Electronic Notes in Theoretical Computer Science 11 (1998)
URL: http://www.elsevier.nl/locate/entcs/volume11.html 12 pages

Coalgebraic Reasoning about Classes in Object-Oriented Languages

Bart Jacobs [1]

Department of Computer Science, University of Nijmegen
P.O. Box 9010, 6500 GL Nijmegen, The Netherlands
bart@cs.kun.nl

Abstract

This note briefly discusses how some of the ideas developed in the theory of coalgebras are used in a front-end tool called LOOP, developed jointly in Dresden and Nijmegen, for reasoning (with a back-end theorem prover) about classes in object-oriented languages. It will describe reasoning both about object-oriented specifications and about JAVA implementations, via examples.

1 Introduction

One of the key aspects of object-orientation is that objects, as instances of a class, have a private state, which can only be accessed and modified via the operations (usually called attributes and methods) of the class of the object. It is precisely this aspect which forms the starting point of the coalgebraic analysis of classes and objects in an object-oriented setting, see [10,5]: a class is seen as a coalgebra, and an object of a class as an element of the state space of the coalgebra. More precisely, a coalgebra is an operation of the form $c: X \to T(X)$, where X is the carrier set (or state space), and T is a functor determining the interface of the operation c. For example, $T(X)$ may be $\mathsf{int} \times X^{\mathsf{bool}}$, so that c can be identified with a pair of functions $\langle c_0, c_1 \rangle$, where $c_0: X \to \mathsf{int}$ and $c_1: X \times \mathsf{bool} \to X$. The first operation c_0 may be called an attribute because it gives some integer information about states in X, and the second operation c_1 may be called a method because it allows us to modify a state in X, given a boolean parameter. In this coalgebraic approach there is no way of constructing elements of the state space X: one can only observe or modify existing states. This is the same for (already constructed) objects.

In the coalgebraic approach to object-orientation, the operations (attributes plus methods) in a class are understood (jointly) as a single coalgebra, acting on some state space. As already mentioned, objects of this class are the elements of its state space. A coalgebraic class specification describes a class as

[1] Research Fellow of the Royal Netherlands Academy of Arts and Sciences.

some (uninterpreted) coalgebra satisfying certain assertions[2]. A coalgebraic class implementation describes a particular state space (like $X = \mathsf{int} \times \mathsf{bool}$) together with a specific coalgebra acting on this state space (possibly satisfying some assertions). Below we shall describe an example of a class specification in the language CCSL (for Coalgebraic Class Specification Language) of the LOOP tool (for Logic of Object-Oriented Programming). Also, we shall present examples of implementations, by constructing concrete coalgebras (as models of specifications).

In the theory of coalgebras there are standard notions of bisimulation and of invariant. A bisimulation is a binary relation on a state space of a coalgebra which is closed under the operations of the coalgebra. And similarly, an invariant is a unary predicate on the state space which is also closed under the operations. For instance, for the above coalgebra $c = \langle c_0, c_1 \rangle$, a relation $R \subseteq X \times X$ is a bisimulation if for all states $x, y \in X$,

$$R(x, y) \Rightarrow \begin{cases} c_0(x) = c_0(y) \\ \qquad \text{and} \\ \forall b \in \mathsf{bool}\ R(c_1(x, b), c_1(y, b)). \end{cases}$$

Two states x, y are called bisimilar if there is a bisimulation R with $R(x, y)$. Bisimilar states are observationally indistinguishable, because the operations are unable to produce an obervational difference between them. Bisimilarity need not mean actual equality of states, because coalgebraic operations give us only limited access to the state space and may leave certain internal differences undetected.

A predicate $P \subseteq X$ is an invariant with respect to the coalgebra $c = \langle c_0, c_1 \rangle$ if for all states $x \in X$,

$$P(x) \Rightarrow P(c_1(x)).$$

Such an invariant gives a property of states which, once true for a state x, remains true no matter how one modifies this state x via the available coalgebraic operations.

