Reasoning about Classes in Object-Oriented Languages: Logical Models and Tools

ULRICH HENSEL[1] MARIEKE HUISMAN[2] BART JACOBS[2] HENDRIK TEWS[1]
[1] Inst. Theor. Informatik, TU Dresden, D-01062 Dresden, Germany.
{hensel,tews}@tcs.inf.tu-dresden.de
[2] Dep. Comp. Sci., Univ. Nijmegen,
P.O. Box 9010, 6500 GL Nijmegen, The Netherlands.
{marieke,bart}@cs.kun.nl

Abstract. A formal language CCSL is introduced for describing specifications of classes in object-oriented languages. We show how class specifications in CCSL can be translated into higher order logic. This allows us to reason about these specifications. In particular, it allows us (1) to describe (various) implementations of a particular class specification, (2) to develop the logical theory of a specific class specification, and (3) to establish refinements between two class specifications.
We use the (dependently typed) higher order logic of the proof-assistant PVS, so that we have extensive tool support for reasoning about class specifications. Moreover, we describe our own front-end tool to PVS, which generates from CCSL class specifications appropriate PVS theories and proofs of some elementary results.

1 Introduction

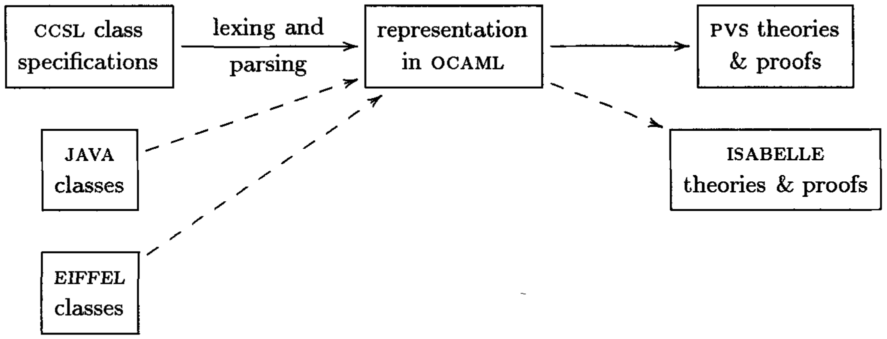
During the last two decades, object-orientation has established itself in analysis, design and programming. At this moment, C^{++} and JAVA are probably the most popular object-oriented programming languages. Despite this apparent success, relatively little work has been done on formal (logical) methods for object-oriented programming. One of the reasons, we think, is that there is no generally accepted formal computational model for object-oriented programming. Such a model is needed as domain of reasoning.

One such formal model has recently emerged in the form of "coalgebras" (explicitly *e.g.* in [21]). It should be placed in the tradition of behavioural specification, see also [6, 8, 4]. Coalgebras are the formal duals of algebras, see [14] for background information. They consist of a (hidden) state space—typically written as Self in this context—together with several operations (or methods) acting on Self. These operations may be attributes giving some information about objects (the elements of Self), or they may be procedures for modifying objects. All access to elements of Self should go via the operations of the coalgebra. In contrast, elements of abstract data types represented as algebras can only be built via the "constructor" operations (of the algebra). We consider coalgebras together with initial states as classes, and elements of the carrier Self of a coalgebra as (states of) objects of the class.

For verification purposes involving coalgebraic classes and objects we are interested in the observable behavior and not in the concrete representation of elements of Self. A behavior of an object in this context is the objects reaction pattern, *i.e.* what we can observe via the attributes after performing internal computations triggered by pressing procedure buttons. This naturally leads to notions like bisimilarity (indistinguishability of objects via the available operations) and invariance.

Based on coalgebras, a certain format has been developed for class specifications, see [21, 11, 10]. This format typically consists of three sections, describing the class specifications' methods, assertions, and creation-conditions—which hold for newly created objects. We have developed this format into a formal language CCSL, for *Coalgebraic Class Specification Language*, which will be sketched below. *Ad hoc* representations of these class specifications in the higher order logic of the proof-tool PVS [18, 17] have been used in [12, 13] to reason about such classes—notably for refinement arguments. Further experiments with formal reasoning about classes and objects have led us to a general representation of CCSL class specifications in higher order logic. Below we explain this model (in the logic of PVS), and also a (preliminary version of a) front-end tool that we use for generating such models from class specifications.

