProMi", http://www.lri.fr/cepromi/

for developing correct-by-construction programs. In a first step, abstract views of objects are specified with so-called *model fields* as an abstract representation of their state. Unlike the standard model fields of JML, our model fields are described as *algebraic data types* instead of immutable objects. The refinement of such an abstract view is a concrete object together with a coupling invariant that connects its concrete fields with model fields of the abstract view. Like all refinement approaches, we want to ensure that reasoning on the abstract view in a client code does not allow establishing properties that are falsified at runtime. Hence, in the presence of arbitrary pointers or references (and thus data sharing), the verification of these coupling invariants requires a strict policy on assignment, for controlling where a given invariant is potentially broken.

This paper is based on the *ownership* policy of Boogie methodology [4]. In Section 3 we propose a variant of ownership to support model fields. The main result (Theorem 1) states that class invariants, including coupling invariants, are preserved during execution. Section 4 then proposes a refinement approach for object-oriented programs, where subclasses are refined programs for abstract classes. An additional ingredient needed is a technique for controlling side effects in subclasses: in this paper we use *datagroups* [22]. We illustrate the methodology on two examples: first, the *calculator* example of Morgan [23], and second, an instance of the observer pattern.

2 Preliminaries

2.1 Deductive verification of contracts

We consider object-oriented programs equipped with a *Behavioral Interface Specification Language* (BISL) such as JML [11] for Java, Spec# [6] for C#, etc. Methods are equipped with *contracts*: pre- and postconditions, frame clauses to specify write effects, etc; and objects are equipped with *class invariants*.

Our goal is to verify that a program satisfies its specification using proof methods. A general approach for that purpose is the generation of *verification conditions* (VCs), which are logical formulas whose validity implies the correctness of the program with respect to the specification. To automatize this process, a popular method is the calculus of weakest preconditions, as available in ESC/Java [14], Spec# [6], and the Why platform [17]. In a slightly different context but for similar purposes, weakest preconditions are used in the B method [1] for developing correct-by-construction programs.

The primary application of BISL is runtime assertion checking. For this reason, assertions used in annotations are boolean expressions. However, it has been noted by several authors [12, 16] that for deductive verification purposes, the language of assertions should be instead based on classical first-order logic. In particular, it allows calling SMT provers to discharge VCs. This is the setting we assume in this paper. More generally, we assume that the specification language allows user-defined algebraic datatypes, such as in B [1], ACSL [7] or Why [17].

Example 1. Multisets, or *bags*, are typically a useful algebraic datatype for specifying programs, that we need later. Here is a (partial) user-defined axiomatization of bags (See [26] for a full one)

```
type bag<X>;
constant emptybag: bag<X>;
function singleton: X -> bag<X>;
function union: bag<X>, bag<X> -> bag<X>;
function card: bag<X> -> integer;
function sumbag: bag<real> -> real;
axiom union_empty: \forall b:bag<X>, union(b,emptybag) = b;
axiom union_assoc: \forall b1,b2,b3:bag<X>,
       union(b1,union(b2,b3)) = union(union(b1,b2),b3);
...
```

2.2 Refinement

Refinement calculus [23, 2] is a program logic which promotes an incremental approach to the formal development of programs: from very abstract specifications down to implementations. The B method [1] has successfully mechanized this logic in some industrial developments [8]. In the B method, an abstract component introduces abstract variables and defines each procedure by an abstract behavior on these variables. A refined component is then given using other variables, a *coupling invariant* which relates them to abstract variables, and refined definitions of procedures. A component may be refined several times in this way, until all behaviors of procedures are given as programs.

Example 2. Morgan's calculator [23] is a typical and simple example of refinement. Such a calculator is aimed at recording a sequence of real numbers, and providing their arithmetic mean on demand. Below, on the left, is an abstract view of a calculator, whereas the right part presents a refinement expressing that two numbers are sufficient to encode the required informations on the whole sequence:

var $values$: $\mathtt{bag}(\mathbb{R})$
init $values \leftarrow \emptyset$;
op add($x : \mathbb{R}$):void =
  $values \leftarrow values \cup \{x\}$;
op mean():$\mathbb{R}$ =
  pre $values \neq \emptyset$;
  result $\leftarrow \dfrac{\mathtt{sumbag}(values)}{\mathtt{card}(values)}$;

var $count$: $\mathbb{N}$
var sum : $\mathbb{R}$
invariant $sum = \mathtt{sumbag}(values) \wedge$
  $count = \mathtt{card}(values)$;
init $sum \leftarrow 0$; $count \leftarrow 0$;
op add($x : \mathbb{R}$):void =
  $sum \leftarrow sum + x$;
  $count \leftarrow count + 1$;
op mean():$\mathbb{R}$ = **result** $\leftarrow sum/count$;

This paper investigates how to adapt this approach to reasoning on object-oriented programs. However, we consider the simpler case with only one abstract level, where behaviors are given as pre/post-conditions together with frame clauses, and one concrete level, the implementations in the underlying programming language.