Notions of bisimulation and invariant are fundamental in the theory of dynamical systems, of automata, and of processes. They form a crucial ingredient of the theory of coalgebras. There, the functor (or interface) of a coalgebra determines the precise formulation of the associated notions of bisimulation and invariant. Had we taken another functor T above, say $T(X) = \mathsf{int} \times X^{\mathsf{bool}} \times X$, then the associated definitions of bisimulation and invariant (with resprect to T) would have been (slightly) different.

There are several ways to define the notion of bisimulation and invariant associated with a particular functor T (see also *e.g.* [11]). In the present setting, the inductive formulation, based on the (polynomial) structure of T,

[2] A coalgebraic specification is thus very much like an algebraic specification: the key difference lies in the operations, which are coalgebraic (going out of a state space) instead of algebraic (going into a state space), see [7] for further discussion about algebras and coalgebras.

is most relevant. It is ultimately based on ideas from categorical logic, see [4,6]. We shall describe it in some detail below.

2 The LOOP tool

Since the summer of 1997 there is a joint project between Ulrich Hensel and Hendrik Tews in Dresden and Marieke Huisman and Bart Jacobs in Nijmegen on the development of a tool (called LOOP) for reasoning about classes. This LOOP tool works as a front-end tool for a theorem prover: it reads, analyses, and transforms a class (in a certain format) into a set of logical theories. The latter can be loaded into a theorem prover, and thus provides a setting for reasoning (with support of the full power of the prover) about the original class. The theorem prover that is currently used as back-end is PVS [9], but it should be possible to produce suitable theories for other provers as well. The inputs that are accepted by the LOOP tool are (currently):

- class specifications in a (especially designed) coalgebraic class specification language CCSL. This is discussed in greater detail in [3].
- JAVA classes. This will be described more elaborately in a future publication.

We shall briefly discuss examples of the use of the LOOP tool for both inputs below.

Aside from translating the inputs, the LOOP tool also generates suitable notions of bisimulation and invariant for the specific input class (together with some associated standard results, stating *e.g.* that invariants are closed under conjunctions and under universal quantification). These definitions of bisimulation and invariant help the user in formulating and proving suitable properties about the classes under consideration.

Within the object-oriented setting there are two ways in which new classes can be constructed from old (see *e.g.* [8, Chapter 5, pages 118-119]): via inheritance (one class is a subclass of another, using the "is a" relation) and via aggregation (one class is a component of another "client" class, using the "has a" relation). Discussing these mechanisms within the LOOP tool would lead too far, but they form of course essential ingredients of the object-oriented paradigm—whatever that may be.

3 LOOP on classes in CCSL

Figure 1 gives the CCSL presentation of the flag example that is often used in hidden algebra (see *e.g.* [1]). A flag has an attribute for describing its status, and methods for setting it up or down and for reverting it. The type `Self` in this specification describes the state space, and is considered as a black box. The `is_up` attribute gives some information about states (elements of `Self`), and the methods `set_up`, `set_down` and `revert` can modify a state, but nothing is told about the interior of `Self`. This is typical of coalgebraic

```
BEGIN Flag : CLASSSPEC

  ATTRIBUTE
    is_up : Self -> bool;

  METHOD
    set_up : Self -> Self;
    set_down : Self -> Self;
    revert : Self -> Self;

  ASSERTION
    is_up_set_up :
      PVS is_up(set_up(x)) ENDPVS
    is_up_set_down :
      PVS NOT is_up(set_down(x)) ENDPVS
    is_up_revert :
      PVS is_up(revert(x)) = NOT is_up(x) ENDPVS

  CONSTRUCTOR
    new : bool -> Self;

  CREATION
    is_up_new :
      PVS FORALL(b : bool) : is_up(new(b)) = b ENDPVS

END Flag
```

Fig. 1. A flag class specification in CCSL

specification. The Flag attribute and methods are required to satisfy some obvious assertions. The latter have a name in CCSL (like `is_up_set_down`), and contain the keywords `PVS` $\cdots$ `ENDPVS`, which indicate that the middle part $\cdots$ is to be regarded as a string for the back-end proof tool PVS. Additionally there is a single constructor `new` (with a parameter), introducing an intial state.

