A Refinement Approach for Correct-by-Construction Object-Oriented Programs

Asma Tafat, Sylvain Boulmé, Claude Marché

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

A Refinement Approach for Correct-by-Construction Object-Oriented Programs

Asma Tafat — Sylvain Boulmé — Claude Marché

N° 7310

June 2010

ISSN 0249-6399 ISRN INRIA/RR--7310--FR+ENG

INSTITUT NATIONAL
DE RECHERCHE
EN INFORMATIQUE
ET EN AUTOMATIQUE

INRIA

centre de recherche
SACLAY - ÎLE-DE-FRANCE

A Refinement Approach for Correct-by-Construction Object-Oriented Programs

Asma Tafat* † , Sylvain Boulmé‡ , Claude Marché†*

Thème : Programmation, vérification et preuves
Équipes-Projets PROVAL

Rapport de recherche n° 7310 — June 2010 — 31 pages

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 language such as Java and C.

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.

Key-words: Formal Specification, Deductive verification, Data invariants, Abstraction, Refinement

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

* Lab. de Recherche en Informatique, Univ Paris-Sud, CNRS, Orsay, F-91405
† INRIA Saclay - Île-de-France, F-91893
‡ Institut Polytechnique de Grenoble, VERIMAG, Gières, F-38610

Centre de recherche INRIA Saclay – Île-de-France
Parc Orsay Université
4, rue Jacques Monod, 91893 ORSAY Cedex
Téléphone : +33 1 72 92 59 00

Une approche par raffinement pour le développement correct de programmes orientés objets

Résumé : Le concept de raffinement, notion importante des méthodes formelles, permet le développement de logiciels corrects par construction. Il a été utilisé, avec succès, pour la production de code de haute qualité, par exemple, tel qu'il est implémenté dans l'atelier B. Néanmoins, de telles techniques de raffinement sont restreintes dans le sens où elles interdisent *l'aliasing*, et plus généralement le partage des structures de données, qui apparaît fréquemment dans les langages de programmation classiques comme C ou Java.

On propose une approche sûre pour le raffinement en présence de partage. Des abstractions convenables sont définies par des types de données algébriques et ce qu'on appelle des *champs modèles*. Ces derniers sont reliés aux données concrêtes à l'aide d'un invariant de collage. La sûreté de notre approche s'appuie sur des méthodes pour (1) le contrôle du partage et ; (2) la gestion des effets de bord ; dans les deux cas de façon modulaire.

Mots-clés : Spécification formelle, Vérification par preuve, Invariants de données, Abstraction, Raffinement

Contents

1 Introduction **5**

2 Preliminaries **5**
2.1 Deductive verification of contracts 5
2.2 Refinement 6
2.3 Model fields 8
2.4 Ownership 9

3 Ownership and Model Fields **12**
3.1 Language setting 12
3.2 pack/unpack for model fields 13
3.3 Invariant preservation 13

4 A refinement methodology **16**
4.1 Hiding effects using datagroups in assigns clauses 17
4.1.1 Notion of datagroup 18
4.1.2 Pivot fields 19
4.2 Modular Reasoning on Shared States: the Observer Pattern Example . 19
4.2.1 Specification of Subject 20
4.2.2 Specification of observers 23
4.2.3 An example of observer 23
4.2.4 Concluding remarks 25
4.3 Completely hidden Side Effects: a Memoization Example 25
4.3.1 Concluding remark 27

5 Conclusions, Related Works and Perspectives **28**

List of Figures

1 Axiomatization of bags 7
2 Public view of `Euros` class 8
3 An implementation of `Euros` class 9
4 An invariant temporarily broken 10
5 Violation of a class invariant by aliasing 10
6 Car example with `rep` field and `pack` and `unpack` statements . . . 11
7 Morgan's calculator with pack/unpack 14
8 Morgan's Calculator, abstract class 16
9 Morgan's Calculator, implementation class 17
10 Hiding side-effects is wrong in case of reentrancy 18
11 Datagroups and pivot fields 20
12 Specification of a Subject calculator (part 1) 21
13 Specification of a Subject calculator (part 2) 22
14 Specification of Calculator Observers 23
15 A specific observer 24
16 A client program 25
17 Fibonacci numbers using memoization 26
18 Abstract Maps and a refinement using hash tables 27

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] 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 of the programming language. 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 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 8) states that class invariants, including coupling invariants, are preserved during execution. Using previous theorem, Section 4 proposes a refinement approach for object-oriented programs, where abstract classes play the role of abstract views, whereas subclasses act as refined programs. 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 three examples: first the *calculator* example of Morgan [23] (a standard example of refinement literature), second an instance of the *observer pattern* (a challenge for invariant-based approaches, see [24]) and, at last, a *memoization* example (that illustrates the use of a private and unshared subobject).