The code for this tool (called LOOP for *Logic of Object-Oriented Programming*) is written in OCAML [22]. It basically performs three consecutive steps: it first translates a CCSL class specification into some representation in OCAML; this representation is then internally analysed and finally transformed into PVS theories and proof. The generated PVS file contains several theories describing the representation of the class specification, via appropriate definitions and associated lemmas (*e.g.* stating that bisimilarity is an equivalence relation). Another file that is generated by our tool contains instructions for proofs of the lemmas in the PVS file. The architecture of our tool allows for easy extensions, *e.g.* to accept JAVA [7] or EIFFEL [16] classes, or to generate files for other proof assistants such as ISABELLE [19]. The diagram below describes (via the solid lines) what our tool currently does, see Section 5 for some more details. The dashed lines indicate possible future extensions.



The idea behind the dashed lines on the left is that classes in actual programming languages should lead to executable class specifications, about which one can reason. We have made some progress—which will be reported elsewhere—in reasoning about JAVA classes in this setting. Here we concentrate on the upper solid lines.

The paper is organised as follows. We start in Sections 2 and 3 with an elaborate discussion of two examples, involving a class specification of a register in which one can store data at addresses, and a subclass specification of a bounded register, in which writing is only allowed if the register is not full. This involves overriding of the original write method. Then, in Section 4 we discuss some further aspects of the way that we model class specifications and that we reason about them. Finally, in Section 5, we describe the current stage of the implementation of our front-end tool.

We shall be using the notation of PVS's higher order logic when we describe our models. It is largely self-explanatory, and any non-standard aspects will be explained as we proceed. The LOOP front-end tool that will be described in Section 5 is still under development. Currently, it does the basics of the translation from CCSL class specification to PVS theories and proofs (without any fancy GUI). It may take some time before it reaches a stable form. Instead of elaborating implementation details, this paper focuses on the basic ideas of our models.

2 A simple register: specification and modeling

We start by considering a simple register, which can store data at an address. It contains read, write and erase operations, satisfying some obvious requirements. It is described in our coalgebraic class specification language CCSL in Figure 1. The type constructor $A \mapsto \mathrm{Lift}[A]$ adds a bottom element 'bot' to a type A, and keeps all elements a from A as $\mathrm{up}(a)$. A *total* function $B \to \mathrm{Lift}[A]$ may then be identified with a *partial* function $B \rightharpoonup A$. We use square brackets $[A_1, \ldots, A_n]$ for the Cartesian product $A_1 \times \cdots \times A_n$.

```
Begin Register[ Data : Type, Address : Type ] : ClassSpec
  Method      read  : [Self, Address] -> Lift[Data];
              write : [Self, Address, Data] -> Self;
              erase : [Self, Address] -> Self
  Assertion   read_write : PVS FORALL(a,b : Address, d : Data)
                read(write(x, a, d), b) =
                  IF a = b THEN up(d) ELSE read(x, b) ENDIF ENDPVS
              read_erase : PVS FORALL(a,b : Address) :
                read(erase(x, a), b) =
                  IF a = b THEN bot ELSE read(x, b) ENDIF ENDPVS
  Constructor new : Self
  Creation    read_new : PVS FORALL(a:Address) : read(new,a) = bot ENDPVS
End Register
```

Fig. 1. A register class specification in CCSL

The types Data and Address are parameters in this specification, which can be suitably instantiated in a particular situation. What is coalgebraic about such specifications is that all methods act on Self, *i.e.* have Self as one of their input types[1]. Usually this type Self is not written explicitly in object-oriented languages, but it is there implicitly as the receiver of method invocations. The constructor section declares **new** as a constructor (without parameters) for creating a new register. Notice that assertions and creation-conditions have names. The PVS and ENDPVS tags are used to delimit strings, which are basically boolean expressions in PVS[2]. It is assumed that **x** is a variable in Self.

In order to reason (with PVS tool support) about such a Register class specification, we first model it in the higher order logic of PVS. This is what our LOOP tool does automatically. It generates several PVS theories to capture this specification. Space restrictions prevent us from discussing all these theories in detail, so we concentrate on the essentials.

The first step is to introduce a (single) type which captures the interface of a class specification, via a labeled product. For Register, this is done in the following PVS theory.

```
RegisterInterface[ Self, Data, Address : TYPE ] : THEORY
BEGIN
  RegisterIFace : TYPE = [# read  : [Address -> Lift[Data]],
                            write : [Address, Data -> Self],
                            erase : [Address -> Self] #]
END RegisterInterface
```

The square brace notation $[A_1, \ldots, A_n \to B]$ is used in PVS for the type of (total) functions with n inputs from $A_1, \ldots, A_n$ and with result in B. Notice that in the types of the operations in this interface the input type Self is omitted[3]. This is intended: a crucial step in our approach is that we use *coalgebras* of the form

```
c : [Self -> RegisterIFace[Self, Data, Address]]
```

as models of the method section of the Register specification, with Self as state space. The individual methods can be extracted from such a coalgebra c via the definitions:

```
read(c) : [Self, Address -> Lift[Data]] =
  LAMBDA(x : Self, a : Address) : read(c(x))(a)
write(c) : [Self, Address, Data -> Self] =
  LAMBDA(x : Self, a : Address, d : Data) : write(c(x))(a,d)
erase(c) : [Self, Address -> Self] =
  LAMBDA(x : Self, a : Address) : erase(c(x))(a)
```

Thus the individual methods of a class can be extracted from such a single coalgebra c.

