Ensuring Invariant Contracts for Modules in Java

Andreas Roth and Peter H. Schmitt

Institut für Logik, Komplexität und Deduktionssysteme
Universität Karlsruhe, Germany
E-mail: {aroth, pschmitt}@ira.uka.de

Abstract. Deductive verification of object-oriented programs suffers from the lack of modularity. One of the obstacles to modular verification are invariant contracts, which classes extending a verified system could break. We introduce a concept of modules for Java and their correctness w.r.t. invariant contracts and give a theoretical criterion on attributes called module-protection. To ensure modular fulfilment of invariant contracts, attribute chains that invariants depend on must be module-protected. Finally, we show that each of four known restrictions to enforce modularity imply module-protectedness of attributes.

1 Introduction

Though deductive verification of programs in object-oriented languages like Java has made substantial progress in recent years, it still suffers from the lack of modularity. If verified systems get extended by additional classes a lot of verification work has had to be re-done. Thus, verification has not scaled up.

Roughly speaking, the reasons for this effect, which will be discussed in detail below, are aliasing and subtyping. A number of papers have been published on how to restrict aliasing and how to curb negative effects of subtyping. We present in this paper a theoretical notion of module-protected attributes which guarantees modularity of verification. We also show that each of four known restrictions or programming patterns to enforce modularity imply module-protectedness of attributes.

In this extended abstract we omit, because of lack of space, the treatment of subtyping which however fits well into the framework for modules presented here. We assume from now on that there is no subtyping (except from the trivial one, namely that all objects inherit from java.lang.Object).

The work presented here has been done in the context of KeY [1], a project that aims at the integration of formal methods into industrial software development languages and tools. The KeY system provides an easy-to-use full-featured theorem prover based on a calculus called JavaCard Dynamic Logic [2], covering the complete JavaCard standard, a non-concurrent subset of Java [9]. Though every JavaCard program can be deductively treated with KeY, practicability will be substantially improved by providing means to modularise verification. The standard specification language for KeY is UML/OCL [21], which we will use in this paper for examples, too. The examples are however transparent enough to substitute UML/OCL by a favourite specification language. As we are concerned with Java programs, we map, for the purpose of specification, every Java class to a UML class, every Java method to a UML operation, and every Java attribute to a UML attribute.

This work does only operate with concrete locations. Abstraction (using model fields, etc.) as doubtlessly required for a proper specification technique with information hiding (as available e.g. in the Java Modeling Language) is not object of this paper, but considered future work as being a second layer above the contents of this paper.

2 An Example

Figure 1 shows an example Java program similar to an example in [4]. The program provides classes Date (which does not contain an attribute for a day, for brevity reasons) and Period, the

```
class Date {
    private Month month;
    private int year;
    public Date(Month month, int year) {
        this.month=month;
        this.year=year;
    }
    public Month getMonth() { return month; }
    public int getYear() { return year; }
    public void setYear(int year) { this.year=year; }
    public boolean earlierThan(Date cmp) {
        return (getYear()<cmp.getYear())
                || (getYear()==cmp.getYear()
                    && getMonth()<=cmp.getMonth());
    }
}

class Month {
    private int val;
    public Month(int val) {setMonth(val);}
    public void setMonth(int val) {this.val=val;}
}
```

```
class Period {
    private Date start, end;
    public Period(Date start, Date end) {
        if (end.earlierThan(start))
            throw new RuntimeException();
        this.start=start;
        this.end=end;
    }
    public void setEnd(Date end) {
        if (end.earlierThan(start))
            throw new RuntimeException();
        this.end=end;
    }
    public Date getEnd() { return end; }
}
```

Fig. 1. A simple Java program

latter composed of two instances of the first. The month of a date is represented by an instance of class Month. The attribute year in Date can be set by operation setYear. Date offers a query method earlierThan to check if the stored date is considered earlier than a given one. The end date of a Period can be retrieved by getEnd and set by setEnd. If we consider (as described above) a UML diagram representing the Java program, Period can be constrained by the OCL invariant

```
inv: self.start.year<self.end.year
    or (self.start.year=self.end.year and start.month.val<=end.month.val)
```

If we assume that Date and Month are developed separately from Period, and Period is verified w.r.t. to its specification, the following problem is encountered. After adding a new class to the system:

```
class Main {
    public Period myPeriod() {
        Date jan=new Date(new Month(1), 2004);
        Date nov=new Date(new Month(11), 2004);
        Period p=new Period(jan, nov);
        jan.setYear(2005);
        return p;
    }
}
```

the contract of Period is broken because the class invariant is not satisfied after myPeriod() has been executed: After its execution a Period object exists that starts in January 2005 and ends in November 2004.