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. The main

elements we need are *contracts* attached to methods, and *class invariants* attached to objects. The main parts of contracts are pre- and postconditions, and frame clauses to specify write effects.

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. This will be used in further examples. A user-defined axiomatization of bags in given on Figure 1. Notice that this axiomatization defines multisets that are not necessarily finite. To enforce finiteness, we could declare that the only way to construct bags are by using* $\texttt{emptybag}$*,* $\texttt{singleton}$ *and* $\texttt{union}$*, that is to declare these symbols as* constructors*. Anyway, finiteness is not needed in further examples.*

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. First, we introduce an abstract view of a calculator, expressing operations in terms of the bags of elements recorded so far:*

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

```
/*@ type bag<X>;
  @
  @ constant emptybag: bag<X>;
  @ function singleton: X -> bag<X>;
  @ function union: bag<X>, bag<X> -> bag<X>;
  @ axiom union_comm:
  @   \forall b1,b2:bag<X>, union(b1,b2) = union(b2,b1);
  @ axiom union_assoc: \forall b1,b2,b3:bag<X>,
  @   union(b1,union(b2,b3)) = union(union(b1,b2),b3);
  @ axiom union_empty: \forall b:bag<X>,
  @   union(b,emptybag) = b;
  @
  @ function card: bag<X> -> integer;
  @ axiom card_empty: card(emptybag) = 0;
  @ axiom card_singleton: \forall x:X,
  @   card(singleton(x)) = 1;
  @ axiom card_union: \forall b1,b2:bag<X>,
  @   card(union(b1,b2)) = card(b1)+card(b2);
  @
  @ function sumbag: bag<real> -> real;
  @ axiom sumbag_empty: sumbag(emptybag) = 0.0;
  @ axiom sumbag_singleton: \forall x:X,
  @   sumbag(singleton(x))=x;
  @ axiom sumbag_union: \forall b1,b2:bag<X>,
  @   sumbag(union(b1,b2)) = sumbag(b1)+sumbag(b2);
  @*/
```

Figure 1: Axiomatization of bags

We can refine this as follows, expressing that two numbers are sufficient to encode the required informations on the whole sequence:

var $count : \mathbb{N}$
var $sum : \mathbb{R}$
invariant $sum = \texttt{sumbag}(values) \wedge count = \texttt{card}(values)$;
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 report investigates how to adapt this approach to reasoning on object-oriented programs. However, in this paper, we will 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, where behaviors are given as 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:

$$\forall c, x, a;\ (P \wedge I) \Rightarrow \exists a'; \mathbf{wp}(S, ((Q \wedge I)[a \mapsto a']))$$

```
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);
}
```

Figure 2: Public view of `Euros` class

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.

There is also a VC for the initialization code S of any component to ensures that I is initially established:

$$\exists a'; \mathbf{wp}(S, I[a \mapsto a'])$$

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

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

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.

Example 4 *In Figure 2, 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. In the corresponding implementation given Figure 3, the real number is coded as two integers: in particular, the fractional part of the real is coded as a byte less than 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

```
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; }
    }
}
```

Figure 3: An implementation of `Euros` class

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?

It should be noted that model fields differ from the so-called *ghost* fields. Even if *ghost* fields are also used only in annotations, ghost fields can be directly assigned at any program point (using the *set* statement in JML) instead of changing value implicitly. Moreover, they can not change non-deterministically so they are not suitable for refinement.

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 following example illustrates the issues.

Example 5 *Figure 4 shows a piece of Java code with a class `Sensor` supposed to read the number of rotations per minute from a car's wheels, and a class `Car` with a field `sensor` and a method `update` supposed to display the car's speed in some unit. The latter is specified with an invariant relating the sensor's value and the displayed speed. Notice that this invariant is temporarily broken in `update()`'s body.*

A natural idea is to consider that an object's invariant does not always hold, but only at the beginning and at the end of each method. However, this is not sufficient: an object can violate the invariant of another one, as illustrated on Figure 5, where we have an alias between s and $c.sensor$, that makes the `read()` method invocation on object s modify $c.sensor.rpm$'s value and violate `Car`'s invariant.

The *ownership* approach proposed by Barnett et. al in 2004 [4] provides a sound verification technique for invariants, suitable for deductive verification, and implemented in the Boogie VC generator [5].

Informally, the *ownership* approach 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

```
class Sensor{
    double rpm;

    public void read() {
        rpm = ... ;
    }
}

class Car {
    int displayed_speed;
    Sensor sensor;
    //@ invariant displayed_speed == round(K*sensor.rpm);

    public void update(){
        sensor.read();
        // invariant is temporarily broken
        displayed_speed = K*sensor.rpm;
        ...
    }

    Car(Sensor s) { sensor = s; update(); }
}
```

Figure 4: An invariant temporarily broken

```
s = new Sensor();
c = new Car(s);

s.read(); // violates invariant of c
```

Figure 5: Violation of a class invariant by aliasing

(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.