Technically, refinement corresponds to the condition below, verified for each operator, where x are the input parameters, a the abstract variables, c the concrete ones, P the abstract precondition, I the coupling invariant, Q the abstract postcondition, S the body of the concrete operation: $\forall c, x, a;\ (P \wedge I) \Rightarrow \exists a'; \mathbf{wp}(S, (Q \wedge I)[a \mapsto a'])$. Let us explain this VC from client's point of view. For any reachable state c, a satisfying I in the execution of a given client code, there exists abstract values a' such that I is

still satisfied. For instance, in a client code, we can safely replace an execution of the concrete sequence S, by a non-deterministic update of variable a that chooses an arbitrary value a' satisfying both Q and I. The VC on any operation call ensures that the remaining of the client code is correct for all possible choices of this non-deterministic update.

Example 3 (Calculator continued). The VC for the add operation is

$$\begin{array}{l}\forall count, sum, values, x;\ (sum = \texttt{sumbag}(values) \wedge count = \texttt{card}(values)) \Rightarrow \\ \quad \exists values'; values' = values \cup \{x\} \wedge \\ \qquad (sum + x = \texttt{sumbag}(values') \wedge count + 1 = \texttt{card}(values'))\end{array}$$

which is a logical consequence of the axiomatization of bags (Example 1).

2.3 Model fields

Model fields have been introduced by Leino [19] as abstract representations of object states. Syntactically, a *model field* is used only for specification purpose and remains invisible from the actual code. Clients can refer to its successive values in their assertions, without knowing how this abstract state is implemented.

We adopt the JML syntax for model fields [13], but the JML *represents* clauses are replaced by coupling invariants, which are more general since they do not enforce a model field to be deterministically determined from concrete fields. Notice that model fields differ from *ghost* fields: the latter can be directly assigned in implementations.

Example 4. In the following, we declare a public view of class `Euros` to compute addition and subtraction on euros. In this public view, the model field `value` represents the state of the object as a *real* number.

```
class Euros {
  //@ model real value=0.0;
  //@ invariant this.value>=0.0;

  /*@ assigns this.value;
    @ ensures this.value==\old(this.value+a.value);  */
  void add(Euros a);
}
```

In the corresponding implementation below, the real number is coded as two integers: in particular, the fractional part of the real is coded as a byte less than 100.

```
class Euros {
  private int euros=0;
  private byte cents=0;
  //@ invariant 0 <= euros && 0 <= cents < 100;
  //@ invariant coupling: value == euros + cents / 100.0;

  void add(Euros a) {
      euros += a.euros; cents += a.cents;
      if (cents >= 100) { euros++; cents -= 100; }
  }
}
```

Giving a semantics to model fields leads to several issues [10, 13, 20] that we will discuss further in Section 5: as model fields are not directly assigned in the code, at which program points the values of model fields are changed? At which program points the coupling invariant, relating the concrete fields (like `euros` and `cents` above) to the model field (`value` above), is ensured? Also, the public view above says that only model field `value` is modified, is it sound to ignore the change on private fields (like `euros` and `cents`) in clients?

2.4 Ownership

Checking preservation of class invariants is known to be a difficult problem because of aliasing and thus sharing of references [18]. The *ownernhip* approach proposed by Barnett et. al in 2004 [4] is suitable for deductive verification, and implemented in the Boogie VC generator [5]. Informally, *ownership* views objects as boxes which can be opened or closed. A closed object ensures that its invariant is satisfied. Conversely, the contents of an object can be updated only when this object is open. The status, open or closed, of an object is represented by some specific boolean field `inv` similar to a model field (that is only accessible in specifications). Concretely, opening and closing an object is performed by using special statements `unpack` and `pack`. Hence, closing an object generates a VC that the invariant of this object holds.