The problem is clear: Although all classes fulfil their contracts "locally", modular soundness cannot be assured. From a verification point of view the observed behaviour is not acceptable as it would become necessary to prove the preservation of invariants for *all* classes whenever new classes are added.

Let us briefly touch the question of what an experienced Java programmer would do if he knew that his classes (as Period) were reused in an unpredictable way (as in Main). There is a large set of patterns that address these problems: e.g. [4] recommends "defensive copies" before storing the start and end attributes. In general it will be enough to ensure that writable references to the start object are not exposed to instances of classes that were not developed together with Period.

We can observe another "point of attack" in this example: the object stored in the month attribute of a Date object can be manipulated in an invariant violating way. As above references to this object could be held in myPeriod and be modified there. This can be resolved by disallowing writing references to Month in Date to leak.

3 A Definition of Modules for Java

In this section we develop a basic module concept. We will use UML/OCL class diagrams to describe the static structure and assume method bodies to be implemented in Java. It will be evident that our approach can be easily adapted to other object-oriented languages. To keep matters simple we only consider *private non-static* attributes. As we have no subclasses, *protected* attributes are useless, *public* non-final attributes are discouraged anyway by all methodologies, and it is easy to take *static* attributes into account. Techniques like reflection will not be considered here.

Definition 1. *We define inductively:*

- $(T_m, E_m, \emptyset)$ *is a module if* T_m *(the member classes) and* E_m *(the exported classes) are sets of classes with* $E_m \subseteq T_m$*.*
- *If* I_m *is a set of modules then* $m = (T_m, E_m, I_m)$ *is a module if* T_m *(the member classes) and* E_m *(the exported classes) are sets of classes with* $E_m \subseteq T_m$ *and the following property holds for* m *and all modules* $m' \neq m$*:*
  If there are classes $t \in T_m$*,* $t' \in T_{m'}$ *and there is a type reference to* t' *or an expression of type* t' *in the class declaration of* t *then* $m' \in I_m$ *and* $t' \in E_{m'}$*.*
  For different modules $m \neq m'$ *we further insist that* T_m *and* $T_{m'}$ *are disjoint.* I_m *is called the set of* imported modules*.*

We will use $\mathcal{M}$ to denote sets of modules. From now on we assume that every module is part of a module system $\mathcal{M}$ which is assumed to be closed under the import relation.

Note, that the inductive definition of modules guarantees that the import relation is acyclic. There are several possible concrete syntaxes for modules. We choose to denote a module by a UML package, label exported classes by an ≪export≫ stereotype, and denote the import set by dependencies to the contained modules marked with the stereotype ≪import≫. In our settings from Sect. 2, Date and Month are developed together, but separately from the other classes, Main and Period. This leads to the module definition depicted in Fig. 2.

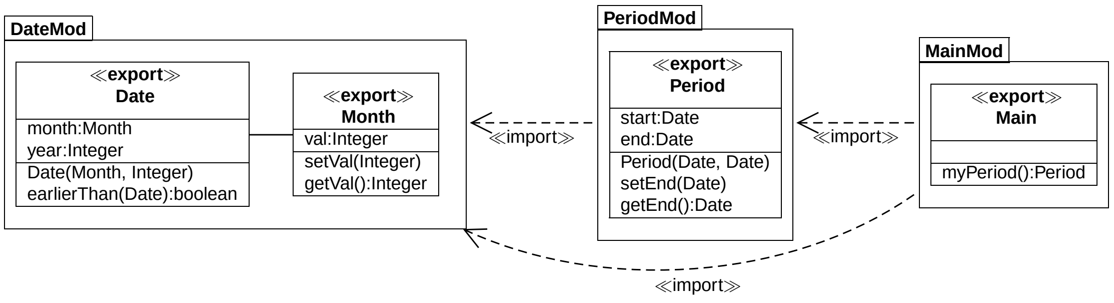

Fig. 2. A UML class diagram for the simple program, modules denoted as UML packages.