Preservation of invariants in presence of nested objects, that is when field are also objects, is tricky: updating an object field must not break the invariant of an other closed object, as shown by the Car example. 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 $o.f_1 \ldots f_n.g$ only if $f_1, \ldots, f_n$ 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.

```
class Sensor{
    double rpm;

    //@ requires this.inv && ! this.committed;
    public void read() {
        rpm = ... ;
    }
}

class Car {
    int displayed_speed;
    rep Sensor sensor;
    //@ invariant displayed_speed == round(K*sensor.rpm);

    //@ requires this.inv && ! this.committed;
    public void update() {
        //@ unpack this;
        sensor.read();
        displayed_speed = K*sensor.rpm;
        //@ pack this;
    }

}
```

Figure 6: Car example with `rep` field and `pack` and `unpack` statements

Example 6 (Car example continued) *We can enforce invariant preservation by using* `pack` *and* `unpack` *statements, as shown on Figure 6.*

The $sensor$ *field is a* Car*'s component, that is* $sensor$ *is owned by* Car*. In the client code of Figure 5 the call* $s.read()$ *is then forbidden, because* c *is closed (*`c.inv` *is true), and its component* $c.sensor$ *is committed.*

In the context of subclassing and 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, 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 similar Barnett et al's [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 postcondition 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.[1] 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 and constructors are verified in a context where $type(this)$ is the current class: inherited methods are rechecked according to the context of the subclass. At object allocation, bodies of the invoked constructor run in a pre-state such that $this$ denotes a fresh object, and $this.committed$ is $false$, and $this.inv$ equals to $Object$.

[1] A consequence of point (4) of theorem 8 page 14 is that rep fields introduced in distinct classes are separated. More formally, we have that for all allocated object o, if $o.inv <: T$ with $T <: S$ and $S \neq T$, then for all $f \in Comp_S$ and $g \in Comp_T$, $o.f = o.g$ implies $o.f = null$. But, nothing forbids two rep fields of $Comp_T$ to denotes the same allocated object.

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: $o.inv = T \wedge Inv_T[this \mapsto o] \wedge (\mathbf{old}(P))[v_i \mapsto o.M_i] \wedge$
$\bigwedge_{f \in Comp_T} o.f \neq null \Rightarrow o.f.committed$

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 7 *Figure 7 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:*

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

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 object. The third states that components of a closed object are committed. The fourth expresses that a committed component can have only one owner.

```
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; }
}
```

Figure 7: Morgan's calculator with pack/unpack

Theorem 8 (invariant preservation) *The following properties hold during any program execution.*

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

where quantifications over references range over allocated objects.

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

Proof. This is done by induction on execution steps. In initial state, no objects are allocated hence (1-4) trivially hold. Now, assuming that they hold in a given program state, let's prove they remain true in the next state, by cases on the executed statement. There are only 4 kinds of statements that modify the heap.

Notice that the proof follows these of [4], only the pack case differs.

Object allocation the `new` C statement produces a fresh object o with $o.committed$ set to $false$, hence establishing (1). It sets $o.inv$ to $Object$ and thus establishes (2), since $Inv_{Object}(o)$ is just $true$. Similarly, since class $Object$ has no **rep** fields, i.e. $Comp_{Object}$ is empty set, (3) and (4) are valid too.

Field update Consider an update statement of the form $x.f = v$ where f is a concrete field declared in a superclass S of o static type such that $\neg(x.inv <: S)$. Field f is different from inv and $committed$, so (1) remains true. Moreover, if f is a rep field, as $\neg(x.inv <: S)$, there is no instance of (3) and (4) with a valid antecedent that concerns location $x.f$. Hence, (3) and (4) are maintained.

The preservation of (2) is a crucial point in this proof, it is done as follows. If that statement has an effect on $Inv_T(o)$ for some o and T, it means the object invariant declared in class T contains an access expression that denotes the same location as $x.f$. Assume this access expression in the invariant has the form $o.g_0.g_1 \dots g_n.f$ for some rep fields g_0,...,g_n declared respectively in classes C_0,..., C_n. Hence, for each $0 \leq j \leq n$, $g_j \in Comp_{C_j}$. Let's call C_{n+1} the class where f is defined. The precondition of the update statement implies $\neg(x.inv <: C_{n+1})$, that is $\neg(o.g_0 \dots g_n.inv <: C_{n+1})$. We then show by downward induction on j, $0 \leq j \leq n+1$ that $\neg(o.g_0.g_1 \dots g_{j-1}.inv <: C_j)$: from $\neg(o.g_0.g_1 \dots g_j.inv <: C_{j+1})$, by static type checking, and by (1), we have $\neg o.g_0.\dots.g_j.committed$, and then the fact that $o.g_0.\dots.g_j$ is not $null$, and by (3) instantiated with $o \mapsto o.g_0.\dots.g_{j-1}$, we get $\neg(o.g_0.\dots.g_{j-1}.inv <: C_j)$. We thus get $\neg o.inv <: C_0 = T$. This shows that (2) holds for o after the update statement, hence invariant (2) is maintained by the update statement.