Updating an object's field must not break the invariant of an other closed object. This crucial property is ensured by a strict discipline. First, the invariant of an object o can constrain only objects accessible via dedicated fields called "`rep` fields". More precisely, the invariant of o may refer to $o.f_1 \dots f_n.g$ only if $f_1, \dots, f_n$ are declared as `rep`. Hence, a `rep` field f declares that whenever o is closed, then $o.f$ must also be closed: in this case, we say that o *owns* $o.f$. Moreover, a given closed object can only have *at most one owner*. Technically, another model boolean field `committed` represents whether an object has a owner or not. This field acts as a lock that is only modified by applying `unpack` and `pack` statements to its owner. This ensures that an object can not be modified without opening its owner first.

With inheritance, this approach is generalized by transforming `inv` field into a class name: "$o.inv = C$" means that object o satisfies invariant of all superclasses of C (C included). Packing and unpack are made relative to a class name: "`pack` o `as` C" means "close the box o with respect to class C"; whereas "`unpack` o `from` C" means "open the box o out of C", i.e set its `inv` to the superclass of C.

This informal description is formalized in next section (see also [26]), together with our proposed extension adding a specific support of model fields.

3 Ownership and Model Fields

3.1 Language setting

We consider a core object-oriented language [4] extended with model fields. A hierarchy of classes is defined together with specifications. First there is a base class `Object` which contains only the two special model fields: inv denoting a class name and $committed$ denoting a boolean. Each class is given by:

- its (unique) name
- the name of its superclass, `Object` by default
- a set of model fields, whose types are logic datatypes
- a set of concrete fields, some of them might be marked as `rep`
- an invariant, that is a logical assertion syntactically limited to mention well-typed locations (according to Java static typing) of the form "this.$f_1 \dots f_n.g$" where f_i are `rep` concrete fields and g is either a model or a concrete field.
- a set of method definitions that consists of a profile "$\tau\ m(x_1 : \tau_1, \dots, x_n : \tau_n)$", a body, and a *contract* defined as:
  - a pre-condition $Pre_m(this, x_1, \dots, x_n)$
  - a post-condition, $Post_m(this, x_1, \dots, x_n, result)$ which might refer to the pre-state using old and to the return value using $result$
  - a frame clause $Assigns(locs)$ specifying the side-effects: it states that any memory locations, allocated in the pre-state, that do not belongs to $locs$, is unchanged in the post-state.
- a set of constructors with a profile $C(x_1 : \tau_1, \dots, x_n : \tau_n)$, a body, and a *contract* similar to those of methods, except that precondition cannot refer to $this$ and post-condition cannot not refer to result, but can refer to $this$ to denote the constructed object.

Pre- and postconditions must be purely logic expressions, in particular we forbid constructor or method calls in them. A class inherits fields of its superclass, in particular it has an inv and a $committed$ field. We denote by $<:$ reflexive-transitive closure of subclass relation. We denote by $Comp_T$ the set of rep fields declared in class T. More precisely, $Comp_T$ contains only rep fields declared in T but not the rep fields declared in a strict superclass of T. A field update $o.f := E$ where f is a concrete field declared in superclass T of o static type, has the precondition $\neg(o.inv <: T)$, meaning that $o.inv$ must be a strict superclass of T. Field update $o.f := E$ where f is a model field is syntactically forbidden. Using `pack` (see below) is the only way to update model fields. Bodies of methods are verified in a context where $type(this)$ is the current class: inherited methods are rechecked according to the context of the subclass.

3.2 pack/unpack for model fields

We define two statements for opening and closing object. Opening an object o is done via the following statement, whose semantics is given by the contract:

unpack o **from** T :
pre: $o \neq null \wedge o.inv = T \wedge \neg o.committed$
assigns: $o.inv, o.f.committed \mid f \in Comp_T$
post: $o.inv = S \wedge \bigwedge_{f \in Comp_T} o.f.committed = false$

where T is a class identifier (using $type(o)$ instead of T is forbidden, hence $Comp_T$ is statically known by VC generator), and S is the direct superclass of T.

The **pack** statement is significantly more complex than the original in Boogie's ownership, because it performs a non-deterministic update of model fields. We adopt here a syntax inspired by unbound choice operator of B:

pack o **as** T **with** $M_0 := v_0, \ldots, M_n := v_n$ **such that** P

where o is the object to close, M_i is a model field to update, v_i is a fresh variable denoting the desired new value for $o.M_i$, and P is a proposition which can mention both v_i and the current values of the model fields or the concrete fields. Syntactically, T is a class identifier and M_i must belong to model fields declared in T (updating model fields of a superclass is forbidden). The semantics is given by the contract:

