Specification and Verification
of Encapsulation in Java Programs

Andreas Roth

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

Abstract. Encapsulation is a major concept in object-oriented designs as design pattern catalogues, approaches for alias control, and the need for modular correctness of components demonstrate. The way encapsulation can be formally *specified* in existing approaches has several shortcomings. We show how encapsulation in sequential Java programs is specified by means of a new concept, called *encapsulation predicates*, in a clearly defined and comprehensible way, well fitting into the concept of *design by contract*. Encapsulation predicates extend existing functional specification languages. There are two kinds: basic predicates, which provide the actual extension, and convenience predicates, which are abbreviations for often used specification patterns. With encapsulation predicates, encapsulation properties in design patterns can be modelled and approaches to control aliasing can be simulated. Specifications containing encapsulation predicates are deductively checkable, but can also be tackled by static analysis methods which are similar to alias control approaches.

1 Introduction

Encapsulation plays a major role in object-oriented software development for at least one important reason: Without it, the complexity of inter-object relations would become uncontrollable, and one of the most basic concepts of computer science, the division of tasks into subtasks, indispensable to master complex problems, would be impossible.

It is thus quite striking that formal methods for software development have discovered only relative lately this problem for real-world object-oriented languages, and have so far provided solutions that are only partially satisfactory (Sect. 2.2).

A successful concept for formal methods in object-oriented software development is the notion of a (formal) *contract* [14]. It provides formal specifications in a way that perfectly reflects the way programmers informally reason about objects: namely by mutual responsibilities and services between objects. Surprisingly the concept of *design by contract* has so far only been applied to pure functional properties and not yet to properties of encapsulation. Our work contributes to the formalisation of encapsulation properties in the natural way of contracts by enriching specification languages with *encapsulation predicates*.

Programs can be verified with respect to formal contracts in a mathematically rigorous way. We thus provide means for making our formal specification of encapsulation properties checkable: We provide a deductive approach to realise this as well as sketch how existing methods from the static program analysis area can be integrated.

As basis of our reasoning we have chosen to investigate single-threaded programs in the language Java [9] for its widespread use in research and practice. The results should however be transferable to similar object-oriented languages.

2 Encapsulation as Important Object-Oriented Concept

The importance of encapsulation in object-oriented programs is a rather empirical phenomenon. It is manifested in properties of "good" software designs (documented in design pattern catalogues), other approaches to restrict aliasing (*alias control*), and the need for encapsulation in object-oriented components. The properties occurring in these areas are analysed in this section and serve as a basis to validate our solution in Sect. 6.

2.1 Design Patterns

We investigate the design pattern catalogues [8, 6, 10] for encapsulation properties. A selection of patterns that affect encapsulation is listed below; the list is far from complete, though these patterns are very clear manifestations of encapsulation properties in designs:

Whole-Part. This is a structural pattern [6] which provides a very strict encapsulation policy. There is an *aggregate object* (`Whole`) at work which hides access to other objects, called `Parts`: "Clients should see the aggregate object as an atomic object that does not allow any direct access to its constituent parts." [6]

Copy Mutable Parameters and

Return New Objects from Accessor Method. These two patterns [10] ensure that, at a method call, the passed mutable objects are copied before being stored at field locations and the returned objects are copied before being returned. The purpose of both patterns is to achieve encapsulation: No client of an object is allowed to directly access its internals.

Iterator. This behavioural pattern [8] ensures that an aggregate object, such as a linked list, does not expose its internal structure, though it provides an `Iterator` object that traverses the aggregate. Other clients of the list than iterators are not allowed to access its internals. They must however be enabled to put elements into the list, which makes the objects referenced by the internals also referenced by the clients.

These—and many other, e.g. the *Memento* [8] or the *Proxy* [8]—patterns have in common that they require some objects to be hidden from others in one or the other way. If the graphs made up by the references between objects in all states

of a system have such a property we speak of *encapsulation*: **An *encapsulation property* describes under which circumstances it is forbidden to have a reference from one object to another.** Obviously, the above mentioned patterns have properties that satisfy this—purposely rather vague—condition.

We believe that there is no sharp distinction between encapsulation properties and functional properties. To require, e.g., an object stored in the field of an object o to be different in all visible states from an object p can be considered both an encapsulation property (since we restrict the accessibility between objects) *and* a traditional functional invariant property.

Example 1. To be more concrete, we take up the example of an application of the Whole-Part pattern given in [6]. We have an object `Triangle` whose instances contain each three references (`p0`, `p1`, `p2`) to objects of class `Point`. `Point`s themselves consist of a pair of primitive integer fields `x` and `y`. Though immutable `Point` objects would be preferable, we explicitly allow in our design that `Point`s are mutable. Applying the Whole-Part pattern, `Triangle` should play the role of a `Whole` object and `Point` should play the role of a `Part`.

Following the pattern, the `Point` objects must not be shared among other graphical objects, since otherwise, e.g., a `rotate` operation on another object could unintendedly change the shape of the `Triangle`. Fig. 1 (without the grey parts) shows a UML class diagram of the design and an object diagram for a snapshot of the system; we disallow the reference labelled with ①.

For a comprehensive specification of the design, we want to specify, in addition to the mere functional behaviour (such as an invariant that the nodes of a triangle are not collinear), that instances of `Point` cannot be accessed by any other object than the specific triangle which it is a node of. Since the desired property describes a behaviour that must be observed in any visible state of a `Triangle` object, we would like to describe it as a class invariant of `Triangle`.