Running the LOOP tool on this example specification produces a series of PVS theories. Space restrictions prevent us from discussing all of them, so we shall focus on the (first) theory describing the interface, and on a theory introducing the notion of bisimulation for flags. The latter will be used to actually prove a property about these flags.

In the interface theory the operations (attributes plus methods) are analysed by the LOOP tool. An interface type is formed, which is actually the functor of the underlying coalgebra. The generated PVS code is as follows.

```
FlagIFace : TYPE =
  [#
    is_up : bool ,
    set_up : Self ,
    set_down : Self ,
```

```
    revert : Self
  #]
```

The right-hand-sides of the operations (possibly after currying) are collected in a labeled product (also called record), indicated by `[# ··· #]`. The type `Self` is a parameter type, which plays the rôle of a state space. Once this (functor) type is defined, we can work with coalgebras of the form:

```
c : [Self -> FlagIFace[Self]]
```

They combine all the flag operations in a single function. The individual operations are extracted via obvious definitions:

```
is_up(c) : [Self -> bool] = (LAMBDA(x: Self) : is_up(c(x))) ;
set_up(c) : [Self -> Self] = (LAMBDA(x: Self) : set_up(c(x))) ;
set_down(c) : [Self -> Self] = (LAMBDA(x: Self) : set_down(c(x))) ;
revert(c) : [Self -> Self] = (LAMBDA(x: Self) : revert(c(x))) ;
```

Hence, operations like `revert` are always operations with respect to some coalgebra `c`, as indicated by the dependence on `c` in these definitions.

The generated bisimulation theory in PVS introduces the notions of bisimulation and bisimilarity with respect to two flag coalgebras `c1` and `c2` with respective state spaces `Self1` and `Self2`. First, the `FlagIFace` functor is lifted from types to relations (following [4,6]) in the following definition (based on the structure of the functor `FlagIFace`):

```
FlagRel : [[[Self1 , Self2] -> bool] ->
           [[FlagIFace[Self1] , FlagIFace[Self2]] -> bool]] =
  (LAMBDA(R: [[Self1 , Self2] -> bool]) :
    (LAMBDA(rec1: FlagIFace[Self1] , rec2: FlagIFace[Self2]) :
      (is_up(rec1) = is_up(rec2)) AND
      (R(set_up(rec1) , set_up(rec2))) AND
      (R(set_down(rec1) , set_down(rec2))) AND
      (R(revert(rec1) , revert(rec2)))))
```

Then, bisimulation and bisimilarity are defined for coalgebras,

```
c1 : VAR [Self1 -> FlagIFace[Self1]]
c2 : VAR [Self2 -> FlagIFace[Self2]]
```

as:

```
bisimulation?(c1 , c2) : [[[Self1 , Self2] -> bool] -> bool] =
  (LAMBDA(R: [[Self1 , Self2] -> bool]) :
    FORALL ( x1: Self1 , x2: Self2) :
      (R(x1 , x2)) IMPLIES (FlagRel(R)(c1(x1) , c2(x2))))

bisim?(c1 , c2) : [[Self1 , Self2] -> bool] =
  (LAMBDA(x1: Self1 , x2: Self2) :
    EXISTS ( R: [[Self1 , Self2] -> bool]) :
      (bisimulation?(c1 , c2)(R)) AND (R(x1 , x2)))
```

Additionally, for a single coalgebra, notions of bisimulation and bisimilarity are generated (using the above definitions for two coalgebras):

```
c : VAR [Self -> FlagIFace[Self]]
```

```
bisimulation?(c) : [[[Self , Self] -> bool] -> bool] =
  bisimulation?(c , c)

bisim?(c) : [[Self , Self] -> bool] =
  bisim?(c , c) ;
```

The assertions in the flag specification in Figure 1 are collected in a single predicate on a coalgebra `c` as above:

```
FlagAssert?(c) : bool = ''The Flag Assertions''
```

Hence, if we wish to develop the theory of flags, we simply declare a coalgebra variable satisfying `FlagAssert?`, and start proving things (in PVS). As an example, we consider the standard result of hidden algebraists, namely that the result of reverting a flag twice is bisimilar (or behaviourally equal, as they call it) to the original flag. In our setting this corresponds to the statement:

```
bisim?(c)(x, revert(c)(revert(c)(x)))
```

where `c` is assumed to be a coalgebra satisfying `FlagAssert?`, and `bisim?` is the automatically generated definition for bisimilarity. We consider two ways of proving this in PVS.