4 Module Contracts

In this paper we aim at a precise but rather informal presentation. A *state* s of a module m is determined by the sets of existing objects for each class, and the values of local variables and attributes.

Contracts for classes and interfaces are well known as restrictions on all objects of this type during their lifetime [15]. An *invariant contract* (φ, t) consists of an OCL invariant constraint φ

and the constrained class[1] t. We write $s \models \varphi$ when the OCL formula φ is true for every object of class t in state s.

Definition 2. *By Inv_t we denote the set of all invariant contracts on the class t. Inv_T extends this designation to sets T of classes. For a module $m = (T_m, E_m, I_m)$ the* module contract[2] *ct_m is given by $ct_m = Inv_{T_m}$.*

The example in Sect. 2 shows that the validity of a class invariant depends on the context in which it is considered. There, the invariant contract was fulfilled as long as only PeriodMod was considered, but got invalid when MainMod was incorporated. This necessitates the following relativised definition:

Definition 3. *Let c be a class of module m which is in a module system $\mathcal{M}$, (φ, c) be an invariant contract, and s_1 be a state of $\mathcal{M}$ with $s_1 \models \varphi$. c fulfils (φ, c) in a set of classes T if all non-private methods p of T invoked in s_1 terminate in some state s_2 with $s_2 \models \varphi$.*

Definition 4. *A module $m = (T_m, E_m, I_m)$ fulfils its module contract ct_m in a set of classes T if all classes $t \in T_m$ fulfil their invariant contracts from ct_m in T.*

Fulfilling a module contract *in every set of classes* is the goal of modular software development. This requirement is often referred to as *modular correctness*. A module is then, as far as its invariants are concerned, independent from the environment it is used in.

Definition 5. *Suppose $m = (T_m, E_m, I_m)$ is a module. We say, m fulfils its invariant contract* locally *if m fulfils it in the set T_m.*

Note, that this definition is stronger than just requiring that invariants be true in all reachable states of m.
We can quite easily prove that modules fulfil their contracts locally. In a system like the KeY tool one would prove that the invariants are true in all initial states and are preserved by any non-private method p occurring in T_m. This could be done by either symbolically executing the body of p or by resorting to the pre and post-conditions and modifies clauses for p.

However, we want modular correctness. So we need to bridge the gap between definitions 5 and 4. This is the goal of the following sections.

5 Invariants and their Dependencies

For a proper analysis of the validity of contracts, it is necessary to know which locations invariants depend on. We first introduce some basic notions.

Definition 6. *Let $\mathcal{M}$ be a module system, s a state of $\mathcal{M}$:*

(a) An attribute chain *$d = a_1.a_2.\cdots.a_n$ is a sequence of attribute names separated by dots. Attribute chains are assumed to be type correct, i.e. the attribute a_{i+1} is defined on the result type of a_i for all $1 \leq i < n$. The class attribute a_1 is defined in is called the start type of d.*
(b) A location *in state s is a pair (e, d) consisting of an attribute chain d and an object e of the same type as the start type of d.*
(c) For a location (e, d) we denote by $e.d_s$ its evaluation in state s in the usual sense.

The attribute chains occurring in the next definition do not contain local variables. In Sect. 6, when we consider aliases to these attribute chains, we make however use of the possibility to have local variables included.

Definition 7. *A depends clause of an invariant φ is a (possibly infinite) set D_φ of attribute chains $a_1.\cdots.a_n$ for which the following property holds for all states s_1 and s_2:*
If, for all $d \in D_\varphi$, and all objects e of the start type of d existing in both states $e.d_{s_1} = e.d_{s_2}$, then $(s_1 \models \varphi$ iff $s_2 \models \varphi)$.

[1] If subtyping was treated interfaces would also be incorporated.
[2] Usually, though out of scope of this paper, operation contracts would be added.