Example 2. To increase the level of complexity a bit, we assume now that, in addition to Example 1, `Point` contains additional references to other objects, such as to instances of `Colour`; the colour of the triangle is determined to be the "gradient" of the colours of its nodes. The representation of a `Colour` object is left open, it may consist of an "RGB" triple of primitive integer fields, or may have references to further objects.

A possible design decision would be to make `Colour`s in general sharable among other graphical objects such as `Point`s, but not if they belong to `Point`s that are constituent parts of *different* `Triangle`s. This restriction still allows `Point`s of the same triangle to reference the same `Colour`. So modifying a colour not belonging to a triangle t does not affect the state of t: Each node together with its associated colour is in fact a true part of the triangle, as the Whole-Part pattern requires. Like in the previous settings, `Point` objects being part of a `Triangle` must not be shared among other graphical objects.

Fig. 1 (including the grey extension) illustrates the design. The references ① and ② are not allowed in our design.

Again we would like to specify this more challenging encapsulation policy by means of an invariant of `Triangle`.
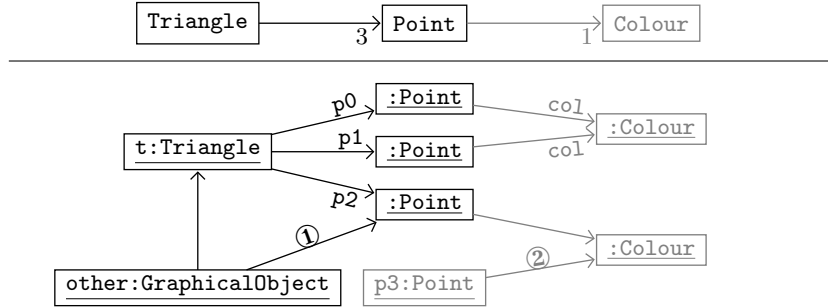
Fig. 1. UML class diagrams (top) and object diagrams (bottom) for Example 1, extensions for Example 2 are grey

The brief investigation of patterns should have shown:

(DP1) Encapsulation is the result of purposely made design decisions.
(DP2) There is not *the* encapsulation property, but there are many varying— and arbitrarily complex—encapsulation properties.
(DP3) There is no sharp line between functional and encapsulation properties.

2.2 Alias Control: Related Work on Encapsulation

Quite a number of techniques have been published in recent years that aim at reducing the complexity introduced by aliasing in programs with pointers, as for example *islands* [12], *balloons* [2], *uniqueness* [5], and different types of *ownership* [7, 15]. We refer to them as *alias control policies*. Overviews are, e.g., in [17]. According to our criteria, these policies ensure properties that can be classified as encapsulation properties.

Most, if not all, of these policies are however technology driven, that is the properties are mostly statically checkable (e.g. by means of a type checker), which is the major justification that the approach exists. We claim that we can formulate each of the properties summarised in [17] with our approach, and we will demonstrate this in some examples below. Moreover we can observe that the investigated design patterns require more generality concerning their encapsulation properties than the existing encapsulation policies provide. Finally, users are facing two ways of writing specifications: the one they are usually used to, *design by contract*, writing invariants and pre-/postcondition contracts, and on the other hand, a completely different way of denoting encapsulation properties, e.g., by labelling fields with a special modifier. We believe that this distinction is unnecessary and unnatural, thus confusing for developers, especially for those who are sceptical towards formal specification anyway.

To sum up, we can state the following weaknesses of the existing *alias control* approaches to master encapsulation:

(AC1) There is an irritating difference between how functional properties and how encapsulation properties are specified in recent approaches.
(AC2) The way encapsulation properties are specified is closely coupled to technologies that check them, which makes it likely that not all desired properties can be formulated.

2.3 Components

Encapsulation is indispensable for the specification and verification of object-oriented software components. Let A be a component that is used by component B and let C be a component that uses both A and B. Assumed that B's methods preserve an invariant φ. Also, φ holds in initial states of B's objects. However, if φ includes statements about the state of objects from A, then these assumptions will *not* ensure that B's invariant holds whenever B is used, since it might be possible that during the use of B in C φ gets violated due to modifications of objects from A which "bypass" B. This undesired behaviour can only be prevented modularly if there is more encapsulation: all uses of A in B that affect B's invariant must only be accessible by means of B.

This observation means however that guarantees on functional properties of components ("trusted components") can only be made if there is a sufficient— and equally well guaranteed—degree of encapsulation to ensure *modular correctness*. More on this issue, in particular on the question of *which* objects are to be encapsulated, can be found in [18].

3 Outline

Taking the results of our reviews of design patterns and alias control policies into account, the central idea of our work is thus: Programmers know how they encapsulate data (DP1), they should be enabled to easily specify their encapsulation concept formally and to check these properties with machine assistance. Especially, (AC1) and (DP3) encourage us to make encapsulation specifiable in a way traditional functional properties are, i.e. by applying *design by contract* and extending a specification language. Moreover, this has the flexibility required by (DP2) and the independence from concrete techniques required by (AC2).

The obvious way to get new features, such as encapsulation properties, into specification languages is to make them accessible as special predicates of the specification language. As any other predicate they can then be connected with other expressions of the language. Language expressions containing the predicates may serve as preconditions, postconditions, or class invariants.