[1] A bit more precisely, the methods can all be written, possibly using currying, of the form Self $\to F_i(\mathrm{Self})$; and they can be combined into a single operation Self $\to F_1(\mathrm{Self}) \times \cdots \times F_n(\mathrm{Self})$.

[2] Our front-end tool simply passes the string in PVS ... ENDPVS on to the PVS tool, where it is parsed and typechecked.

[3] Categorically, the type RegisterIFace captures the functor associated with the signature of operations in the class specification, see [14].

Next, our formalisation deals with invariants and bisimulations. These are special kinds of predicates and relations on Self which are suitably closed under the above operations. For example, an invariant $P \subseteq$ Self with respect to a Register coalgebra c satisfies, by definition:

$$P(x) \Rightarrow \begin{cases} \forall a\colon \text{Address}, d\colon \text{Data}.\ P(\text{write}(c)(x, a, d)) \\ \forall a\colon \text{Address}.\ P(\text{erase}(c)(x, a)). \end{cases}$$

A bisimulation *w.r.t.* c is a relation $R \subseteq$ Self $\times$ Self satisfying:

$$R(x, y) \Rightarrow \begin{cases} \forall a\colon \text{Address}.\ \text{read}(c)(x, a) = \text{read}(c)(y, a) \\ \forall a\colon \text{Address}, d\colon \text{Data}.\ R(\text{write}(c)(x, a, d), \text{write}(c)(y, a, d)) \\ \forall a\colon \text{Address}.\ R(\text{erase}(c)(x, a), \text{erase}(c)(y, a)). \end{cases}$$

Bisimilarity `bisim?` is then the greatest bisimulation relation. Interestingly, these notions of invariant and bisimulation are completely determined by the class interface RegisterIFace. They are generated automatically (by our tool) by induction on the structure of the types in the interface, based on liftings of these types to predicates and relations, as introduced in [9] (see also [13]). These PVS theories about invariants and bisimulations contain several standard lemmas (stating *e.g.* that invariants are closed under finite conjunctions $\wedge$ and universal quantification $\forall$), for which proof instructions are generated automatically (again using induction).

The next theory RegisterSemantics deals with the assertions and creation-conditions. The two assertions in Figure 1 are translated into two predicates on the carrier type Self of a Register coalgebra c: [Self $\rightarrow$ RegisterIFace[Self, Data, Address]]. Assuming that x is a variable of type Self, we generate:

```
read_write?(c)(x) : bool =
  FORALL(a,b : Address, d : Data) : read(c)(write(c)(x, a, d), b) =
    IF a = b THEN up(d) ELSE read(c)(x, b) ENDIF
read_erase?(c)(x) : bool =
  FORALL(a,b : Address) : read(c)(erase(c)(x, a), b) =
    IF a = b THEN bot ELSE read(c)(x, b) ENDIF
```

For convenience, these predicates are collected in a single predicate:

```
RegisterAssert?(c) : bool =
  FORALL(x : Self) : read_write?(c)(x) AND read_erase?(c)(x)
```

Similarly, we put the creation-condition in a predicate

```
RegisterCreate?(c) : PRED[Self] =
  {x : Self | FORALL(a : Address) : read(c)(x, a) = bot}
```

At this stage we are able to say what actually constitutes a class implementation that satisfies a class specification as in Figure 1: it is a coalgebra c: [Self $\rightarrow$ RegisterIFace[Self, Data, Address]] satisfying the predicate RegisterAssert?, together with some element new: Self satisfying the predicate RegisterCreate?(c). This is formalised in the following theory using a (dependent!) labeled product.

```
RegisterClassStructure [Self, Data, Address : TYPE] : THEORY
BEGIN
  IMPORTING RegisterSemantics[Self, Data, Address]
  RegisterClass : TYPE = [# clg : (RegisterAssert?),
                            new : (RegisterCreate?(clg)) #]
END RegisterClassStructure
```

The notation `(P)` for a predicate `P:[A -> bool]` on `A:TYPE` is used in PVS as an abbreviation for the predicate subtype `{x:A|P(x)}`. A class thus consists of a state space Self with appropriate operations (combined in a coalgebra `clg` on Self) and with an appropriate constructor **new**. An object of such a class is then simply an inhabitant of the state space Self. Thus, in the way that we model classes and objects, the methods are part of the class, and not of the object. This is called the delegation implementation, in contrast to the embedding implementation, where the operations are part of the object, see [1, Sections 2.1 and 2.2].