The existence of a depends clause, or even a minimal depends clause, for every φ is trivial since the set D of *all* attribute chains satisfies the requirements and the intersection of a decreasing chain of depends clauses is again a depends clause. Computing the exact depends clause or a good approximation to it is a completely different matter. There are two approaches: [18] requires an invariant to be restricted such that it has to obey a certain form and the locations it depends on can be extracted easily. Another option is to require explicit depends clauses for invariants, as done in [16, 14, 12] and to apply static checks.
Example. The set $\{\mathsf{start.year}, \mathsf{end.year}, \mathsf{start.month.val}, \mathsf{end.month.val}\}$ is a depends clause of the invariant of Period.

6 Module-Protection

In the example of Sect. 2 we have observed that exposing references to certain attributes (in the example: start and end) is harmful to the preservation of invariants. In this section we propose a theoretical criterion, called *module-protection*, or m-protection for short, which strictly eliminates such effects. In Sect. 7 we consider some known and easily checkable modularity requirements and show that each of them implies module-protection.

We need some auxiliary notions, defined as follows, where $m = (T_m, E_m, I_m)$ is a module and $\mathcal{M}$ a module system containing m and s a state of $\mathcal{M}$:

$Obj_s(m)$ denotes the set of all objects that exist in s and are instances of some class in T_m. Further we define:

$$IObj_s(m,\mathcal{M}) := \bigcup_{\substack{m' \in \mathcal{M} \\ m' \text{ imports } m}} Obj_s(m') \qquad\qquad Obj_s(\mathcal{M}) := \bigcup_{m' \in \mathcal{M}} Obj_s(m')$$

Since only those locations that are changed by some method can affect the evaluation of invariants, we make the modification of locations explicit by defining the modifier set $Mod_s(p)$ [3]: For every method p in $\mathcal{M}$, $Mod_s(p)$ denotes the set of locations (e', d'), $e' \in Obj_s(\mathcal{M})$, d' an attribute chain in $\mathcal{M}$ such that $e'.d'_s \neq e'.d'_{s_1}$, where s_1 is the state reached by executing p on e' in s.

For $e \in Obj_s(m)$ we define:

$$\begin{aligned} R_s(e) = \{e' \in Obj_s(\mathcal{M}) \mid\ & e' = e.d'_s \text{ for some attribute chain } d' \text{ and} \\ & \text{for all } e_0 \in IObj_s(m,\mathcal{M}) \text{ and all attribute chains } d = a_1.\cdots.a_n \\ & \text{with } e_0.d_s = e' \text{ there is some } i \text{ with } e_0.(a_1.\cdots.a_i)_s = e \\ & \text{or for all methods } p \text{ of } e_0 \ \ (e_0, d) \notin Mod_s(p)\} \end{aligned}$$

Thus $R_s(e)$ consists of all objects e' that can be referenced from e and any reference to e' from any object from a module that imports m passes through e or is readonly. We say, $R_s(e)$ is the set of objects *completely controlled* by e.

Analogously, the set of objects *completely controlled* by m consists of those objects that are referenced by objects of m but which are only referenced from importing modules of m by passing through m or in a readonly way:

$$\begin{aligned} R_s(m) = \{e' \in Obj_s(\mathcal{M}) \mid\ & e' = e.d'_s \text{ for some attribute chain } d' \text{ and some } e \in Obj_s(m) \\ & \text{and for all } e_0 \in IObj_s(m,\mathcal{M}) \text{ and all attribute chains } d = a_1.\cdots.a_n \\ & \text{with } e_0.d_s = e' \text{ there is some } i \text{ with } e_0.(a_1.\cdots.a_i)_s \in Obj_s(m) \\ & \text{or for all methods } p \text{ of } e_0 \ \ (e_0, d) \notin Mod_s(p)\} \end{aligned}$$

Note, that $e \in R_s(e)$ and $Obj_s(m) \subseteq R_s(m)$. The attribute chains d and d' in the above definitions may contain arbitrary attributes from $\mathcal{M}$. In extension to Def. 6 we allow attribute chains to contain local variables as virtual attributes here.