Basically two predicates, *acc* and *reachable* which we will introduce in Sect. 5 are needed to express any encapsulation property we can reasonably expect. They are usually not available in specification languages such as JML[1] [13] or UML/OCL [20]. Since these predicates are most probably hard to handle and do not intuitively reflect the way one wants to specify encapsulation, there is a second layer on top of these basic predicates. Predicates defined in this layer can easily be applied to the reference patterns and alias control properties from above, as demonstrated in Sect. 6.

Finally, in Sect. 7, we investigate how feasible it is to check encapsulation predicates. Since it was left open which technology to use (AC2), we will consider both a deductive approach and approaches from the static analysis area.

First we have a look at the formal basis of our reasoning.

---

[1] In fact, there is an equivalent for the predicate *reachable* available in JML.

4 Formal Background

This work has been done in the context of KeY [1], a project to establish formal specification and deductive verification within commercial software development. The KeY prover, integrated in a CASE tool or an IDE, enables developers to prove properties of Java[2] programs using a program logic called JavaDL [4]. JavaDL expressions may, in KeY, either be the result of a translation from specification languages such as UML/OCL or JML or of direct specification. Functional properties expressible in the aforementioned specification languages are as well expressible in JavaDL, and the other way round, extensions made to (the first order fragment of) JavaDL concerning additional predicates, can also be made to those languages. In this paper we rely on the JavaDL logic and trust in the ability of the reader to translate the definitions to his favourite language.

As follows, some formal properties of a first-order fragment[3] JavaFOL* of JavaDL [4] are defined, which will be extended by encapsulation predicates in the following sections. Note, that these properties are necessary to consolidate encapsulation predicates (Sect. 5) and their axiomatisation (Sect. 7.1).

We assume to have fixed a Java program (i.e. set of classes) P. By *available types* we characterise the Java types declared by P and the built-in Java types. The set *Term* of terms is built inductively from program variables, logical variables, the literal `null`, the `boolean`, `int`, and `String` literals, and in addition: $a_C \in$ *Term* for static fields $C.a$ of P, $t.a_C \in$ *Term* for terms t and (instance) fields a declared in class C of P[4], and $t_0[t_1] \in$ *Term* for terms t_0 and terms t_1.

The formulae *Fma* of JavaFOL* are constructed as usual from terms with user-defined predicate symbols, the predicate $\doteq$, junctors $\neg, \wedge, \vee, \rightarrow, \leftrightarrow$, and quantifiers $\forall$ and $\exists$ which bind logical variables. More precisely, each quantifier is indexed with one of the available types, i.e. for every available type tp there are quantifiers $\forall \cdot : tp$ and $\exists \cdot : tp$. $\forall x$ is an abbreviation for $\forall x$:`java.lang.Object` for every logical variable x. Additionally there is the unary predicate $instanceof_{tp}(\cdot)$ for every available type tp.

The semantics is defined by mapping terms to the domain $\mathcal{D}$ of Java objects or values and by the validity relation $\models$ for formulae. Both depend on a state s and a variable assignment β. For undefinedness a choice function ch is employed which delivers, for a term t, an arbitrary unknown but fixed domain element $ch(t)$ as advocated in [11]. The valuation $val_{s,\beta} : Term \rightarrow \mathcal{D}$ is defined as follows:

- for local variables and static fields $\underline{v}$: $val_{s,\beta}(\underline{v})$ is the object or value assigned to the variable (or static field) v in state s.
- for a logical variable x: $val_{s,\beta}(x) = \beta(x)$
- for a term $t.\underline{a}$: $val_{s,\beta}(t.\underline{a}) = \begin{cases} val_{s,\beta}(t).a & \text{if } a \text{ is defined for } val_{s,\beta}(t) \\ ch(t.\underline{a}) & \text{otherwise} \end{cases}$

---

[2] More precisely, only the sequential subset JavaCard is considered.
[3] Though JavaDL is a typed logic, our presentation provides a version that has no types on the syntax level.
[4] The subscript C is skipped if it is clear from the context where a field is declared.

– for a term $t_0[t_1]$:

$$val_{s,\beta}(t_0[t_1]) = \begin{cases} val_{s,\beta}(t_0)[val_{s,\beta}(t_1)] & \text{if } val_{s,\beta}(t_0) \text{ is of an array type} \\ & \text{and } val_{s,\beta}(t_1) \text{ is an int value } i \\ & \text{with } 0 \leq i < val_{s,\beta}(t_0).\texttt{length} \\ ch(t_0[t_1]) & \text{otherwise} \end{cases}$$

The validity $\models$ of formulae is (for formulae φ, terms t_1, t_2, logical variables x):

- $s, \beta \models t_1 \doteq t_2$ iff $val_{s,\beta}(t_1) = val_{s,\beta}(t_2)$
- $s, \beta \models \neg\varphi$ iff not $s, \beta \models \varphi$, etc.
- $s, \beta \models \exists x{:}\, tp\ \varphi$ iff there is an initialised object $d \in \mathcal{D}$ which is assignment compatible [9] to type tp such that $s, \beta_x^d \models \varphi$, analogously for $\forall x{:}\, tp\ \varphi$
- $s, \beta \models instanceof_{tp}(t)$ iff $s, \beta \models \exists x{:}\, tp\ x \doteq t \wedge \neg(t \doteq \texttt{null})$

If $s, \beta \models \varphi$ holds for all β, we just write $s \models \varphi$. In the sequel sets of fields A are considered. A is partitioned in sets A^{inst} and A^{stat} such that A^{stat} contains all the static fields of A and $A^{inst} = A \backslash A^{stat}$. We further denote the set of all fields of a program P as $Fields(P)$.