Once we have all this settled, we can start reasoning about the class specification. The two things we can do immediately are: (1) describing an implementation of the specification, and (2) developing its theory. Both are user tasks: the tool only provides theory frames which the user can fill in. We give a sketch of what can be done.

As to (1), it is a wise strategy to write out an implementation, immediately after finishing the specification. It is notoriously hard to write "good" specifications which capture the informal description of the matter in question and, at the same time, are consistent in the logic used. This is sometimes called the "ground problem". Usually, specialists have a good understanding of a particular implementation. Once this implementation is formally written out it can be checked against the assertions and creation-conditions.

For example, for the Register class specification, an obvious implementation describes registers as partial functions from addresses to data. This can be done via the Lift[-] type constructor, and yields as state space:

```
FunctionSpace : TYPE = [Address -> Lift[Data]]
```

This type can be equipped with a suitable coalgebra structure and a constructor:

```
c : [FunctionSpace -> RegisterIFace[FunctionSpace, Data, Address]] =
  LAMBDA(f : FunctionSpace) :
    (# read := LAMBDA(a : Address) : f(a),
       write := LAMBDA(a : Address, d : Data) : f WITH [(a) := up(d)],
       erase := LAMBDA(a : Address) : f WITH [(a) := bot] #)
new : FunctionSpace = LAMBDA(a : Address) : bot
```

(The notation `g WITH [(y) := z]` is an abbreviation for `LAMBDA x : IF x = y THEN z ELSE g(x) ENDIF`.) This coalgebra structure on the state space FunctionSpace clearly captures our intuition, and it is not hard to prove that both propositions RegisterAssert?(c) and RegisterCreate?(c)(new) hold. Actually, PVS can prove both of them with a single command, (GRIND).

Of course, we can also define other implementations. For example, one can define an implementation in which the sequence of operations applied to an object is recorded for each address. This can be done by taking as state space:

```
HistorySpace : TYPE = [Address -> list[Lift[Data]]]
```

The implementation of the methods and constructor on this state space is left to the interested reader. Again, (GRIND) in PVS proves that the assertions and creation-conditions hold (for our implementation).

When class specifications are used as components in other classes (*e.g.* via class-valued attributes, see Section 4) we need a model for them. Obvious choices for a model are (1) an arbitrary, so-called "loose" model and (2) a final model. Both are generated. Once we know that our class specification has a non-trivial model (and hence that it is consistent) we can safely postulate the existence of a loose model. A final model enables the use of subclasses for components, but its existence is an open question in presence of binary methods. Due to a lack of space, only the loose model is described here. It has the following form.

```
LooseRegisterClass[Data, Address : TYPE] : THEORY
BEGIN
  LooseRegisterType : TYPE
  IMPORTING RegisterClassStructure[LooseRegisterType, Data, Address]
  loose_Register_existence : AXIOM
    EXISTS(cl : RegisterClass) : TRUE
  LooseRegisterClass : RegisterClass
END LooseRegisterClass
```

In this theory the existence of an arbitrary model of the class specification is guaranteed via an axiom. In principle this can be dangerous, because it may lead to inconsistencies. However, as long as a non-trivial implementation has been given (earlier) by hand, there is no such danger. The type LooseRegisterType in this theory is simply postulated, and we know nothing about its internal structure. This ensures that when this model is used as a component in another class, no internal details can be accessed (simply because there are no such details).

We turn to the second way to reason about a (translated) specification. Our tool generates an almost empty PVS theory frame called RegisterUserTheory. This theory starts by declaring a coalgebra structure c satisfying the predicate RegisterAssert?, together with a constructor satisfying the creation-condition RegisterCreate?(c). Under these assumptions a user can start proving various logical consequences of the assertions in the class specification. For example, a useful proposition that can be proved in RegisterUserTheory is the following characterisation of bisimilarity.

```
bisim_char : LEMMA
  bisim?(c)(x,y) IFF FORALL(a : Address) : read(c)(x,a) = read(c)(y,a)
```

This expresses that two objects (or states) x, y: Self are bisimilar (*i.e.* indistinguishable) *w.r.t.* the assumed (arbitrary) model c if and only if they give the same read output at each address. Intuitively this may be clear: if we cannot see a difference between two objects via reading, then using a write or erase will not create a difference between these objects (because a read after a write or erase is completely determined by the Register assertions).