unpack statement (1) is maintained since this statement changes $committed$ only from $true$ to $false$ and it changes inv only for an uncommitted object. (2) and (4) are maintained, because the statement only weakens these propositions (indeed, using (3), we deduce that (4) applies only on committed fields).

For the preservation of (3): the statement **unpack** obj **from** C changes $obj.f.committed$ from $true$ to $false$ only when $f \in Comp_C$. The statement also changes $obj.inv$ to the direct superclass of C. Hence, for $f \in Comp_C$, $obj.f$ is not concerned by (3). It only remains to proved that there is no other o, T such that $o.inv <: T$ and $o.f.committed$ is changed by the statement. But from the invariant (4) *before* the unpack, we know that for such o and T, we have $obj = o \wedge C = T$.

pack statement Let's consider a statement

pack obj **as** C **with** $M_i := v_i$ **such that** P

For each $o.f.committed$ set to $true$, there is a precondition that $o.f$ is entirely valid: $o.f.inv = type(o.f)$. Since the dynamic type of an object is always a subclass of its static type, we have $type(obj) <: C$. From precondition

```
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();
}
```

Figure 8: Morgan's Calculator, abstract class

$obj.inv = S$, and (1) before the statement, we get $\neg obj.committed$. Hence, (1) is maintained.

$Inv_C(obj)$ is established by the post-condition of the pack hence (2) is maintained. The checked precondition $\exists v_0, \ldots, v_n,\ Inv_C[this.M_i \mapsto v_i][this \mapsto obj]$ guarantees that there exists at least one possible non-deterministic assignment for model fields, so that we don't get an inconsistent state.

The statement sets $obj.f.committed$ to $true$ for every non-null component $obj.f$ so (3) is also maintained.

To prove maintenance of (4), let's consider an arbitrary instance o, o', T, T' such that $o.inv <: T$ and $o'.inv <: T'$. Let $f \in Comp_T$ and $f' \in Comp_{T'}$ such that $o.f = o'.f'$ and $o.f \neq null$. There are two cases. First, if $o.f.committed$ was $false$ before the **pack with** statement, then by (3) it was not "inside" any object before, hence o can only be the object obj packed in the statement, and so is o'. Thus, (4) holds after the statement. If $o.f.committed$ was $true$ before the statement, then from the precondition of the **pack with** statement, we conclude that neither o nor o' refers to obj. Thus, (4) is maintained.

This complete the proof of Theorem 8. □

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 more complex examples.

```
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; }
}
```

Figure 9: Morgan's Calculator, implementation class

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 Figure 7 into two classes: first, an abstract class `Calc` (Figure 8) mentioning only the model field and contracts for methods; second, an implementation `SmartCalc` (Figure 9) 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 cannot be observed from clients. Hence, in B, it is safe to simply ignore modifications on private fields in clients, since clients can not 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. This is illustrated by

```
abstract class A {
    //@ assigns \nothing;
    void m();
}

class B{
    //@ assigns \nothing;
    void g(A a){ a.m();}
}

class C extends A{
    private int x;

    void m() { x++; }

    void f(B b){
        x = 0;
        b.g(this);
        //@ assert x == 0;
    }
}
```

Figure 10: Hiding side-effects is wrong in case of reentrancy

the example below. Actually such a problem would also occur in B, if mutual recursion between components was allowed.

Example 9 *Figure 10 is a toy example showing that hiding private side effects in public interface might be wrong in presence of reentrancy. The abstract class* `A` *provides a method* `m()` *which is declared to have no side-effects. Another class B has a method which calls method* `A.m()` *on some object of class* `A`*, and thus also has no side-effect.*

Then the class `C` *refines* `A` *adding a private field modified in method* `m()`*. If we assume that this side-effect is "private" and thus can be hidden, the* `assigns \nothing` *clause is valid for this method. But then for its other method* `f()`*, we would be able to prove its assertion from the assigns clause of* `B.g()`*, whereas at runtime* x *would have value* 1 *instead of* 0*.*

This shows that ignoring the side-effect on private field x *is wrong.*

4.1.1 Notion of datagroup

To our knowledge, the first approach for modular reasoning in side-effects, sound even in case of reentrancy, was proposed by Leino [19, 22] and amounts to *abstract* side-effects 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 speaking, 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.