For the proof of the main theorem the following auxiliary lemma is needed:

Lemma 1. *(a) Let e, e' be objects in state s with $e \notin R_s(e')$ and p some method in e. If $(e, d.b) \in Mod_s(p)$ for an attribute chain d with $e.d_s \in R_s(e')$ and an attribute b then p modifies $e.d_s.b_s$ only by (indirectly) calling a method of e'.*

(b) Let e be an object in state s and m a module with $e \notin R_s(m)$ and p some method in e. If $(e, d.b) \in Mod_s(p)$ for an attribute chain d with $e.d_s \in R_s(m)$ and an attribute b then p modifies $e.d_s.b_s$ only by (indirectly) calling a method of m.

Definition 8. *Suppose a is an attribute defined in a class c which is part of a module m and m is itself part of a module system $\mathcal{M}$.*
Attribute a is called m-protected *in $\mathcal{M}$ if for every state s of $\mathcal{M}$, and any instance e of c we must have $e.a_s \in R_s(m)$. Attribute a is* strictly m-protected *in $\mathcal{M}$ if for every state s of $\mathcal{M}$, and every instance e of c we must have $e.a_s \in R_s(e)$.*

If an attribute is strictly m-protected it is also m-protected. Note that the easiest way to fulfil the requirements of module-protection is not to expose references to the object stored in $o.a$ at all.

We can formally state that it is sufficient to make the attributes of those locations the invariants of a module contract depend on—and not even all of them—module-protected. For the Theorem recall that we have excluded subtyping which would invalidate the statement. In the presence of subtyping we would have to require that the module system "respects behavioural subtyping".

Theorem 1. *Let m be a module that fulfils its invariant contract locally, let D be the union of depends clauses of all invariants of m and let T be the union of classes of those modules that (transitively) import m. If, for all attribute chains $a_1.\cdots.a_n \in D$ either $n = 1$ or for all $i = 1, \ldots, n-1$ one of the following is true:*

- *a_i has a type of a class defined in m,*
- *a_i defined in a class of the module m is m-protected, or*
- *a_i defined in a class of a module $m' \neq m$ is strictly m'-protected,*

then m fulfils its invariant contract in T.

Proof. (Sketch) Let φ be an invariant for a class in m, T defined as above, and s_0 a state of $\mathcal{M}$ with $s_0 \models \varphi$. Let p' be an arbitrary method in T, $e' \in Obj_s(T)$. By executing p' on e', a number of statements $sta_1, sta_2, \ldots, sta_k$ of T are executed, which yield the states $s_1, s_2, \ldots, s_k$ (resp.). We show by induction a stronger condition than necessary, namely that if $s_0 \models \varphi$ then for all $r = 1, \ldots, k$: $s_r \models \varphi$.

For the induction step, we assume $s_k \models \varphi$ and perform a case distinction on the kind of statement sta_k (we present only the obviously relevant ones according to the Java semantics):

- The statement is of the form x.a=y; where x is an instance of a class of T, which cannot be part of some $a_1.\cdots.a_n \in D_\varphi$ because the occurring attributes are from m and I_m. Thus for all $d \in D_\varphi$ and all objects e existing in s_k and s_{k+1}: $e.d_{s_k} = e.d_{s_{k+1}}$. By Def. 7: $s_{k+1} \models \varphi$.
- The statement is a method call on an object of $Obj_{s_k}(m)$. By locally fulfilling m's invariant contract it is ensured that $s_{k+1} \models \varphi$.
- Method calls within T. There is no state change except from exchanging the current this object and sta_{k+1} is the first statement of the called method.
- Other method calls p.
  If for all attribute chains $d \in D_\varphi$ and all objects e_0 in s_k of the start type of d we have $e_0.d_{s_k} = e_0.d_{s_{k+1}}$ then we are by Def. 7 finished. So let us assume $e_0.d_{s_k} \neq e_0.d_{s_{k+1}}$ for some e_0 and some $d = a_1.\cdots.a_n \in D_\varphi$. This implies that there is some i, $0 \leq i < n$ and a location (e', d') with $e'.d'_{s_k} = e_0.(a_1.\cdots.a_i)_{s_k}$ and $(e', d'.a_{i+1}) \in Mod_{s_k}(p)$. Let us abbreviate $e'.d'_{s_k}$ by e_p.
  Case 1: $e_p \in R_{s_k}(m)$. Because of Lemma 1, the modification is done during invocation of a method p_m of m which ensures $s_{k+1} \models \varphi$ when p returns.
  Case 2: $e_p \notin R_{s_k}(m)$. Since $e_0 \neq e_p$ we must have $n > 1$. Since the type t of a_i is also the type of e_p the case assumption yields that t is no class of m. If a_i was defined in m it cannot be m-protected since that would contradict $e_p \notin R_{s_k}(m)$.