5 Basic Encapsulation Predicates

In this section, the two basic encapsulation predicates, the *acc* and the *reachable* predicate, are defined for JavaFOL*. As already mentioned, they can likewise be defined for any specification language that is capable of making statements about program states, such as JML or UML/OCL. In Sect. 6 they are complemented with a mere convenience layer, i.e. we provide handy abbreviations. This however means that this section presents all the needed extensions to express encapsulation properties. All applications to design patterns and alias control properties shown in Sect. 6 could be done with the basic predicates of this section only. The formulae would just be more intricate.

5.1 The *acc* Predicate

In object-oriented specification languages as well as in JavaDL there are only means to reason about concrete field accesses but there is no way to talk about an *arbitrary* field access, such as "there is a field such that...". Without getting too much into the spheres of higher order logic, this restriction needs to be relaxed by defining an *acc* predicate. It defines the relation of objects which can be accessed with exactly one field or array access from a list of allowed fields.

Definition 1 (Syntax of *acc*). *Let A be a set of fields defined in a Java program P, and t_1 and t_2 JavaFOL* terms for P. Then $acc[A](t_1, t_2)$ is a formula of JavaFOL*. For a program P, $acc(t_1, t_2)$ is regarded as abbreviation for $acc[Fields(P)](t_1, t_2)$.*

What does *accessibility* between two objects exactly mean? Since our goal is to cope with the design patterns mentioned above, we clarify the question by

looking at one of them more carefully. In the Whole-Part pattern, the restricted accessibility is supposed to ensure that the state of Parts cannot be modified by clients others than from the corresponding Whole. State changes on a Part object p are performed by invoking methods of p or directly assigning to a field or to an array slot of p. Both possibilities require that the object that performs the modification holds a reference to p. References may be held either in a local variable, a field, or an array slot. Like when invariants are considered, we are however only interested in states directly before and after method invocations (*visible states*). Since such a method invocation cannot change the assignments to local variables of the caller with the exception of the returned value, all local variables but the one assigned to by the call, can be ignored. The return value can be taken into consideration if we, without loosing generality, look at method calls that assign to a field of an anonymous class (see Sect. 7). This justifies the following two possibilities of how to access an object e_1 directly from a given one e_0: e_1 may be stored in a field of a class instance e_0 or e_0 is an array object of which one slot stores e_1.

In addition we say that e_1 is accessed from e_0 if there is some static field that references e_1. For encapsulation this is crucial: With static fields, encapsulation is practically completely compromised since they provide global access to the object of question. In fact, it would be sufficient only to consider *visible* static fields. For simplicity, the slightly more conservative approach to take *all* static fields into account is taken here.

The three possibilities for an access are reflected in the semantics of *acc*:

Definition 2 (Semantics of *acc*). $s, \beta \models acc[A](t_1, t_2)$ *is defined to hold for a state s iff* $(val_{s,\beta}(t_1), val_{s,\beta}(t_2)) \in Acc[A]$. $Acc[A]$ *is defined to be a relation on D (that is a subset of $\mathcal{D} \times \mathcal{D}$) with $(e_0, e_1) \in Acc[A]$ iff*

- e_0 *is a class instance (i.e. not an array object) and there exists $a \in A^{inst}$ with $e_1 = e_0.a$, or*
- *there exists $a \in A^{stat}$ with $e_1 = val_{s,\beta}(\underline{a})$, or*
- e_0 *is an array and there exists $j \in \{0, \ldots, e_0.\texttt{length}\}$ with $e_1 = e_0[j]$.*

5.2 The *reachable* Predicate

Reasoning about encapsulation often means reasoning about restricted reachability. In this section a *reachable* predicate is defined that can be used to reason about these restrictions. Essentially, reachability is the reflexive and transitive closure of the *acc* relation defined in the last section. So we define *reachable*, a binary predicate for each set of fields, as follows:

Definition 3 (Syntax and Semantics of *reachable*). *Let A be a set of fields defined in a Java program P, and t_1 and t_2 JavaFOL* terms for P. Then reachable$[A](t_1, t_2)$ is a formula of JavaFOL*. Again reachable(t_1, t_2) is considered to be an abbreviation for reachable$[Fields(P)](t_1, t_2)$.*

For a state s and a variable assignment β, $s, \beta \models$ reachable$[A](t_1, t_2)$ iff there is a finite sequence of objects $(e_0, e_1, \ldots e_k)$ $(k \in \mathbb{N})$ such that $e_0 = val_{s,\beta}(t_1)$ and $e_k = val_{s,\beta}(t_2)$ and for all $i = 1, \ldots, k$: $(e_{i-1}, e_i) \in Acc[A]$.

6 Applications and Convenience Encapsulation Predicates

Though the predicates *acc* and *reachable* provide a basic vocabulary for specifying encapsulation behaviour, their use is still tedious. We thus provide handy abbreviations, or *convenience encapsulation predicates*, for useful application patterns. Below, their practicability is measured by formulating properties of the design patterns and the alias control approaches. In the end, all predicates introduced throughout this section are summarised in Table 1.