Example 10 *Coming back to Morgan's calculator, we introduce a datagroup called* `Gvalues` *that consists of model field* `values` *in abstract class* `Calc` *of Figure 8, and which is extended with concrete fields* `count` *and* `sum` *in its implementation* `SmartCalc` *of Figure 9. 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 further examples, a datagroup will contain several model fields.*

4.1.2 Pivot fields

In the case of nested objects, that is when an object's field is itself an object like in the Car/Sensor example (Example 5), the datagroups approach requires to be able to declare inclusions between datagroups. More precisely, if o is an object with a datagroup G, and $o.f$ is itself an object with datagroup H, then we may want to specify that $o.f.H$ is a subset of $o.G$. This is done by adding to the declaration of field f an annotation of the form **maps** H **into** G. In such a case, f is called a *pivot field* [22]. Moreover, syntactic rules prevent such a pivot field to be aliased: these restrictions ensure that modifications on a shared object can not be hidden through datagroups [22].

Example 11 (Car example continued) *Figure 11 shows a variation of Example 6 where concrete fields are made private, and side effects of methods* `read()` *and* `update()` *are abstracted by datagroups. The* `sensor` *field is a pivot field: the datagroup* `sensor.Gsensor` *is included is* `Gcar`*, meaning that the assigns clause of* `update()` *may allow modification of* `sensor.rpm`*.*

4.2 Modular Reasoning on Shared States: 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 components. Indeed, at first sight, ownership discipline forbids components constraining *simultaneously* a given substate through an invariant. A main 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 components.

We illustrate this claim below on an instance of the well-known *observer pattern*, a standard way to implement *event programming*, in which 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 update their own state according to the new state of Subject, usually by calling back some accessor of Subject. Hence, the Subject is shared between observers. Moreover, observers are themselves shared between the Subject and some clients of the whole pattern.

The ownership relation is a delicate point in this example: if each observer owns Subject, then only a single observer can be closed at a time. Hence, our solution consists in making Subject to own all its observers. However, as a consequence, ownership discipline forbids observers to set an invariant constraining *directly* the state of Subject. To circumvent this restriction, *we clone, only for reasoning purpose, an abstraction of*

```
class Sensor{
    //@ datagroups GSensor;
    private double rpm; //@ \in GSensor;

    //@ assigns GSensor;
    public void read() {
        rpm = ... ;
    }

    public double get(){
        return rpm;
    }

}

class Car {
    //@ datagroups GCar;
    private int displayed_speed; //@ \in GCar;
    Sensor sensor; //@ \maps Gsensor \into GCar;
    //@ invariant displayed_speed == round(K*sensor.rpm);

    //@ assigns Gcar;
    public void update(){
        sensor.read();
        // invariant is temporarily broken
        displayed_speed = K*sensor.get();
        ...
    }

    Car(Sensor s) { sensor = s; update(); }
}
```

Figure 11: Datagroups and pivot fields

the observed state of Subject in each observer. These clones are *model fields*: thus, they exist only in assertions, not at runtime. Hence, each observer can thus freely constrain its own clone of Subject's state. Subject itself ensures that each clone in observers is consistent with the actual observed state.

4.2.1 Specification of Subject

We consider here observers of a Morgan's calculator. Hence, concretly, Subject is a wrapper of the Morgan's calculator that allows observers to monitor the mean value. It is introduced by a new class `SubjectCalc` which points to a calculator `mc` and whose specification is split between Figure 12 and Figure 13.

Here, we want to observe the behavior of a Morgan's calculator objects, we define a new class `SubjectCalc` which points to a calculator `mc` and whose specification is shown on Figure 12 and 13. We provide a wrapper of the Morgan's Calculator that allows observers to monitor the mean value. The ownership relation is a delicate point

```
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 m >= 1;
    @ requires c.inv = \type(c) && !c.committed;
    @ assigns c.committed;
    @ ensures this.inv == \type(this) && !this.committed;
    @ ensures \fresh(this.obs) && obs.length==m && obs_nb==0;
    @ ensures this.mc == c && c.committed;
    @*/
  SubjectCalc(Calc c,int m){
      obs = new CalcObs[m];
      obs_nb = 0;
      mc = c;
      //@ pack this \as SubjectCalc;
  }

  /*@ requires mc.inv == \type(mc);
    @ requires mc.values != empty_bag;
    @ assigns \nothing;
    @ ensures \result == sum_bag(mc.values)/card(mc.values);
    @*/
  double mean(){ return mc.mean(); }

  /*@ requires mc.inv == \type(mc);
    @ assigns \nothing;
    @ ensures \result==card(mc.values);
    @*/
  int size(){ return mc.size(); }

  ( (remaining on next figure)}
}
```