(i) Simply expanding the definitions leads to the requirement to prove:

```
EXISTS (R: [[Self, Self] -> bool]):
  (bisimulation?(c!1, c!1)(R)) AND
  (R(x!1, revert(c!1)(revert(c!1)(x!1))))
```

where `c!1` and `x!1` are arbitrary choices (made by PVS) for the above coalgebra `c` and state `x`. The instantion to take for `R` is:

```
{x,y : Self | is_up(c!1)(x) = is_up(c!1)(y)}
```

The resulting proof obligations are easily discharged using the flag assertions.

(ii) A slightly neater way is first to establish a result characterising bisimilarity for flags:

```
bisim_char : LEMMA
  bisim?(c)(x, y) IFF is_up(c)(x) = is_up(c)(y)
```

The (only-if) part is easy, and for the (if)-part one uses the same instantiation as above. The required bisimilarity result about reverting twice is then an easy consequence of this lemma.

To conclude we briefly describe two implementations of the flag specification, by constructing concrete coalgebras. The first implementation uses booleans as state space, with coalgebra:

```
final : [bool -> FlagIFace[bool]] =
  LAMBDA(b : bool) :
    (#
      is_up := b ,
      set_up := TRUE ,
      set_down := FALSE ,
      revert := NOT b
```

```
    #)
```

This is probably the most obvious implementation—and is in fact the final coalgebra satisfying the assertions. Notice that bisimilar states in this model are actually equal—a property that is typical for final models. We also construct a model where bisimilar states need not be equal. This second implementation is a "history" model, in which one records the values of a flag. Hence two states may be bisimilar (have equal `is_up` values), but be quite different because they have different histories. The state space of this history model is the type `list[bool]` of lists of booleans, with coalgebra:

```
history : [list[bool] -> FlagIFace[list[bool]]] =
  LAMBDA(h : list[bool]) :
    (#
      is_up := CASES h OF
                 null : FALSE,
                 cons(b, k) : b
               ENDCASES ,
      set_up := cons(TRUE, h) ,
      set_down := cons(FALSE, h) ,
      revert := CASES h OF
                  null : cons(TRUE, null),
                  cons(b, k) : cons(NOT b, cons(b, k))
                ENDCASES
    #)
```

Both the following results

```
final_assert : LEMMA
  FlagAssert?(final)
history_assert : LEMMA
  FlagAssert?(history)
```

establishing that these implementations satisfy the flag assertions, are easy to prove. In fact, PVS does all the work automatically via the single (GRIND) command.

Finally we remark that for convenience we have ommitted the constructor `new` from our discussion. It may be added easily to the above implementations: in the final model simply as the identity function `bool` $\rightarrow$ `bool`, and in the history model as the function `bool` $\rightarrow$ `list[bool]` sending `b` to `cons(b, null)`, for example.

More information about the specification language CCSL and the working of the LOOP tool may be obtained from [3]. Most of the development of the LOOP tool on CCSL classes takes place in Dresden.

4 LOOP on classes in JAVA

As mentioned, the LOOP tool can also translate JAVA classes into PVS code. This development is mostly done in Nijmegen, with contributions from Joachim van den Berg and Martijn van Berkum (both graduate students).

```
class Parent {

  int i;

  void base() {
    i = 4;
  }
}

class Child extends Parent {

  int i;
  int j;

  void deriv() {
    j = 1;
    base();
  }
}

class GrandChild extends Child {

  void base() {
    i = 8;
  }
}
```

Fig. 2. Late binding example in JAVA

The way LOOP operates on JAVA classes is very similar to how it operates on CCSL classes, and much of the implementation is used for both.