6.1 The *guardAcc* and *uniqueAcc* Predicates

We define an abbreviation for specifying the following property: If there is a direct reference between an arbitrary *guard* object x and an object u then x must satisfy $\varphi(x)$. In JavaFOL* we formalise this property as

$$guardAcc_x[A; \varphi(x)](u) \leftrightarrow \forall y \big(acc[A](y, u) \rightarrow \varphi(y) \big) \ .$$

As before the parameter A is optional and skipping it can be seen as abbreviation for using $Fields(P)$. The formula $\varphi(x)$ is the characteristic function of those objects, the *guard* objects, which are allowed to hold a reference to u. In the easiest and most common case, $\varphi(x)$ will consist just of an equality $x \doteq g$, thus having just one guard object g. This specification pattern is in fact so common that we introduce another convenience predicate called *uniqueAcc* which is defined by the equivalence

$$uniqueAcc(g, u) \leftrightarrow guardAcc_x[g \doteq x](u) \ .$$

Alias Control: Unique Pointer. A *unique object* is an object that is referenced by at most one object [5]. The *guardAcc* predicate can easily be used to model this property, e.g. to say that u is a unique object, we require that for every object z that has a direct reference to u, all other objects referencing u must be equal to z. Or simpler: z is the only guard object. Moreover there is an equivalent formulation with *uniqueAcc*:

$$\forall z(acc(z, u) \rightarrow guardAcc_x[x \doteq z](u)), \qquad \forall z(acc(z, u) \rightarrow uniqueAcc(z, u)) \ .$$

By inserting the definition of *guardAcc* and simplifying we get the following formula, which obviously fits our expectations of a unique object:

$$\forall z \forall y \big(acc(z, u) \wedge acc(y, u) \rightarrow y \doteq z \big) \ .$$

Pattern: Whole-Part. The *guardAcc* or the *uniqueAcc* predicate can be employed for simple versions of the Whole-Part pattern, namely if the part's state does not depend on additional objects. In this case, the Whole-Part pattern forbids direct access to the parts (instances of `Part`), only indirect accesses through the `Whole` object are allowed. We can now formally note this property as

$$\forall p{:}\texttt{Part} \ \exists w{:}\texttt{Whole} \ guardAcc_x[x \doteq w](p) \ .$$

or if we already know that the value in field `p` is a part of a `Whole` and using *uniqueAcc*, we write simpler: $\forall w{:}\texttt{Whole} \ uniqueAcc(w, w.\underline{\texttt{p}})$.

Example 3. For the settings in Example 1, we would require the following invariant:

$$\forall t\text{:}\texttt{Triangle} \left(uniqueAcc(t, t.\underline{\texttt{p0}}) \land uniqueAcc(t, t.\underline{\texttt{p1}}) \land uniqueAcc(t, t.\underline{\texttt{p2}}) \right) \ .$$

This exactly describes the desired property that nodes may only be accessed by means of `Triangle`. It is however *not* sufficient for the settings of Example 2 since then references to `Colour` objects would be allowed, even if they "bypass" the corresponding `Triangle` object.

Patterns: Copy Mutable Parameters, Return New Objects from Accessor Method. These two patterns ensure that an object stored in a field a of object o is only accessed through o itself, denoted in JavaFOL* as $guardAcc_x[x \doteq o](o.\underline{a})$. In contrast to many other patterns, these two patterns exactly define *how* encapsulation must be achieved, namely by copying parameters and return values. Only the effect of the pattern can be specified with encapsulation predicates.

Alias Control: Confinement. The formula that specifies the confinement [19] aliasing policy demonstrates that it is useful to have the possibility to use a formula $\varphi(x)$ to qualify guard objects.

   We assume that we have a unary predicate $confined_p$ which holds for every object whose type is confined to package p. Let $tp_1, \ldots, tp_n$ be the classes in package p. Only these classes may access types confined to p. The following formula describes this property:

$$\forall y \left(confined_p(y) \rightarrow guardAcc_x[instanceof_{tp_1}(x) \lor \ldots \lor instanceof_{tp_n}(x)](y) \right) \ .$$

6.2   The *guardReg* and *uniqueReg* Predicates

The *guardAcc* predicate restricts the accessibility of one single object. Often however, it is necessary to restrict the access to *all* objects (indirectly) referenced by a particular one. To ensure, e.g., that in the object graph of Fig. 1 references ① and ② are not allowed, it must be required that all objects reachable from `t.p0` are only reachable through `t`. No restriction should however be imposed on references within the group of objects reachable from `t.p0`. Using the *guardAcc* predicate we can formalise such properties as follows and define the *guardReg* predicate, an informal description follows afterwards:

$$guardReg_x[A; \varphi(x)](u) \leftrightarrow$$
$$\quad \forall z \left(reachable[A](u, z) \rightarrow z \doteq u \lor guardAcc_x[\varphi(x) \lor reachable[A](u, x)](z) \right)$$

or equivalently:

$$guardReg_x[A; \varphi(x)](u) \leftrightarrow$$
$$\quad \forall y \forall z \left((reachable[A](u, z) \land acc(y, z)) \rightarrow (z \doteq u \lor reachable[A](u, y) \lor \varphi(y)) \right) \ .$$

In this formalisation one can associate the objects of the protected "region" as those objects z which are reachable with fields A starting from an object u. For

u itself no restriction regarding incoming references is imposed (this explains $z \doteq u$ on the right side of the implication). Any reference from an object y to such a z must either satisfy $\varphi(y)$ or is itself part of this region (i.e. reachable from u via fields A).