Using this characterization, it is easy to prove, for example,

```
write_commutation : LEMMA
  FORALL(a,b : Addresses, d,e : Data) : a /= b IMPLIES
    bisim?(c)(write(c)(write(c)(x, a, d), b, e),
              write(c)(write(c)(x, b, e), a, d))
```

This result says that one can exchange write operations at different addresses. Notice that we are careful in only stating that the outcomes are bisimilar, and not necessarily equal. We avoid the use of equality of objects/states, since we regard these as hidden, and we restrict access to (public) methods. In addition, the use of bisimilarity entails that the results that we prove also hold in implementations where bisimilar states need not be (internally) equal, like in the above HistorySpace model. There we can have equal reads at all addresses in two states, even though the histories of these states are quite different. Hence such states are bisimilar, but internally different.

At the end, it may be instructive to compare this coalgebraic way of combining methods, with the approach taken in [1] (explicitly *e.g.* in Section 8.5.2). There the methods of a class are combined in a slightly different manner, namely in a labeled product, called "trait type":

$$\begin{aligned}\text{RegisterTrait} = [\#\ & \text{read: Self} \to [\text{Address} \to \text{Lift[Data]}],\\ & \text{write: Self} \to [\text{Address, Data} \to \text{Self}],\\ & \text{erase: Self} \to [\text{Address} \to \text{Self}]\ \#]\end{aligned}$$

What we do is basically the same, except that our methods are combined "coalgebraically", with the common input type Self on the outside. What is called a "class type" in [1] is such a "trait type" together with a constructor new, see the RegisterClass type above. Thus, when it comes to interfaces, there is no real difference between our approach and the one in [1]. But we go further in two essential ways: (a) we restrict the methods and constructors so that they satisfy certain requirements (given in the assertions and creation-conditions in the specification), and (b) we (automatically) generate appropriate notions of invariance and bisimilarity for (the interface of) each class specification, and use them systematically in reasoning about these specifications.

3 A bounded register: inheritance and overriding

Having described an implementation for the Register class specification—and developed part of its theory—we now introduce a new class specification BoundedRegister by inheritance. A bounded register is a subclass of a register, which overrides the write operation and defines a new attribute count. A bounded register can only store a limited number of data elements, and the count attribute is

```
Begin BoundedRegister[ Data : Type, Address : Type, n : nat ] : ClassSpec
 Inherit from Register[Data,Address]
 Method      write : [Self,Address,Data] -> Self;
             count : Self -> nat
 Assertion   override_write_def : PVS FORALL(a : Address, d : Data) :
               bisim?(write(x, a, d), IF count(x) < n OR up?(read(x,a))
                                      THEN super_write(x, a, d)
                                      ELSE x
                                      ENDIF) ENDPVS
             count_super_write : PVS FORALL(a : Address, d : Data) :
               count(super_write(x, a, d)) = IF bot?(read(x,a))
                                             THEN count(x) + 1
                                             ELSE count(x)
                                             ENDIF ENDPVS
             count_erase : PVS FORALL(a : Address) :
               count(erase(x, a)) = IF bot?(read(x, a))
                                    THEN count(x)
                                    ELSE max(0, count(x) - 1)
                                    ENDIF ENDPVS
 Constructor new : Self
 Creation    count_new : PVS count(new) = 0 ENDPVS
End BoundedRegister
```

Fig. 2. A bounded register class specification in CCSL

used to keep track of how much data is currently stored. When the bounded register is full (*i.e.* when its count is above a certain number *n* given as parameter), a write operation does not have any effect; otherwise it acts as the write operation from the superclass Register. Further, the read and erase operations from Register are used without modification. A CCSL class specification of a bounded register is given in Figure 2. The predicates `bot?` and `up?` on `Lift[Data]` tell us whether an element `x : Lift[Data]` is `bot` or `up(d)`, for some `d : Data`.

Again, our tool generates several PVS theories from this specification. This section will discuss the essential consequences the use of inheritance (in combination with overriding) has on the generated theories.