Figure 12: Specification of a Subject calculator (part 1)

```
/*@ requires inv == \type(this) && !committed;
  @ assigns obs[0..obs_nb-1].Gsubject;
  @ assigns mc.Gvalues ;
  @ ensures mc.values==union(\old(mc.values),singleton(x));
  @*/
void update(double x){
    //@ unpack this \from SubjectCalc;
    mc.add(x) ;
    /*@ loop invariant 0<=i<=obs_nb &&
      @   (\forall integer j; 0 <= j < i ==>
      @    && obs[j].size==card(mc.values)
      @    && obs[j].size*obs[j].mean==sumbag(mc.values);
      @ loop assigns obs[0..i-1].Gsubject ;*/
     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) ;
}
```

Figure 13: Specification of a Subject calculator (part 2)

in this example: if observers own the calculator, then only one observer can be closed at a time. Hence, our solution consists in making the calculator to own all its observers.

For simplicity, we put observers in a concrete array of bounded size. Notice that this array is a component (`rep` field). Here, we assume that our methodology is extended in order to consider that each of the element is also consider as `rep`. Hence, as explained above, Subject owns all its observers.

This class provides methods `mean` and `size` to observe the contents of the calculator. Here, we assume that class `Calc` provides a method `size` similar to method `mean` (we have omitted it in `Calc` to save space).

The other methods in this class are `register` and `update`. Method `register` allows to add a new observer `CalcObs` (detailed in the next section), passed as argument, to the set of observers. Here, method `register` assumes that the new object o is already packed: we discuss this choice Section 4.2.4. Method `update` notifies the observers that subject (calculator) has changed, after adding a double value to the calculator's real collection.

```
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();
}
```

Figure 14: Specification of Calculator Observers

Here, since we want to allow the implementation of `notify` to perform *reentrant* calls on methods `size` and `mean`, these methods must not have a too strong precondition: they must allow calls when the wrapper is itself open.

4.2.2 Specification of observers

The specification for Observers in given on Figure 14. Each observer has a public field `sub` to point to its calculator subject. This field is not *rep* and hence, can not be dereferenced in the coupling invariant of its implementation. Indeed, as we want `sub` to own `this`, `this` must not own `sub`. Thus, `CalcObs` introduces two model fields `size` and `mean` to represent the current value of the size and the mean of `sub.mc` and which can be used in coupling invariants of observers: these two fields correspond to the *clone* described above. The invariant `observers_notified` of Figure 12 formalizes the fact that observers are notified of any changes of their values in `sub.mc`.

Method `notify` is provided for Subject: as mentionned in `update`'s body, `notify` is invoked whenever the subject changes.

4.2.3 An example of observer

The observers can be refined independently by using their own clone of the shared state: in particular, they can introduce a coupling invariant relating their own actual state to the clone. For observers, the possibility to update their model field non-deterministically is crucial here. Indeed, observers updates their clone when notified by Subject which has been modified in a undetermined way from observers point of view.

Figure 15 shows a specific example of an observer, namely an observer that detects whether there are at least 4 values, such that their mean is at least 10. As explained before, `notify` updates the internal state of the observer by calling back the dedicated methods of the Subject.

Of course, `Success` must also provide a constructor. This constructor registers automatically the new observer in the Subject. As `register` of `SubjectCalc` re-

```
class Success extends CalcObs {

    boolean passed;
    //@ invariant coupling: passed==(size>=4 && mean>=10.0) ;

    /*@ requires c.inv == SubjectCalc;
      @ assigns c.obs_nb, c.obs[\old(c.obs_nb)];
      @ ensures this.inv == Success && this.committed == true;
      @ ensures this.sub == c;
      @ ensures c.obs_nb==\old(c.obs_nb)+1
      @         && c.obs[\old(c.obs_nb)]==this;
      @*/
    Success(SubjectCalc c){
        sub = c;
        /*@ pack this \as CalcObs \with size:=s
          @                       \such_that s==0;
          @*/
        passed = false ;
        //@ pack this \as Success ;
        sub.register(this);
    }

    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;
    }
}
```

Figure 15: A specific observer

quires a closed observer, we pack this new observer in a dummy state which is not related to the calculator. The link between the state of the new observer and the calculator is indeed established in `register` (which callbacks `notify` of the new observer). We discuss this choice in Section 4.2.4.