pack o **as** T **with** $M_0 := v_0, \ldots M_n := v_n$ **such that** P :
pre: $o \neq null \wedge o.inv = S \wedge$
$\exists v_0, \ldots, v_n,\ Inv_T[this.M_i \mapsto v_i][this \mapsto o] \wedge P \wedge$
$\bigwedge_{f \in Comp_T} o.f = null \vee (o.f.inv = type(o.f) \wedge \neg o.f.committed)$
assigns: $o.M_0, \ldots, o.M_n, o.inv, o.f.committed \mid f \in Comp_T$
post: $o.inv = T \wedge Inv_T[this \mapsto o] \wedge (\textbf{old}(P))[v_i \mapsto o.M_i] \wedge$
$\bigwedge_{f \in Comp_T} o.f \neq null \Rightarrow o.f.committed$

where S is the superclass of T, $type(e)$ denotes the dynamic type of expression e and $Inv_T[this.M_i \mapsto v_i][this \mapsto o]$ is the coupling invariant in which model fields M_i mentioned in the clause **with** are substituted by v_i.

Example 5. Figure 1 is a variant of Morgan's calculator equipped with pack/unpack statements and pre- and postconditions to state the values of `inv` and `committed` fields. The VC generated from the precondition of pack statement in method `add` is:

$$\begin{array}{l} this \neq null \wedge this.inv = Object \wedge \\ \exists v,\ this.sum = sumbag(v) \wedge this.count = card(v) \wedge \\ \qquad v = union(this.values, singleton(x)) \end{array}$$

Hence, notice that the weakest precondition of `add` is thus very similar formula to the VC of the refinement given in Example 3.

3.3 Invariant preservation

We state below our main result. The first proposition means that committed objects must be fully packed. The second states the most important property: invariants are valid for packed objects. The third states that components of a closed object are committed. The fourth expresses that a committed component can have only one owner.

Theorem 1 (invariant preservation). *The following properties hold during any program execution.*

$$\forall o; o.committed \Rightarrow o.inv = \textbf{\textit{type}}(o) \quad (1)$$
$$\forall o, T; o.inv <: T \Rightarrow Inv_T(o) \quad (2)$$
$$\forall o, T; o.inv <: T \Rightarrow \textstyle\bigwedge_{f \in Comp_T} o.f = null \vee o.f.committed \quad (3)$$
$$\forall o, T, o', T'; \textstyle\bigwedge_{f \in Comp_T, f' \in Comp_{T'}} (o.inv <: T \wedge o'.inv <: T' \wedge o.f \neq null \wedge o.f = o'.f') \Rightarrow (o = o' \wedge T = T') \quad (4)$$

where quantifications over references range over allocated objects.

See [26] for the proof. It is similar to the one of [4]. Differences come from the presence of model fields, coupling invariants and our extended pack statement.

```
class SimpleCalc {
  //@ model bag<real> values;
  private int count;
  private double sum;
  //@ invariant sum==sumbag(values) && count==card(values);

  /*@ assigns \nothing;
    @ ensures inv==\type(this) && !committed
    @         && values == empty_bag; */
  SimpleCalc() {
      sum = 0.0; count = 0;
      /*@ pack this \as SimpleCalc \with values:=v
        @    \such_that v==empty_bag; */
  }

  /*@ requires inv==\type(this) && !committed;
    @ assigns values, count, sum;
    @ ensures values==union(\old(values),singleton(x)); */
  void add(double x) {
      //@ unpack this \from SimpleCalc;
      sum += x; count++;
      /*@ pack this \as SimpleCalc \with values := v
        @   \such_that v == union(values,singleton(x)); */
  }

  /*@ requires inv==\type(this) && values != empty_bag;
    @ assigns \nothing;
    @ ensures \result==sum_bag(values)/card(values); */
  double mean() { return sum/count; }
}
```

Fig. 1. Morgan's calculator with pack/unpack

4 A refinement methodology

We have a notion of model fields with a proper nondeterministic semantics, similar to abstract variables as they are used in the B method. To go further, we now describe a methodology for the development of OO programs which mimics the refinement approach. This methodology is simply a combination of our notion of model fields with datagroups as proposed by [19, 22]. We introduce this methodology below on Morgan's Calculator before considering a more complex example.