We model inheritance by letting the interface of the BoundedRegister not only contain the operations write and count, but also the superclass as a field (super_Register). This enables access to the methods of the superclass.

```
 BoundedRegisterIFace : TYPE =
  [# super_Register : RegisterIFace[Self, Data, Address],
     write : [Address, Data -> Self],
     count : nat #]
```

Now we provide access not only to the individual methods of the BoundedRegister class but also to the methods from the superclass, via the following definitions.

```
 c : VAR [Self -> BoundedRegisterIFace[Self, Data, Address]]

 super_Register(c) : [Self -> RegisterIFace[Self, Data, Address]] =
   LAMBDA (x:Self) : super_Register(c(x))
 read(c) : [[Self, Address] -> Lift[Data]] =
    LAMBDA (x:Self, a:Address) : read(super_Register(c(x)))(a)
  super_write(c) : [[Self, Address, Data] -> Self] =
    LAMBDA (x:Self, a:Address, d:Data) : write(super_Register(c(x)))(a, d)
  write(c) : [[Self, Address, Data] -> Self] =
    LAMBDA (x:Self, a:Address, d:Data) : write(c(x))(a, d)
  erase(c) : [Self, Address -> Self] =
    LAMBDA(x : Self, a : Address) : erase(super_register(c(x)))(a)
  count(c) : [Self -> nat] =
    LAMBDA(x : Self) : count(c(x))
```

Via these explicit definitions, all methods of superclasses can be used in subclasses. The number of such definitions may be considerable when there are high inheritance trees, but our tool generates all of them automatically. In fact, this is one of the reasons for developing such a tool.

The write operation in the subclass specification in Figure 2 also occurs in the superclass. This double occurrence is used to signal overriding. Our tool recognizes it, and generates as a result two write operations. A "direct" one from the current subclass (simply called `write`) and an "indirect" one from the superclass (called `super_write`). Notice that the coalgebra c—used as variable in this theory—combines both the structure of the subclass and the superclass.

The theories about invariants and bisimulations are generated incrementally, *i.e.* they extend the predicates and relations on Register with appropriate clauses for the additional methods of the subclass.

The assertions and creation-conditions of BoundedRegister are translated into PVS predicates, just as in the Register example. The resulting predicate BoundedRegisterAssert? combines these assertions with the assertions in RegisterAssert?. The predicate BoundedRegisterCreate? similarly combines the new creation-conditions with the "super" creation-conditions from Register. This implies that, although we override a method, we can still expect the superclass to behave as specified.

```
  BoundedRegisterAssert?(c) : bool =
    RegisterAssert?[Self, Data, Address](super_register(c))
      AND FORALL(x : Self) : override_write_def?(c)(x)
                              AND count_super_write?(c)(x)
                              AND count_erase?(c)(x)
  BoundedRegisterCreate?(c) : PRED[Self] =
    {x : Self | count(c)(x) = 0
       AND RegisterCreate?[Self, Data, Address](super_register(c))(x)}
```

The BoundedRegisterStructure theory now contains an additional casting operation from BoundedRegisterClass to RegisterClass.

```
  BoundedRegisterClass : TYPE =
    [# clg : (BoundedRegisterAssert?),
       new : (BoundedRegisterCreate?(clg)) #]

  cast : [BoundedRegisterClass  -> RegisterClass] =
    LAMBDA(cl : BoundedRegisterClass) :
      [# clg := super_Register(clg(cl)),
         new := new(cl) #]
```

(Well-definedness of `cast` involves proving two easy results.) When an implementation for a bounded register is described, definitions for the methods in BoundedRegister (*i.e.* count and write) and for those in the superclass (*i.e.* read, write, erase) have to be given. An obvious implementation of the bounded register specification uses the Cartesian product `[nat, FunctionSpace]` as state space, where `FunctionSpace` is the state space of the first Register implementation in the previous section. The first component `nat` describes the value of count. Appropriate operations on this state are easily defined, by re-using the Register implementation on `FunctionSpace`. The contents of the theory with the loose model is not influenced by inheritance and also the way the theory is generated is not altered.

4 Modeling other object-oriented aspects

This section briefly discusses how—and to what extend—various typically object-oriented features are realised in our formalisation. Not all of the aspects that we touch upon have fully crystalised into stable form, and the further development and use of our tool may lead to certain changes.

Component classes. When specifying a new class one often wishes to use another class as a component. By component we mean an attribute which is an instance of another class. This is also known as an aggregation realising a *has-a* relationship between two classes.

```
Begin Counter [ n: posnat, val_init : nat] : CLASSSPEC
  Method      val : Self -> nat;
              next : Self -> Self;
              clear : Self -> Self
  Assertion   val_next : PVS val(next(x)) =
                              IF val(x) = n-1 THEN 0 ELSE val(x)+1 ENDIF
                         ENDPVS
              val_init : PVS val_init <= n ENDPVS
              val_clear : PVS val(clear(x)) = 0 ENDPVS
  Constructor new : Self
  Creation    val_new : PVS val(new) = val_init ENDPVS
End Counter
```

Fig. 3. A counter (modulo n) class specification in CCSL

To demonstrate the use of components we adopt an example from [12]. Suppose that we have a class Counter, which counts modulo a parameter n, as in Figure 3. This class Counter is used (twice) as a component in the class specification of a DoubleCounter in Figure 4. A DoubleCounter has two counters as components, both counting modulo n. It has operations next, val and clear. The first counter is incremented every time a next operation is executed. The second counter is only incremented when the first counter reaches n.