Thus, according to the assumptions of the theorem: a_i is defined in a module $m' \neq m$ (thus $i > 1$) and is strictly m'-protected. This implies that for $e = e_0.(a_1.\cdots.a_{i-1})_{s_k}$: $e_p \in R_{s_k}(e)$. By Lemma 1 the state change is performed in a method p_e of e; since $a_1, \ldots, a_{i-1}$ are either m-protected, strictly m'-protected or have a type of m, and a_1 is declared in m we can conclude inductively that there is a method p_m of m that has (indirectly) invoked all calls to p_e and ensures again $s_{k+1} \models \varphi$. □

Note that we do not need to check for the last component of a depends location. This implies that invariants that only depend on attribute chains of the form $o.a$ are harmless. This fact is the immediate consequence that objects fully control their direct references.
Example. It is sufficient to make self.start and self.end PeriodMod-protected, and self.start.month and self.end.month strictly DateMod-protected because it must be prevented that references to them are available in importing modules of PeriodMod.

7 Establishing Module-Protection

In the following subsections we address pragmatic questions and observe that some well-known and statically checkable patterns, such as *final attributes*, *uniqueness*, *ownership*, and *confinement*, are sufficient to show (strict) module-protection of an attribute.

7.1 Final Attributes

References to objects with only final attributes (i.e. attributes that can be assigned a value only once, namely when constructing an object [9]) cannot be modified by any method. Thus, if an attribute a is of such a type, a is strictly m-protected. A pervasive use of final attributes results in immutable objects.

Lemma 2. *Suppose a is an attribute defined in some class of module m. If the type of a is a class c_0 and all attributes of c_0 are final then a is strictly m-protected.*

Example. We make year and month in Date as well as val in Month final. The setYear method is then not compilable and must be replaced by a method that returns a new Date object. Now, the attributes start and end are PeriodMod-protected as well as month is strictly DateMod-protected. Theorem 1 ensures that Period's invariant is fulfilled in any context it is put.

7.2 Uniqueness

Definition 9. *Suppose a is an attribute defined in a class c. a is called unique if, in every state and for every instance e of c there is no object $e' \neq e$ with a direct reference to $e.a_s$.*

Uniqueness can be statically checked [6] or proven with a JavaCard Dynamic Logic proof obligation.

Lemma 3. *A unique attribute defined in a class of module m is strictly m-protected.*

Example. Uniqueness helps to preserve the mutability of Date, by using defensive copies in constructors and getter methods [4] as shown in Fig. 3. The constructor of Period and the setYear method, both create copies of the parameter Date objects before they are stored in the attributes. As well, getEnd only returns a copy. Thus, an object stored in these attributes is never shared, especially not with objects outside PeriodMod. If we assume that month in Date is still m-protected by having val final, all relevant attributes are module-protected, giving an example of how a combination of two techniques, finality and uniqueness, ensures modular correctness.

```
class Period {
    private Date start , end;
    public Period(Date start , Date end) {
        if (end.earlierThan ( start )) throw new RuntimeException();
        this . start =new Date(start.getMonth(), start . getYear ());
        this .end=new Date(end.getMonth(), end.getYear());
    }
    public void setEnd(Date end) {
        if (end.earlierThan ( start )) throw new RuntimeException();
        this .end=new Date(end.getMonth(), end.getYear());
    }
    public Date getEnd() {
        return new Date(end.getMonth(), end.getYear());
    }
}
```

Fig. 3. The attributes of Period made unique.