4.1 Hiding effects using datagroups in assigns clauses

Let us consider Morgan's Calculator of Example 2. We would like to mimic this example in Java by splitting class `SimpleCalc` of Fig. 1 into two classes: first, an abstract class `Calc` (Fig. 2) mentioning only the model field and contracts for methods; second,

```
abstract class Calc {
  //@ datagroup Gvalues;
  //@ model bag<real> values \in Gvalues;

  /*@ requires this.inv == \type(this) && !this.committed;
    @ assigns Gvalues;
    @ ensures values == union(\old(this.values),singleton(x));
    @*/
  abstract void add(double x);

  /*@ requires inv == \type(this) && values != empty_bag;
    @ assigns \nothing;
    @ ensures \result == sum_bag(values)/card(values); */
  abstract double mean();
}
```

Fig. 2. Morgan's Calculator, abstract class

```
class SmartCalc extends Calc {
  private int count;  //@ \in Gvalues;
  private double sum; //@ \in Gvalues;
  /*@ invariant this.sum == sumbag(this.values)
    @  && this.count == card(this.values); */

  /*@ assigns \nothing;
    @ ensures this.values == empty_bag;
    @ ensures this.inv == \type(this) && !this.committed; */
  SmartCalc() {
    sum = 0.0; count = 0;
    /*@ pack this \as Calc \with values:=c
      @    \such_that c == empty_bag;
      @ pack this \as SmartCalc;     */
  }

  void add(double x) {
    //@ unpack this \from SmartCalc;
    //@ unpack this \from Calc;
    sum += x; count++;
    /*@ pack this \as Calc \with values:=c
      @    \such_that c == union(values,singleton(x));
      @ pack this \as SmartCalc;     */
  }

  double mean() { return sum/count; }
}
```

Fig. 3. Morgan's Calculator, implementation class

an implementation `SmartCalc` (Fig. 3) using concrete fields `count` and `sum`. Two successive `unpack` or `pack` statements are needed to open or close an object from class `SmartCalc` to `Calc` then to `Object`. A key issue arises here, about the specification of side effects: the abstract class is not supposed to mention `count` and `sum` in `assigns` clauses, since those fields are not even known.

In the B method [1], a simple encapsulation mechanism of private fields ensures that their modifications can not be observed from clients. Hence, in B, it is safe to simply ignore modifications on private fields in clients, since clients cannot access them. Unfortunately, such a simple approach is not sound for OO programs. Indeed, a given object can be indirectly a client of itself via a reentrant call, and observes modifications made by this reentrant call on its own private fields. Actually such a problem would also occur in B, if mutual recursion between components was allowed.

In presence of reentrancy, we can not ignore modifications on private fields. Alternatively, [19, 22] proposes to *abstract* such modifications using *datagroups*. We use this approach in this paper since it smoothly integrates into any VC generator using classical logic (see Section 5 for further discussion). Roughly, a datagroup is a name for a set of memory locations and used in `assigns` clauses to express that all its memory locations may have been modified. The main feature of datagroups is that they can be extended in subclasses with new fields (public or private). The inclusion of a field to a datagroups must appear in the declaration of that field and is defined all over its scope. Datagroups may also include other datagroups (hence, we may have nested datagroups) and a field may belong to several datagroups.

Hence, coming back to Morgan's calculator, we introduce a datagroup called `Gvalues` that consists of model field `values` in abstract class `Calc` of Fig. 2, and which is extended with concrete fields `count` and `sum` in its implementation `SmartCalc` of Fig. 3. Of course, on this example, it would be more user-friendly to identify syntactically the datagroup `Gvalues` and the model field `values`. However, in this paper, we prefer to keep a clear distinction between the two notions, since in other examples, a datagroup may contain several model fields.

4.2 Modular Reasoning on Shared State: the Observer Pattern Example

In the literature (see for instance [24]), ownership discipline is often considered as incompatible with modular reasoning on a shared state between objects. Indeed, at first sight, ownership discipline forbids objects constraining *simultaneously* a given substate through an invariant. A contribution of our work is to show that this common belief is wrong. Ownership extended with nondeterministic refinement of model fields allows some modular reasoning on a *shared state* between objects.

We illustrate this claim on *observer pattern*, a generic implementation of *event programming* in OO languages. In this pattern, an object, called Subject, maintains a list of its dependents, called observers, and notifies them automatically of any state changes, by calling their `notify` methods. When notified, observers updates their own state according to the new state of Subject, usually by calling back some accessor of Subject. Hence, Subject is shared between observers. Moreover, observers are themselves shared between Subject and some clients of the whole pattern.