As we have seen, our tool automatically generates loose and final models (without any internal structure) for every specification, and presents an option for the user. Both these models can be used for components, but a final model enables subclassing for components.

```
Begin DoubleCounter[ n: posnat ] : CLASSSPEC
  Method    val : Self -> nat;
            first : Self -> Counter[n,0];
            second : Self -> Counter[n,0];
            next : Self -> Self;
            clear : Self -> Self
 Assertion val_def : PVS val(x) =
                         n * val(second(x)) +
                             val(first(x)) ENDPVS
          first_next : PVS bisim?(first(next(x)), next(first(x))) ENDPVS
          second_next : PVS bisim?(second(next(x)),
                               IF val(first(x)) = n-1
                               THEN next(second(x))
                               ELSE second(x) ENDIF) ENDPVS
          first_clear : PVS bisim?(first(clear(x)), clear(first(x))
                            ENDPVS
          second_clear : PVS bisim?(second(clear(x)), clear(second(x)))
                             ENDPVS
  Constructor new : Self
  Creation first_new: PVS bisim?(first(new), new)  ENDPVS
          second_new: PVS bisim?(second(new), new)  ENDPVS
End DoubleCounter
```

Fig. 4. A double counter class specification in CCSL

As an example, the interface for DoubleCounter, using a loose model for the components, will be generated as follows.

```
  DoubleCounterIFace : TYPE = [# val : nat,
                                 first : LooseCounterType[n,0],
                                 second : LooseCounterType[n,0],
                                 next : Self,
                                 clear : Self #]
```

When generating the other theories for DoubleCounter, components are handled just as normal attributes (with bisimilarity as their equality relation).

Refinement. Earlier we mentioned how to implement a class specification and how to develop its theory. A third important activity is proving refinements between class specifications. We say that a "concrete" class refines an "abstract" class when a model (*i.e.* an implementation) of the abstract class can be described in terms of the concrete class. We construct this model as abstract(c): [Self$\rightarrow$ AbstractIFace[Self, $\cdots$]], where c: [Self $\rightarrow$ ConcreteIFace[Self,$\cdots$]] is an arbitrary model of the concrete class[4]. Following [13] we do not need the entire state space Self to obtain an "abstract" model, but we can restrict ourselves to the subtype (P) of Self arising from an invariant P on Self (*w.r.t.* the abstract class). Then abstract(c) restricts to an operation of type $[(P) \rightarrow \text{AbstractIFace}[(P), \cdots]]$. Of course, it has to be proven that the model satisfies the assertions and creation-conditions of the abstract class, as expressed by the following lemma.

```
  Abstract_refine : LEMMA
    AbstractAssert?(abstract(c)) AND AbstractCreate?(abstract(c))(new)
```

[4] Such a model abstract(c) should actually incorporate models of all the superclasses of the abstract class. Therefore, in practice, the model abstract(c) is best constructed by first constructing all these "super" models.

As an example, we can prove that DoubleCounter with parameter n refines a counter modulo n^2. The model for this refinement uses the invariant that the values of both component counters are bounded by n.

Overloaded methods. Some object-oriented languages allow overloading of methods: multiple methods with the same name may occur in the same class as long as their types are different. This is also possible in CCSL. PVS does allow overloading of functions, but field names in a labeled product—used as types of interfaces—are not permitted, hence we use ordinary products in interfaces with overloading.

Multiple inheritance. In our formalization we allow multiple inheritance (even though some object-oriented languages do not). This requires coping with name clashes, for instance: (1) if different superclasses define a method with the same name, and (2) if one class is inherited twice via different paths. To solve the first problem, the user can rename the conflicting methods in the **INHERIT FROM** section in the CCSL specification, like in EIFFEL [16]. As an example, a class can inherit both from Counter and from DoubleCounter in the following manner.

```
INHERIT FROM Counter[n,0] RENAMING val AS val_c AND
                                   next AS next_c AND
                                   clear AS clear_c,
             DoubleCounter[n] RENAMING val AS val_d AND
                                   next AS next_d AND
                                   clear AS clear_d
```

This will lead to method definitions like

```
val_c(c) : [Self -> nat] =
  LAMBDA(x : Self) : val(super_Counter(c(x)))
val_d(c) : [Self -> nat] =
  LAMBDA(x : Self) : val(super_DoubleCounter(c(x)))
```

Renaming is also necessary for different instances of the same class. The second problem of multiple paths to the same method is solved essentially by using sets of ancestor methods.