7.3 Ownership

We give the following characterisation of ownership [8, 16, 18, 5]:

Definition 10. *Given the acyclic partial* ownership function *$owner_s$ on the objects of some state s with the property for all objects e, e' (ingoing and outgoing reference invariant):*

$$\textit{if } e \textit{ holds a direct reference to } e' \textit{ then } e = owner_s(e') \textit{ or } owner_s(e) = owner_s(e')$$

If, for all states s and for all instances e of a class, $owner_s(e.a_s) = e$ then a is owned.

This property can be statically checked [16, 18] on Java programs by ownership type systems.

Let a be an owned attribute. In state s, let e be an instance of c (of module m) with $owner_s(e.a_s) = e$. Thus (1) $e.a_s$ is referenced by e and (2) the reference invariant implies that from all objects $e_0 \in IObj(m, \mathcal{M})$ all attribute chains to $e.a_s$ pass through e. It can be concluded that a is strictly m-protected.

Lemma 4. *An owned attribute a defined in a class c that belongs to a module m is strictly m-protected.*

7.4 Non-exported Confined Types

In [20], Vitek and Bokowski describe how confinement to Java packages allows to restrict aliasing and give a procedure to statically check confinement. We require, in addition to Def. 1 (which provides a weaker condition) that non-exported types of modules are confined to the module. The procedure of [20] can be used to check this condition.

Lemma 5. *All attributes that have a non-exported type are m-protected for the module m they are defined in.*

8 Related Work

Much recent work has been devoted to techniques for alias protection. We are however aware of only few papers concerned with combining and exploiting different approaches for the purpose of verification.

Ownership types as the most popular approach to control alias effects have been first introduced by [8], from which different variants emerged [7, 16, 17, 5], e.g. [17] introduced readonly references to access owner contexts. In the context of Müller's work [16] on universe types, theories on how ownership models can be exploited for modular verification of invariants have been developed: Recent work by Müller et al. [18] relies completely on an ownership model giving up some flexibility compared to our approach; there is also no explicit notion of modules, which makes it harder to formulate invariants that describe properties that are only internal to classes developed at the same time. Leino and Müller [13] developed another approach with the same restrictions as above; moreover it seems difficult to convince developers to use the required explicit code annotations.

Unique pointers, which provide much stricter control of aliases than ownership types, have been applied in the Island concept [10] and by Boyland[6]. Approaches for alias protection on the type level (i.e. confinement) have been proposed by [20]. All three techniques are employed in our work.

Depends clauses were introduced and refined in [12, 14, 19] and are implemented in the Java Modeling Language (JML) [11]. The details of this notion slightly differ from ours.

9 Conclusions and Future Work

We have presented a framework for the modular verification of object-oriented programs. Much needs still to be accomplished: The concept of depends clauses has to be refined: especially how clauses are extracted or proved has to be investigated and, as well, how infinitely large clauses are denoted. The effects of subtyping to modularity have not been covered here, and are planned to be subject of another paper. Our requirements on imported modules are moreover quite strict: If an attribute is prone to be aliased it gets very strictly protected. Thereby, importing modules are enabled to rely on the contracts of the imports. This is advantageous, since in importing modules only such checks have to be performed that are fast static analyses (with the only exception—omitted in this abstract—that subtypes have to be checked to be behavioural subtypes, what remains a prover task). A variant of our method could provide *weak* contracts where importing modules would have to prove certain additional properties (similar to operation preconditions) for the import in order to rely on the contract of the imported module.

An important question we are going to investigate is how to formalise module contracts on a higher abstraction level instead of referring to concrete locations and thus not to expose implementation details. Only then, the effective use of our modules is feasible.

A main contribution is a framework for modules and module contracts in object-oriented languages on the basis of the concept of contracts. As an advantage compared to existing approaches, our way to modular correctness does not depend on a particular type system. Instead a programmer is free to choose, whether he prefers an ownership model or simpler techniques like using immutable classes, as long as the applied pattern ensures the criterion stated here, namely the module-protection property on relevant attributes. Moreover, the effort for establishing modular correctness is in a sense minimal: All measurements are based on an analysis of the module contract. Only for the extracted depends-clauses, restrictions on legal programs must be applied.