Like there was a *uniqueAcc* predicate introduced for the *guardAcc* predicate we define a *uniqueReg* predicate as follows to capture the most common application of *guardReg*:

$$uniqueReg(g, u) \leftrightarrow guardReg_x[x \doteq g](u) \ .$$

Again, a survey of design patterns and alias control policies follows, to demonstrate the usefulness of the two additional convenience predicates. In addition, we mention how the predicate enables us to formally specify *compositions* of the modelling language UML.

Alias Control: Islands. An object *bridge* plays the role of a bridge if in all states the formula *uniqueReg(bridge, bridge)* holds [12]. By applying the definition and simplifying we get:

$$\forall y \forall z \big((reachable(bridge, z) \land acc(y, z)) \rightarrow (z \doteq bridge \lor reachable(bridge, y)) \big) \ .$$

Alias Control: Balloons. For balloons, it is required [2] that for an object b of a balloon type and the objects B indirectly referenced by b the property holds: b is referenced at most once, the referencing object is not in B, and all objects in B are only referenced by objects in $B \cup \{b\}$. Formalised in JavaFOL*, this property is:

$$uniqueReg(b, b) \land \forall v \big(acc(v, b) \rightarrow (uniqueAcc(v, b) \land \neg reachable(b, v)) \big) \ .$$

Pattern: Whole-Part. The Whole-Part pattern requires that there is no direct access to the parts (instances of `Part`), only indirect accesses through the `Whole` object are allowed. For simple structures, we have already observed above that it is sufficient to use a formulation using the *guardAcc* predicate.

If, however, the `Part` objects' representations consist of a more complex object structure, this formulation is insufficient as Example 3 has illustrated. Instead, we can express the desired property with the help of *guardReg*. The basic formalisation is:

$$\forall p\!: \texttt{Part} \ \exists w\!: \texttt{Whole} \ uniqueReg(w, p) \ . \tag{1}$$

The validity of this formula implies that all objects that are (indirectly) referenced by a `Part` are only accessed among each other or by a particular `Whole` object. For patterns like the *Proxy* or the *Memento* pattern similar conditions can be formalised with *guardReg*, and even variations (like that some objects may bypass a *Proxy*) are possible to formalise.

Example 4. We take up the settings of Example 2. The special variant of the Whole-Part pattern imposed there is that parts of the same aggregate may access internals of each other. The guard object is thus not only the `Whole` but also all objects reachable from the parts, i.e. those x satisfying

$$\varphi_g := x \doteq t \lor reachable(t.\underline{\texttt{p0}}, x) \lor reachable(t.\underline{\texttt{p1}}, x) \lor reachable(t.\underline{p2}, x)$$

The desired property is now formalised in JavaFOL* as follows:

$$\forall t\text{:}\texttt{Triangle}\ (\ guardReg_x[\varphi_g](t.\underline{\texttt{p0}})\wedge$$
$$guardReg_x[\varphi_g](t.\underline{\texttt{p1}})\wedge guardReg_x[\varphi_g](t.\underline{\texttt{p2}}))\ .$$

Pattern: Iterator. With this pattern, we show that the parameterisation of the predicate with a set of fields A is in fact useful. The objects protected by guard objects are the internals of the aggregate. For simplicity, we assume that the aggregate object is a linked list implemented in a class `LinkedList` (like the Java implementation `java.util.LinkedList`). We assume that the internals of the list are made up by objects of a class `Entry` connected by a `next` field. The first object of the internals is stored in the `header` field of a `LinkedList`. The guard objects are both the `LinkedList` instance and the `ListItr` iterator object (this could be made more precise by describing only `ListItr` instances of the particular list). The encapsulation property for the Iterator pattern is thus:

$$\forall l\text{:}\texttt{LinkedList}\ guardReg_x[\{\texttt{next}\}; x\doteq l\vee instanceof_{\texttt{ListItr}}(x)](l.\texttt{header})\ .$$

UML Compositions. In the modelling language UML, classes and their interrelation can be modelled in class diagrams. The relation between instances of classes are represented as *associations* between classes. *Compositions* are special kinds of associations, which are depicted with a filled diamond adornment (see Fig. 2). They emphasise that one partner in the relation has sole responsibility for man-

Fig. 2. UML composition

aging the parts of the other partner. As a consequence, accessing these objects directly is forbidden. This property can be formally captured as in (1):
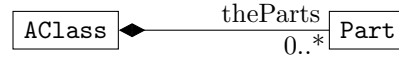
$$\forall p\text{:}\texttt{Part}\ \exists w\text{:}\texttt{AClass}\ uniqueReg(w,p)\ .$$

6.3   Direct Applications of the Basic Predicates

Ownership. We assume to have given an acyclic partial function *own* [15], which we formalise as predicate: $own(t_1, t_2)$ holds iff an object t_1 owns t_2 or t_1 is *null* and t_2 is not owned. We assume that the acyclicity of *own* is checked externally before, since this is out of reach of a first order formulation, and can in fact be enforced [15]. With the *acc* predicate, the ownership relation of [15] can simply be written down in JavaFOL*:

$$\forall e\forall e'\ acc(e,e')\rightarrow\Big(own(e,e')\vee\exists e_0\big(own(e_0,e)\wedge own(e_0,e')\big)\Big)\ .$$

7   Checking Encapsulation Predicates

This section deals with the question how specifications containing encapsulation predicates can be checked against an implementation. Using deduction for this is