Here, we instantiate this pattern to define observers of a Morgan's calculator (example fully detailed in [26]). The key idea, that makes this example work with ownership discipline, is the following: *in observers, we clone an abstraction of their shared state using model fields* (below `size` and `mean`). Thus, these clones exist only in assertions, not at runtime:

```
abstract class CalcObs {
  SubjectCalc sub;

  //@ datagroup Gsubject;
  //@ model int size \in Gsubject;
  //@ model real mean \in Gsubject;

  /*@ requires this.inv == \type(this) && !this.committed;
    @ requires sub != null && sub.mc != null
    @          && sub.mc.inv==\type(sub.mc);
    @ assigns this.Gsubject;
    @ ensures size == card(sub.mc.values)
    @         && size*mean == sumbag(sub.mc.values);
    @*/
  abstract void notify();
}
```

A given object (here Subject) glues the actual shared state with its clones through an invariant. Here is an excerpt of its specification, where the important part is the `observers_notified` invariant:

```
class SubjectCalc {
  int obs_nb;
  rep CalcObs[] obs;
  //@ invariant obs_size: obs != null && 0<=obs_nb<obs.length;

  rep Calc mc;
  /*@ invariant observers_notified: mc != null &&
    @   \forall integer i; 0 <= i < obs_nb ==>
    @      obs[i] != null && obs[i].sub == this
    @       && obs[i].size == card(mc.values)
    @       && obs[i].size*obs[i].mean == sumbag(mc.values); */

  /*@ requires inv == \type(this) && !committed;
    @ assigns obs[0..obs_nb-1].Gsubject, mc.Gvalues ;
    @ ensures mc.values==union(\old(mc.values),singleton(x)); */
  void update(double x){
      //@ unpack this \from SubjectCalc;
      mc.add(x) ;
      for (int i = 0; i < obs_nb; i++) obs[i].notify();
      //@ pack this \as SubjectCalc ;
  }

  /*@ requires inv==\type(this) && !committed ;
    @ requires o!=null && o.inv==\type(o) && !o.committed;
```

```
    @ requires o.sub==this && obs_nb < obs.length ;
    @ assigns o.committed, o.Gsubject;
    @ assigns obs_nb, this.obs[\old(this.obs_nb)];
    @ ensures o.committed;
    @ ensures this.obs_nb==\old(this.obs_nb)+1
    @       && this.obs[\old(this.obs_nb)]==o; */
  void register(CalcObs o){
      //@ unpack this \from \type(this);
      this.obs[obs_nb++]=o;
      o.notify();
      //@ pack this \as \type(this) ;
  }
}
```

The observers can then be implemented independently by refining their own clone of the shared state: they can introduce a coupling invariant relating their own actual state to the clone. For observers, the possibility to update their model fields non-deterministically is crucial here. Indeed, observers update their clone when notified by Subject which has been modified in a undetermined way from their point of view. Here is an example of such an observer:

```
class Success extends CalcObs {
  boolean passed;
  //@ invariant coupling: passed==(size>=4 && mean>=10.0) ;

  void notify(){
      //@ unpack this \from Success ;
      //@ unpack this \from CalcObs ;
      /*@ pack this \as CalcObs \with size:=s, mean:=m
        @   \such_that s==card(sub.mc.values) &&
        @              s*m==sumbag(sub.mc.values); */
      passed = (sub.size() >= 4 && sub.mean() >= 10.0);
      /*@ pack this \as Success; */
  }
}
```

In conclusion, this cloning technique through model fields offers some freedom in the design of an architecture that is both compatible with ownership discipline and that fits the particular needs of the application. However, this example reveals the need of several improvements in our approach:

- We would like a more abstract interface for Subject. First, a more abstract representation of the set of observers is desirable. Second, it would be more convenient to include all internal state of observers in one datagroup of Subject. However, the datagroups discipline (with the use of *pivot fields* [22, 26]) would then prevent access to observers from outside of Subject, which not desirable.
- This architecture would be more elegant if Subject was allowed to unpack observers: `notify` method of observers could hence be used to (re)pack them.[4]

[4] Indeed, method `register` of Subject, that registers a new observer, could be called on a open observer before to pack it via `notify`. Thus, inside their constructor, observers would not be obliged to be pack in a dummy state before the call to `register`.

However, if we want to allow a given object $\circ$ to be an unknown instance of a given class, we can not unpack $\circ$, because this would produce an uncontrolled side-effect on the committed field of $\circ$ rep fields (which are not fully known).

5 Conclusions, Related Works and Perspectives

In 2003, Cheon et al. [13] propose foundations for the model fields in JML, which are presented as a way to achieve abstraction. Their main concern is the runtime assertion checker of JML, hence they naturally propose that model fields are Java objects as any other field (although immutable objects for obvious reasons), and not logical datatypes. Moreover, a model field is related to concrete fields by a *represents* clause which amounts to giving a function from concrete fields to the associated model field. Consequently, they cannot support non-deterministic updates of model fields as in Morgan's calculator: there is more than one bag having a given cardinal and a given sum of its elements.