Acknowledgements. We would like to thank Richard Bubel and Steffen Schlager for useful comments on earlier versions of this paper. We also thank the anonymous referees for their detailed reviews.

References

1. W. Ahrendt, T. Baar, B. Beckert, R. Bubel, M. Giese, R. Hähnle, W. Menzel, W. Mostowski, A. Roth, S. Schlager, and P. H. Schmitt. The KeY tool. *Software and System Modeling*, 2004. To appear.
2. B. Beckert. A dynamic logic for the formal verification of Java Card programs. In I. Attali and T. Jensen, editors, *Java on Smart Cards: Programming and Security. Revised Papers, Java Card 2000, International Workshop, Cannes, France*, volume 2041 of *LNCS*, pages 6–24. Springer-Verlag, 2001.
3. B. Beckert and P. H. Schmitt. Program verification using change information. In *Proceedings, Software Engineering and Formal Methods (SEFM), Brisbane, Australia*, pages 91–99. IEEE Press, 2003.
4. J. Bloch. *Effective Java: Programming Language Guide.* The Java Series. Addison-Wesley, 2001.
5. C. Boyapati, B. Liskov, and L. Shrira. Ownership types for object encapsulation. In *ACM Symposium on Principles of Programming Languages (POPL)*, New Orleans, Louisiana, Jan. 2003.

6. J. Boyland. Alias burying: Unique variables without destructive reads. *Software—Practice and Experience*, 31(6):533–553, May 2001.
7. D. Clarke, J. Noble, and J. Potter. Who's afraid of ownership types? Technical report, Microsoft Research Institute, 1999.
8. D. Clarke, J. Potter, and J. Noble. Ownership types for flexible alias protection. In *ACM Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'98)*, Vancouver, Canada, October 1998.
9. J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java Language Specification.* Addison Wesley, 2nd edition, 2000.
10. J. Hogg. Islands: aliasing protection in object-oriented languages. In *Conference proceedings on Object-oriented programming systems, languages, and applications*, pages 271–285. ACM Press, 1991.
11. G. T. Leavens, A. L. Baker, and C. Ruby. Preliminary design of JML: A behavioral interface specification language for Java. Technical Report 98-06i, Iowa State University, Department of Computer Science, Feb. 2000.
    ftp://ftp.cs.iastate.edu/pub/techreports/TR98-06/TR.ps.gz.
12. K. R. Leino. *Toward reliable modular programs.* PhD thesis, 1995.
13. K. R. Leino and P. Müller. Object invariants in dynamic contexts. Available at http://research.microsoft.com/~leino/papers/krml132.pdf.
14. K. R. M. Leino and G. Nelson. Data abstraction and information hiding. *ACM Trans. Program. Lang. Syst.*, 24(5):491–553, 2002.
15. B. Meyer. *Object-Oriented Software Construction.* Prentice-Hall, Englewood Cliffs, second edition, 1997. http://www.prenhall.com/allbooks/ptr_0136291554.html.
16. P. Müller. *Modular Specification and Verification of Object-Oriented Programs.* PhD thesis, FernUniversität Hagen, 2001. Available from www.informatik.fernuni-hagen.de/pi5/publications.html.
17. P. Müller and A. Poetzsch-Heffter. Universes: A type system for alias and dependency control. Technical Report 279, Fernuniversität Hagen, 2001. Available from www.informatik.fernuni-hagen.de/pi5/publications.html.
18. P. Müller, A. Poetzsch-Heffter, and G. T. Leavens. Modular invariants for object structures. Available from http://www.sct.inf.ethz.ch/publications, 2003.
19. P. Müller, A. Poetzsch-Heffter, and G. T. Leavens. Modular specification of frame properties in JML. *Concurrency and Computation: Practice and Experience*, 15:117–154, 2003.
20. J. Vitek and B. Bokowski. Confined types in Java. *Software—Practice and Experience*, 31(6):507–532, 2001.
21. J. Warmer and A. Kleppe. OCL: The constraint language of the UML. *Journal of Object-Oriented Programming*, 12(1):10–13,28, Mar. 1999.