At last, Figure 16 shows a client program, which ends with an assertion that is proved as follows: from the post-condition of `update()`, we get that `c.mc.values` $= B = \{12, 9, 10, 14\}$. From the post-condition of `Success` we know that $c.obs.[0] = s$, hence from the invariant `observers_notified` for `c`, we get `s.size` $= 4$ and `s.size` $\times$ `s.mean` $= 45$. The assertion then follows form the invariant `coupling` of s. Notice how most of the reasoning is done at the level of the model fields.

```
class Test {

    Test(){
        SmartCalc calc = new SmartCalc();
        SubjectCalc c = new SubjectCalc(calc,1);
        Success s = new Success(c);

        c.update(12.0);
        c.update(9.0);
        c.update(10.0);
        c.update(14.0);
        //@ assert s.passed == true;
    }
}
```

Figure 16: A client program

4.2.4 Concluding remarks

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. For instance, it would be more convenient to include all internal state of observers in one datagroup of Subject. However, pivot fields can not allow this since we want also access to observers from outside of Subject. A more abstract representation of the set of observers is also 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.[2] However, if we want to allow a given object `o` to be an unknown instance of a given class, we can not unpack `o`, because this would produce an uncontrolled side-effect on the committed field of `o` rep fields (which are not fully known).

4.3 Completely hidden Side Effects: a Memoization Example

In some case, it happens that some method has a completely *pure* specification, that is its public interface says it has no side-effect at all, but which has some smart implementation which makes some side-effects that are in principle invisible from their clients (also called *benevolent* side effect).

Figure 17 shows an example of this situation. The first part introduces an abstract class `Fib` for computing the mathematical Fibonacci numbers, specified using a algebraic axiomatization. Whereas this function should have no visible side effects, class `Fib` introduces a datagroup `Gfibo` in order to allow implementations with hidden side effects.

[2]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`.

```
/*@ logic integer math_fib(integer n);
  @ axiom fib0: math_fib(0) == 0;
  @ axiom fib1: math_fib(1) == 1;
  @ axiom fibn: \forall integer n; n >= 2 ==>
  @     math_fib(n) == math_fib(n-1) + math_fib(n-2);
  @*/

abstract class Fib {
    //@ datagroup Gfibo;

    /*@ requires n >= 0;
      @ requires this.inv == \type(this) && !this.committed;x
      @ assigns Gfibo;
      @ ensures \result == math_fib(n);
      @*/
    abstract long fib(int n);
}

class FibMemo extends Fib {
  private /*@ rep */ HashMap<Integer,Long> memo;
                                    //@ \maps Gmap \into Gfibo;
  /*@ invariant \forall Integer x, Long y ;
    @   y == acc(memo.map,x) ==>  y.value = math_fib(x.value);
    @*/

  /*@ assigns \nothing;
    @ ensures this.inv == \type(this) && !this.committed;*/
  FibMemo () {
      //@ pack this \as Fib;
      memo = new HashMap<Integer,Long>();
      //@ pack this \as FibMemo;
  }

  long fib(int n) {
      //@ unpack this \from FibMemo;
      if (n <= 1) { return 1; }
      Long x = memo.get(n);
      if (x != null) { return x.longValue(); }
      Long y = fib(n-1) + fib(n-2);
      memo.put(n,y);
      return y.longValue();
      //@ pack this \as FibMemo;
  }
}
```

Figure 17: Fibonacci numbers using memoization

The implementation, or refinement, `FibMemo` given further uses memoization to store the results computed so far. It uses a Map, more concretely a hashtable, specified in Figure 18. Notice that giving a complete specification of maps is not a simple taks

```
/*@ type mapping<K,V>;
  @ logic <K,V> V acc(mapping<K,V> m, K key);
  @ logic <K,V> mapping<K,V> upd(mapping<K,V> m,
  @                              K key, V value);
  @ (some axioms)
  @*/

abstract class AbstractMap<K,V> {
    //@ datagroup Gmap;
    //@ model mapping<K,V> map \in Gmap;

    /*@ requires this.inv == \type(this) ;
      @ assigns \nothing;
      @ ensures \result != null ==> \result == acc(m,k) ;
      @*/
    V get(K k);

    /*@ requires k != null;
      @ requires this.inv == \type(this) && !this.committed;
      @ assigns Gmap;
      @ ensures map == upd(\old(m),k,v);
      @*/
    void put(K k, V v);
}

class Hashmap extends AbstractMap {
    (implementation using hash tables)
}
```

Figure 18: Abstract Maps and a refinement using hash tables

because of genericity, we give only a sketch of it here, and refer to Tushkanova et al. [28, 29] for a detailed discussion about specification of generic programs.

This example illustrates the use of an *unshared* private object `memo` hidden to clients of `Fib`: here, we use `memo` as a pivot field to include `memo.Gmap` into `Gfibo` datagroup.

Finally, notice that HashMap class is also a typical example of a refinement of an abstract class for maps, that requires to provide a suitable abstraction of the state to the clients. The task of specifying a concrete implementation of hashtables, with an appropriate coupling invariant, is straightforward and is left to readers.