Basically, what LOOP does on a JAVA class is:

(i) extract the interface (or functor) underlying the fields (JAVA speak for attributes) and methods of the class;

(ii) analyse the inheritance and component structure;

(iii) transform the method definitions into suitable equations;

(iv) generate PVS theories on the basis of this information, including definitions of invariant and bisimulation which are appropriate for the input JAVA class.

For example, for the third step, consider in a JAVA Point class a move method with definition,

```
void move(int da, int db) {
  fst = fst + da;
  snd = snd + db;
}
```

where `fst`, `snd` are integer fields describing the first and second coordinate of a point. This method definition is translated into an assertion

```
FORALL(x : Self, da : int, db : int) :
  move(c)(x, da, db) = (
    fst_becomes(c)(fst(c) + da) #
    snd_becomes(c)(snd(c) + db)
    )(x)
```

where `fst_becomes` and `snd_becomes` are assignment operations generated for `fst` and `snd`. The sharp operation `#` (defined in a PVS prelude to this translation) is the translation of the composition operation ; of JAVA. An obvious property that we can prove about such a method is:

```
fst(c)(move(c)(x, da, db)) = fst(c)(x) + da
```

This requires some basic reasoning about assignments.

But things are not always so trivial. Consider for example the series of JAVA classes `Parent` – `Child` – `GrandChild` in Figure 2. The declaration `int i` in `Child` "hides" the `i` from `Parent`, see [2, Section 8.3], but running `deriv` in `Child` will affect `i` in `Parent`, and not `i` in `Child`. In contrast, running `deriv` in `GrandChild` will affect `i` in `Child`, but not `i` in `Parent`, due to the late binding mechanism which determines that within the `GrandChild` class `deriv` will call the (redefined) `base` method from `GrandChild`.

The challenge is to prove the right values of the `i`'s and `j` after running `deriv` in `Child` and in `GrandChild`. In the LOOP translation of these JAVA classes into PVS we first have to show that the method `deriv` terminates normally (and does not hang or throw and exception). Then we can express the values of the variables in the resulting state after `deriv` in terms of the orignal values as follows. For a `Child` coalgebra `c` this is expressed in the following result.

```
Child_deriv : LEMMA
  norm??(deriv(c)(x))
    AND
  i(c)(ns?(deriv(c)(x))) = i(c)(x)
    AND
  j(c)(ns?(deriv(c)(x))) = 1
    AND
  Super_i(c)(ns?(deriv(c)(x))) = 4
```

The first assertion in the conjunction states that running `deriv(c)` in an arbitrary state `x` is normal (*i.e.* terminates normally). The next three statements describe the values of the variables `i(c)`, `j(c)` and `Super_i(c)` (*i.e.* `i` from the super class `Parent` of child coalgebra `c`) when evaluated in the normal state (accessed by `ns?`) resulting from running `deriv(c)`.

For a `GrandChild` coalgebra `gc` the required result is:

```
GrandChild_deriv : LEMMA
  norm??(deriv(gc)(x))
    AND
  i(gc)(ns?(deriv(gc)(x))) = 8
    AND
  j(gc)(ns?(deriv(gc)(x))) = 1
    AND
```

```
class Counter {

  private int max;
  private int val;

  int maximum() {
    return max;
  }

  void next() {
    if ( val < max ) {
      val = val + 1;
    } else {
      val = 0;
    }
   }

  void clear() {
    val = 0;
  }

  Counter(int n) {
    max = n;
  }

}
```

Fig. 3. A Counter class in JAVA

```
Super_i(gc)(ns?(deriv(gc)(x))) = Super_i(gc)(x)
```

We succeed in proving both these lemmas, using the PVS files generated by the LOOP tool (together with certain special prelude PVS files which define the setting for the translation of JAVA classes). Even more, these lemmas are proved entirely by automatic rewriting, so that basically only two proof commands are needed: (load-rewrite-rules) followed by (do-rewrite). Much effort in this project goes into formulating and (automatically) generating suitable rewrite rules, so that "simple" results like the above ones can be handled by only a few proof commands in PVS. Of course, more complicated results involving for and while loops, require more intelligence and interaction from the verifier.