Table 1. Overview of encapsulation predicates

Predicate	Definition
$acc[A](t_1, t_2)$	(Axiom, "there are direct references from t_1 to t_2")
$reachable[A](t_1, t_2)$	(Axiom, "there are access paths from t_1 to t_2")
$guardAcc_x[A; \varphi(x)](u)$	$\forall y(acc[A](y, u) \rightarrow \varphi(y))$
$uniqueAcc(g, u)$	$guardAcc_x[g \doteq x](u)$
$guardReg_x[A; \varphi(x)](u)$	$\forall y \forall z \big((reachable[A](u, z) \wedge acc(y, z))$ $\rightarrow (z \doteq u \vee reachable[A](u, y) \vee \varphi(y)) \big)$
$uniqueReg(g, u)$	$guardReg_x[g \doteq x](u)$

most natural, since this is the preferred way to verify programs w.r.t functional specifications; encapsulation properties can, as shown above, be formulated in the very same way. Static analysis has the appeal of a fully automated technique which should thus be made use of whenever possible. Our approach aims at a framework which integrates both techniques. Due to space restrictions, we can only give a brief impression of our approach and do not show correctness proofs.

7.1 Deductive Approach

The two predicates defined in Sect. 5 must be axiomatised to deductively treat encapsulation predicates. Both *acc* and *reachable* predicates are challenging, since both are beyond the original JavaDL expressibility (for the need to identify *all* fields in the program) and the latter has no complete axiomatisation.

The acc Predicate. The following axiom for the *acc* predicate is correct:

$$\forall x \forall y \Big(acc[A](x, y) \ \leftrightarrow \ \big(\quad \exists i : int \ (x[i] \doteq y) \\ \vee \bigvee_{a \in A^{stat}} \underline{a} \doteq y \quad \vee \bigvee_{a \in A} x.\underline{a} \doteq y \big) \Big) \ . \tag{2}$$

In contrast to the logic presented in Sect. 4, the logic implemented in the KeY prover is a *typed logic*, so even on the syntactic level there are types. Then, for every s, β and every term t of type tp, $val_{s,\beta}(t)$ is always assignment compatible to tp, so basically they are subtypes. This gives rise to an optimisation. It is obvious that we do not need to generate sub-formulae $x.\underline{a} \doteq y$ if we already know by the (static) type of x that x can not have a field a. By similar considerations we can skip one disjunct of (2) since we know, just by considering the type of x, that a term is either of an array type or not, and so one disjunct is irrelevant.

The *reachable* Predicate. The *reachable* predicate is axiomatised in a similar way as other attempts in literature [16]. Basically it is the reflexive and transitive closure of *acc*. The following axiomatisation is correct:

$$\forall x \forall y \Big(reachable[A](x, y) \leftrightarrow \big(x \doteq y \vee \exists z(acc[A](x, z) \wedge reachable[A](z, y)) \big) \Big)$$
$$\forall x \forall y \Big(reachable[A](x, y) \ \leftrightarrow \ \exists z \big(reachable[A](x, z) \wedge reachable[A](z, y) \big) \Big) \ .$$

It is well known that there is no complete axiomatisation of the transitive closure, and thus of *reachable*, in first order logic since the access graph may contain cycles and our domain is infinite [3]. However, it seems that the above (incomplete) axiomatisation is sufficient for practical purposes.

Proving Encapsulation Properties of Programs. For treating programs we need more than the first order fragment JavaFOL* used so far, that is full JavaDL plus the extensions made in the last two sections. On the syntactic level, JavaDL introduces a modality $\langle \cdot \rangle$: For every JavaDL formula φ and every sequence α of (correctly typed) Java statements, $\langle \alpha \rangle \varphi$ is a JavaDL formula. JavaDL semantics [4] is defined formally in terms of Kripke structures. Here, we just say informally that $\langle \alpha \rangle \varphi$ denotes the property of α to terminate in a state in which φ holds (total correctness). So $\psi \rightarrow \langle \alpha \rangle \varphi$ means (similar to a Hoare triple) that, if ψ holds initially then α terminates and afterwards φ holds.

With the axiomatisation of encapsulation properties and modalities, we can prove with a theorem prover that methods preserve encapsulation properties. If all public methods and constructors preserve an encapsulation property the property holds in all visible states of the program.

In order to prove the preservation of an (encapsulation) property φ for a method call `e.m(e`$_0$`, ..., e`$_n$`);` we have the following proof obligation:

$$\varphi \rightarrow \langle \texttt{Ano.o = e.m(e}_0\texttt{, ..., e}_n\texttt{);} \rangle \varphi \ . \tag{3}$$

Since our notion of accessibility refers only to field or array accesses, and deliberately *not* to local variables, the method call we investigate assigns its return value to a suitable static field of an "anonymous" class `Ano` which is unused in the rest of the code. This reflects the fact that callers of the method may reference the returned object and, thus, have direct access to it.

We have extended the KeY prover [1] with some of the predicates from above. The example below reports on a sample application.

Example 5. Let us take up the settings from Example 1. To class `Triangle`, we add a method `getPoint0()`. Our first attempt is to attach to `getPoint0()` the implementation `return p0;`. This is however not acceptable since callers reference `p0` after `getPoint0()` has terminated, though the encapsulation specification requires unique access through the `Triangle` object.