4.3.1 Concluding remark

This example aimed to illustrate a complete hiding of side-effects to clients. However, it may be argued that this goal is not fully reached since datagroup `Gfibo` is visible from clients of class `Fib`. Let us precise here that clients are intended to let this datagroup empty, that is, without any field. Hence the verification conditions for those clients are as if method `fib` was free of side-effects. In other words, only implementations of class `Fib` that put a field into `Gfibo`, are obliged to consider that their own method `fib` produces indeed a side-effect.

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 methods 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 b which have a 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 the public specification of classes, which is a very important property for modularity of reasoning on programs.

More recently, the Jahob verification system [30] 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, using some kind of inheritance like presented here might be a way to achieve this.

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 [3].

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 [26, 27, 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)*, 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 F. S. de Boer, M. M. Bonsangue, S. Graf, and W.-P. de Roever, editors, *Formal Methods for Components and Objects: 4th International Symposium*, volume 4111 of *Lecture Notes in Computer Science*, pages 364–387, 2005.

[3] Actually, ESC/Java2 appears to support ownership-related annotations in its input syntax and type checks those, but such annotations are not used in VC generation.

[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 *Lecture Notes in Computer Science*, 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 *Lecture Notes in Computer Science*, 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 *Lecture Notes in Computer Science*. Springer, 2007.

[10] C.-B. Breunesse and E. Poll. Verifying JML specifications with model fields. In *Formal Techniques for Java-like Programs (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 *Proceedings of the 8th Workshop on Formal Techniques for Java-like Programs (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 *Lecture Notes in Computer Science*, 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 J. Davies, W. Schulte, and M. Barnett, editors, *6th International Conference on Formal Engineering Methods*, volume 3308 of *Lecture Notes in Computer Science*, pages 15–29, Seattle, WA, USA, Nov. 2004. Springer.

[17] J.-C. Filliâtre and C. Marché. The Why/Krakatoa/Caduceus platform for deductive program verification. In W. Damm and H. Hermanns, editors, *19th International Conference on Computer Aided Verification*, volume 4590 of *Lecture Notes in Computer Science*, 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*, pages 144–153, 1998.

[20] K. R. M. Leino and P. Müller. A verification methodology for model fields. In P. Sestoft, editor, *15th European Symposium on Programming (ESOP)*, volume 3924 of *Lecture Notes in Computer Science*, 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*. 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 T. Wrigstad, editor, *3rd International Workshop on Aliasing, Confinement and Ownership in Object-Oriented Programming (IWACO), in conjunction with ECOOP 2007*, Berlin, Germany, July 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] J.-P. Talpin and P. Jouvelot. Polymorphic type, region and effect inference. *Journal of Functional Programming*, 2(3):245–271, 1992.

[27] M. Tofte and J.-P. Talpin. Region-based memory management. *Information and Computation*, 132(2):109–176, 1997. Academic Press.

[28] E. Tushkanova, A. Giorgetti, C. Marché, and O. Kouchnarenko. Modular specification of Java programs. Technical Report RR-7097, INRIA, 2009.

[29] E. Tushkanova, A. Giorgetti, C. Marché, and O. Kouchnarenko. Specifying generic Java programs: two case studies. In *Selected papers presented at LDTA'2010 workshop*. ACM Press, 2010.

[30] K. Zee, V. Kuncak, and M. Rinard. Full functional verification of linked data structures. In *ACM Conf. Programming Language Design and Implementation (PLDI)*, pages 349–361, 2008.

Centre de recherche INRIA Saclay – Île-de-France
Parc Orsay Université - ZAC des Vignes
4, rue Jacques Monod - 91893 Orsay Cedex (France)

Centre de recherche INRIA Bordeaux – Sud Ouest : Domaine Universitaire - 351, cours de la Libération - 33405 Talence Cedex
Centre de recherche INRIA Grenoble – Rhône-Alpes : 655, avenue de l'Europe - 38334 Montbonnot Saint-Ismier
Centre de recherche INRIA Lille – Nord Europe : Parc Scientifique de la Haute Borne - 40, avenue Halley - 59650 Villeneuve d'Ascq
Centre de recherche INRIA Nancy – Grand Est : LORIA, Technopôle de Nancy-Brabois - Campus scientifique
615, rue du Jardin Botanique - BP 101 - 54602 Villers-lès-Nancy Cedex
Centre de recherche INRIA Paris – Rocquencourt : Domaine de Voluceau - Rocquencourt - BP 105 - 78153 Le Chesnay Cedex
Centre de recherche INRIA Rennes – Bretagne Atlantique : IRISA, Campus universitaire de Beaulieu - 35042 Rennes Cedex
Centre de recherche INRIA Sophia Antipolis – Méditerranée : 2004, route des Lucioles - BP 93 - 06902 Sophia Antipolis Cedex

Éditeur
INRIA - Domaine de Voluceau - Rocquencourt, BP 105 - 78153 Le Chesnay Cedex (France)
http://www.inria.fr
ISSN 0249-6399