In 2003, Breunesse and Poll [10] explore the possible use of model fields in the context of deductive verification instead. They also analyse the potential use of non-deterministic coupling relations via `\such_that` clauses. They propose four possible approaches. The first one, which indeed originates from Leino and Nelson [21], amounts to assume that the coupling invariant holds at any program point. This is impracticable and indeed unsound since it does not check for existence of a model. Two other approaches amount to systematically replace each predicate refering to a model field by a complex formula with proper quantifiers, these are impracticable too. The last approach replaces the model fields by an underspecified function which returns any possible value for it. In some sense it is similar to our **pack with** but clearly less flexible.

In 2006, Leino and Müller [20] proposed a technique to deal with model fields via ownership. This work was the main inspiration of ours: we wanted to remove a limitation of their approach which prevent them from dealing with Morgan's calculator. Precisely, the post-condition of their pack statement for the `add` method is just the coupling invariant

$$this.sum = sumbag(this.values) \wedge this.count = card(this.values)$$

from which it is not possible to prove the postcondition

$$this.values = union(old(this.values), singleton(x))$$

because the latter is not the only bag b which have the given sum and cardinal. In other words, Leino-Müller approach [20] can only deal with deterministic coupling invariants, which impose only one possible value for model field from the values of the concrete fields.

Our methodology for refinement has a few originalities: unlike previous approaches, it allows non-deterministic refinement, as it exists classically in refinement paradigm; it permits to safely hide the side-effects on private data from the public specification of classes, which is a very important property for modularity of reasoning on programs.

More recently, the Jahob verification system [29] also uses algebraic data types to model programs. However, again the relation from concrete data to abstract is done by

logic functions, hence as previous approaches they are deterministic and not amenable to refinement in general.

On the other way around, there have been attempts to apply ownership systems to refinement-based techniques as in B. Boulmé and Potet [9] have shown that the ownership policy of Boogie is a strict generalization of the verification of invariants in B. More precisely, they have encoded the component language of B (without refinement) in a pseudo-Boogie language, and have shown that the VCs induced by this encoding imply those of B. Moreover, syntactic restrictions of B that limit data-sharing between components can be safely relaxed using a Boogie approach. However they have only considered B without refinement. By extending their encoding using a **pack with** statement, we can also derive the VCs of B for a subset of B limited at one level of refinement. However, extending this to several levels of refinements is not obvious.

Our refinement methodology combines modular techniques for (1) ensuring invariant preservation (ownership) and (2) checking side effects. Although such a combination was already said possible in the past [20], it seems strange that to the best of our knowledge, no tool currently propose both, e.g., Spec# has ownership but no datagroups, whereas ESC/Java2 has datagroups but no ownership.

Datagroups provide quite a simple technique to check side-effects, in particular because it naturally fits in a standard weakest precondition calculus in classical first-order logic. It is clearly interesting to investigate more recent approaches like *separation logic* [25], *dynamic frames*, or region-based access control [27, 28, 3].

In this paper we choose that model fields are algebraic data types because it is handy for deductive verification. However our refinement technique is certainly usable with immutable objets as models, more suitable for runtime verification; such as by approaches of Darvas [15] which map model classes to algebraic theories.

Acknowledgments We thank Marie-Laure Potet, Wendi Urribarri, Christine Paulin and others CeProMi members for their fruitful discussions on this work.

References

1. J.-R. Abrial. *The B-Book, assigning programs to meaning*. Cambridge University Press, 1996.
2. R.-J. Back and J. von Wright. *Refinement Calculus: A Systematic Introduction*. Springer-Verlag, 1998.
3. A. Banerjee, D. A. Naumann, and S. Rosenberg. Regional logic for local reasoning about global invariants. In *European Conference on Object-Oriented Programming (ECOOP'08)*, Paphos, Cyprus, July 2008.
4. M. Barnett, R. DeLine, M. Fähndrich, K. R. M. Leino, and W. Schulte. Verification of object-oriented programs with invariants. *Journal of Object Technology*, 3(6):27–56, June 2004.
5. M. Barnett, R. DeLine, B. Jacobs, B.-Y. E. Chang, and K. R. M. Leino. Boogie: A Modular Reusable Verifier for Object-Oriented Programs. In *FMCO'05*, volume 4111 of *LNCS*, pages 364–387, 2005.
6. M. Barnett, K. R. M. Leino, and W. Schulte. The Spec# Programming System: An Overview. In *Construction and Analysis of Safe, Secure, and Interoperable Smart Devices (CASSIS'04)*, volume 3362 of *LNCS*, pages 49–69. Springer, 2004.