Late binding in JAVA, as occuring the in the example classes in Figure 2, is handled by the LOOP tool by suitably repeating method definitions from superclasses in subclasses.

Once again, all this is based on a coalgebraic semantics for JAVA classes. As a final remark we emphasise that appropriate coalgebraic definitions of invariance and bisimilarity are generated for each translated JAVA class. Consider for example the simple counter class in Figure 3. Feeding this class into the LOOP tool leads to a series of PVS theories in which a suitable invariant definition is given:

```
invariant?(c) : [[Self -> bool] -> bool] =
  (LAMBDA(P: [Self -> bool]) :
    FORALL ( x: Self) : (P(x)) IMPLIES (CounterPred(P)(c(x))))
```

where `CounterPred` is a lifting of the `CounterIFace` functor to predicates. The definition of invariance is generated in such a way that invariant predicates are closed under the *publicly* available methods—which are in this case: `maximum`, `next` and `clear`, but not assignments for the private variables `max` and `val`.

Then a user may wish to prove an invariance result like:

```
val_below_max(c) : [Self -> bool] =
  LAMBDA(x : Self) :
    0 <= max(c)(x) AND 0 <= val(c)(x) AND val(c)(x) <= max(c)(x)

val_below_max_inv : LEMMA
  invariant?(c)(val_below_max(c))
```

What this amounts to is showing that if this predicate `val_below_max(c)` holds in a state `x`, then it still holds if one of the methods `maximum(c)`, `next(c)` or `clear(c)` of a Counter coalgebra `c` is applied to `x`. This should be obvious, and is easy to prove.

Acknowledgement

Thanks to Marieke Huisman for her feedback on the first version of this note.

References

[1] J.A. Goguen and G. Malcolm. An extended abstract of a hidden agenda. In J. Meystel, A. Meystel, and R. Quintero, editors, *Proceedings of the Conference on Intelligent Systems: A Semiotic Perspective*, pages 159–167. Nat. Inst. Stand. & Techn., 1996.

[2] J. Gosling, B. Jay, and G. Steele. *The Java Language Specification*. Addison-Wesley, 1996.

[3] U. Hensel, M. Huisman, B. Jacobs, and H. Tews. Reasoning about classes in object-oriented languages: Logical models and tools. In Ch. Hankin, editor, *European Symposium on Programming*, Lect. Notes Comp. Sci. Springer, Berlin, 1998.

[4] C. Hermida and B. Jacobs. Structural induction and coinduction in a fibrational setting. Inf. & Comp., to appear, 1998.

[5] B. Jacobs. Objects and classes, co-algebraically. In B. Freitag, C.B. Jones, C. Lengauer, and H.-J. Schek, editors, *Object-Orientation with Parallelism and Persistence*, pages 83–103. Kluwer Acad. Publ., 1996.

[6] B. Jacobs. Invariants, bisimulations and the correctness of coalgebraic refinements. In M. Johnson, editor, *Algebraic Methodology and Software*

Technology, number 1349 in Lect. Notes Comp. Sci., pages 276–291. Springer, Berlin, 1997.

[7] B. Jacobs and J. Rutten. A tutorial on (co)algebras and (co)induction. *EATCS Bulletin*, 62:222–259, 1997.

[8] B. Meyer. *Object-Oriented Software Construction*. Prentice Hall, 2^{nd} rev. edition, 1997.

[9] S. Owre, J.M. Rushby, N. Shankar, and F. von Henke. Formal verification for fault-tolerant architectures: Prolegomena to the design of PVS. *IEEE Trans. on Softw. Eng.*, 21(2):107–125, 1995.

[10] H. Reichel. An approach to object semantics based on terminal co-algebras. *Math. Struct. in Comp. Sci.*, 5:129–152, 1995.

[11] J. Rutten. Universal coalgebra: a theory of systems. CWI Report CS-R9652, 1996.