Proof obligation (3), instantiated as follows, can thus not be proven:

$$\forall t\!:\!\texttt{Triangle} \ guardAcc_x[x \doteq t](t.\underline{\texttt{p0}})$$
$$\rightarrow \langle \texttt{Ano.o = tria.getPoint0();} \rangle \forall t\!:\!\texttt{Triangle} \ guardAcc_x[x \doteq t](t.\underline{\texttt{p0}}) \ .$$

Implementing `getPoint0()` with `return new Point(p0.getX(),p0.getY());` makes the proof obligation however—as desired—provable. The KeY prover requires some manual, though trivial, quantifier instantiations to close the proof.

7.2 Static Analysis Techniques

Sect. 2.2 already referenced static analysis techniques called *alias control policies* and the encapsulation properties they check. Though encapsulation predicates

provide a much more general framework than these rather specialised properties, the use of deductive verification to prove properties expressed by means of encapsulation predicates has a major disadvantage compared to the static analysis techniques employed for *alias control*: In general, deduction requires user interaction, which is especially in the case of treating *reachable* non-trivial, while the techniques to prove *alias control* properties are usually fully automated static analyses. Automated techniques can nevertheless be used even in the general setting of encapsulation predicates. By a simple type checker like algorithm (similar to those of ownership type systems) we can, e.g., check that a piece of code α preserves the property $uniqueReg(w, w.p)$. So, instead of deductively verifying this, this algorithm can be invoked by the following "sequent calculus rule":

$$\frac{\Gamma,\, uniqueReg(w, w.p) \vdash \#analyse(\langle\alpha\rangle uniqueReg(w, w.p)),\, \Delta}{\Gamma,\, uniqueReg(w, w.p) \vdash \langle\alpha\rangle uniqueReg(w, w.p),\, \Delta} \; .$$

If the static check is successful $\#analyse(\ldots)$ is rewritten to *true*, otherwise the argument $\langle\alpha\rangle uniqueReg(w, w.p)$ is returned.

8 Future Work and Conclusions

This work contributes to the application of formal methods to object-oriented software development by

- emphasising the importance of encapsulation properties for the formally correct realisation of design decisions,
- giving specifiers means to easily formulate a wide range of encapsulation properties *within* the traditional methodology *design by contract* by using basic and convenience *encapsulation predicates*,
- providing means to verify encapsulation properties from within an interactive theorem prover and integrating static analysis methods.

We have demonstrated that the approach is suited to formulate encapsulation properties occurring in relevant software designs by formalising design patterns.

As future work, we want to further work on integrating advanced static procedures into our framework to resolve encapsulation properties. Moreover, we envisage the automated generation of a formal encapsulation specification from software components with an attached formal specification. As pointed out in Sect. 2.3 such an encapsulation specification of a component is a major step towards components with guaranteed functionality.

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*, 4:32–54, 2005.

2. P. S. Almeida. Controlling sharing of state in data types. In M. Aksit and S. Matsuoka, editors, *ECOOP '97–Object-Oriented Programming*, volume 1241 of *Lecture Notes in Computer Science*, pages 32–59. Springer, 1997.

3. T. Baar. The definition of transitive closure with OCL—limitations and applications. In *Proceedings, Fifth Andrei Ershov International Conference, Perspectives of System Informatics, Novosibirsk, Russia*, volume 2890 of *Lecture Notes in Computer Science*, pages 358–365. Springer, July 2003.

4. 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 *Lecture Notes in Computer Science*, pages 6–24. Springer, 2001.

5. J. Boyland. Alias burying: Unique variables without destructive reads. *Software—Practice and Experience*, 31(6):533–553, May 2001.

6. F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal. *Pattern-Oriented Software Architecture - A System of Patterns*. John-Wiley and Sons, 1996.

7. 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.

8. E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading/MA, 1995.

9. J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java Language Specification*. Addison Wesley, 2nd edition, 2000.

10. M. Grand. *Patterns in Java*, volume 1 and 2. John Wiley & Sons, 1998 and 1999.

11. D. Gries and F. B. Schneider. Avoiding the undefined by underspecification. In J. van Leeuwen, editor, *Computer Science Today*, volume 1000 of *Lecture Notes in Computer Science*, pages 366–373. Springer, 1995.

12. 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.

13. G. T. Leavens, A. L. Baker, and C. Ruby. Preliminary design of JML: A behavioral interface specification language for Java. Technical Report 98-06z, Iowa State University, Department of Computer Science, Dec. 2004. See www.jmlspecs.org.

14. B. Meyer. Applying "design by contract". *IEEE Computer*, 25(10):40–51, Oct. 1992.

15. P. Müller. *Modular specification and verification of object-oriented programs*. Springer-Verlag New York, Inc., 2002.

16. G. Nelson. Verifying reachability invariants of linked structures. In *Proceedings of the 10th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 38–47. ACM Press, 1983.

17. J. Noble, R. Biddle, E. Tempero, A. Potanin, and D. Clarke. Towards a model of encapsulation. In *International Workshop on Aliasing, Confinement, and Ownership (IWACO)*, Darmstadt, Germany, July 2003.

18. A. Roth and P. H. Schmitt. Ensuring invariant contracts for modules in java. In *Proceedings of the ECOOP Workshop FTfJP 2004 Formal Techniques for Java-like Programs*, number NIII-R0426 in Technical Report, University of Nijmegen, pages 93–102, June 2004.

19. J. Vitek and B. Bokowski. Confined types in Java. *Software—Practice and Experience*, 31(6):507–532, 2001.

20. J. Warmer and A. Kleppe. *The Object Constraint Language: Precise Modelling with UML*. Object Technology Series. Addison-Wesley, Reading/MA, 1999.