7. P. Baudin, J.-C. Filliâtre, C. Marché, B. Monate, Y. Moy, and V. Prevosto. *ACSL: ANSI/ISO C Specification Language*, 2008. `http://frama-c.cea.fr/acsl.html`.
8. P. Behm, P. Benoit, A. Faivre, and J.-M. Meynadier. Météor: A successful application of B in a large project. In *Formal Methods'99*, volume 1708 of *LNCS*, pages 348–387. Springer, Sept. 1999.
9. S. Boulmé and M.-L. Potet. Interpreting invariant composition in the B method using the Spec# ownership relation: a way to explain and relax B restrictions. In J. Julliand and O. Kouchnarenko, editors, *B 2007*, volume 4355 of *LNCS*. Springer, 2007.
10. C.-B. Breunesse and E. Poll. Verifying JML specifications with model fields. In *FTfJP'03*, 2003.
11. L. Burdy, Y. Cheon, D. Cok, M. Ernst, J. Kiniry, G. T. Leavens, K. R. M. Leino, and E. Poll. An overview of JML tools and applications. *International Journal on Software Tools for Technology Transfer*, 2004.
12. J. Charles. Adding native specifications to JML. In *FTfJP'06*, 2006.
13. Y. Cheon, G. Leavens, M. Sitaraman, and S. Edwards. Model variables: cleanly supporting abstraction in design by contract. *Softw. Pract. Exper.*, 35(6):583–599, 2005.
14. D. R. Cok and J. Kiniry. ESC/Java2: Uniting ESC/Java and JML. In G. Barthe, L. Burdy, M. Huisman, J.-L. Lanet, and T. Muntean, editors, *CASSIS*, volume 3362 of *LNCS*, pages 108–128. Springer, 2004.
15. A. P. Darvas. *Reasoning About Data Abstraction in Contract Languages*. PhD thesis, ETH Zurich, 2009.
16. J.-C. Filliâtre and C. Marché. Multi-prover verification of C programs. In *ICFEM'04*, volume 3308 of *LNCS*, pages 15–29. Springer, 2004.
17. J.-C. Filliâtre and C. Marché. The Why/Krakatoa/Caduceus platform for deductive program verification. In *CAV'07*, volume 4590 of *LNCS*, pages 173–177, Berlin, Germany, July 2007. Springer.
18. G. T. Leavens, K. R. M. Leino, and P. Müller. Specification and verification challenges for sequential object-oriented programs. *Formal Aspects of Computing*, 2007.
19. K. R. M. Leino. Data groups: Specifying the modification of extended state. In *OOPSLA'98*, pages 144–153, 1998.
20. K. R. M. Leino and P. Müller. A verification methodology for model fields. In *ESOP'06*, volume 3924 of *LNCS*, pages 115–130. Springer, 2006.
21. K. R. M. Leino and G. Nelson. Data abstraction and information hiding. *ACM Trans. Prog. Lang. Syst.*, 24(5):491–553, 2002.
22. K. R. M. Leino, A. Poetzsch-Heffter, and Y. Zhou. Using data groups to specify and check side effects. In *PLDI'02*. ACM, 2002.
23. C. Morgan. *Programming from specifications (2nd ed.)*. Prentice Hall International (UK) Ltd., 1994.
24. M. Parkinson. Class invariants: The end of the road? In *IWACO'07*, 2007. `http://www.cs.purdue.edu/homes/wrigstad/iwaco/`.
25. J. C. Reynolds. Separation logic: a logic for shared mutable data structures. In *17h Annual IEEE Symposium on Logic in Computer Science*. IEEE Comp. Soc. Press, 2002.
26. A. Tafat, S. Boulmé, and C. Marché. A refinement approach for correct-by-construction object-oriented programs. Technical Report RR-7310, INRIA, 2010.
27. J.-P. Talpin and P. Jouvelot. Polymorphic type, region and effect inference. *Journal of Functional Programming*, 2(3):245–271, 1992.
28. M. Tofte and J.-P. Talpin. Region-based memory management. *Information and Computation*, 132(2):109–176, 1997. Academic Press.
29. K. Zee, V. Kuncak, and M. Rinard. Full functional verification of linked data structures. In *PLDI'08*, pages 349–361. ACM Press, 2008.