Creation with parameters. So far we have simply used 'new' in CCSL specifications as a constructor which returns a new instance of a class. In object-oriented languages one can usually parametrise such constructors with the initial values of the attributes. Typically, in a point class (specification) with attributes **fst** and **snd** for first and second coordinate, one may wish to have **new** as a (binary) constructor satisfying the following creation-conditions.

```
fst(new(a, b)) = a AND snd(new(a, b)) = b
```

This option also exists in CCSL: one can put constructors as functions with type $[A_1, \cdots, A_n] \to$ Self in the constructor section. They are handled in PVS via a labeled product containing all these constructors, instead of a single constructor **new**, as in the examples in Sections 2 and 3. Since we have not yet reached agreement on whether or not constructors should be inherited in object-oriented specifications, we included both options.

Subtyping. The usual object-oriented view is that inheritance (subclassing) implies subtyping (see [1, Section 3.2]), namely of the form: in every place where an object from a superclass is expected, an object from a subclass may be used as well. This is because all methods from the superclass also exist in the subclass—possibly in overridden form, but still with the same type. Precisely this aspect of subclassing exists in our formalisation because all methods from superclasses are explicitly (re-)defined in subclasses, see the definitions of `read(c)` *etc.* for bounded registers in Section 3. This "structural" subtyping (see again [1, Section 3.2]) arises because the Register interface is part of the BoundedRegister interface. Also we use explicit casting operations from subclasses to superclasses, as described for bounded registers in Section 3. Such casting operations are generated for components as final models.

Binary methods. Binary methods are a topic of intense debate in the object-oriented community, see [3]. They are allowed in many object-oriented languages, but can lead to various problems (notably type insecurities). A standard example of a binary method is the union (or intersection) operation in a class (specification) of sets (over some parameter type A).

```
...
elem? : [Self, A -> bool];
add, delete : [Self, A -> Self];
union, intersection : [Self, Self -> Self];
...
```

Typically, a binary (or n-ary, for $n > 1$) method takes multiple inputs of type Self. Methods of type $[\mathrm{Self}, \mathrm{A}_1, \cdots, \mathrm{A}_n] \to F(\mathrm{Self})$ are allowed in CCSL under the following two restrictions: (1) if Self occurs in A_i then $\mathrm{A}_i = \mathrm{Self}$, (2) Self occurs only positively in F.

Late binding. Consider a Point class specification with attributes `fst` and `snd` (as above) and with a `move` method satisfying:

```
fst(move(x,da,db)) = fst(x) + da AND snd(move(x,da,db)) = snd(x) + db
```

Suppose now that we often need the `move` operation with parameters `da` = `db` = 1, and decide to define it explicitly as `move1(x)` = `move(x,1,1)`. Late binding means that if we later override `move` in a subclass of Point, then the `move1` method will change accordingly: its definition will then use the overridden `move`. At this moment we have an *ad hoc* solution to model late binding, and we are still testing its appropriateness in various examples.

5 The front-end LOOP tool

Thus far we have seen how (CCSL) class specifications can be translated into higher order logic. This translation is done automatically by our tool, which is constructed as a front-end to a proof assistant. In general, front-end tools provide a higher level interface tailored to a specific application domain [2,20,23, 15,5]. They vary in the degree of sophistication and user support. While simple systems feature theory blueprints where the user fills out special slots in combination with specialised high level tactics [2,5], more advanced approaches define a special language and provide command line compilers [20] or even interactive user interfaces [15].

Our development aims at an environment in which the user can specify classes in several languages and frameworks and can then reason about their properties and relationships in a suitable proof assistant of choice. Ultimately, we desire a tool, called LOOP (for: Logic of Object-Oriented Programming), which provides an interactive (emacs) shell for the proof assistant. Thus far, as a first step, we focus on the compiler, which generates for a given class specification the corresponding theory and proof representations for the target proof assistant. It should be easy to extend the tool to other object-oriented languages and proof assistants. Also, it should come with a suitable graphical user interface. These aims influenced the choice of the implementation language and the architecture of the compiler.

We use the typed functional language Objective Caml (OCAML) [22], the current release of the French ML dialect CAML. Objective Caml provides, above the strict typing and readable syntax of an ML dialect, a typed module system, command line compilers with the capability of generating native machine code, lexer and parser generators, and an extensive library including an X-Window interface.

The architecture of the compiler (see Figure 5) exploits standard compiler construction techniques. It is organised in a number of passes which work on fixed interface data structures. This enables us to easily plug-in modules for other input languages (than CCSL) and other target proof assistants (than PVS).

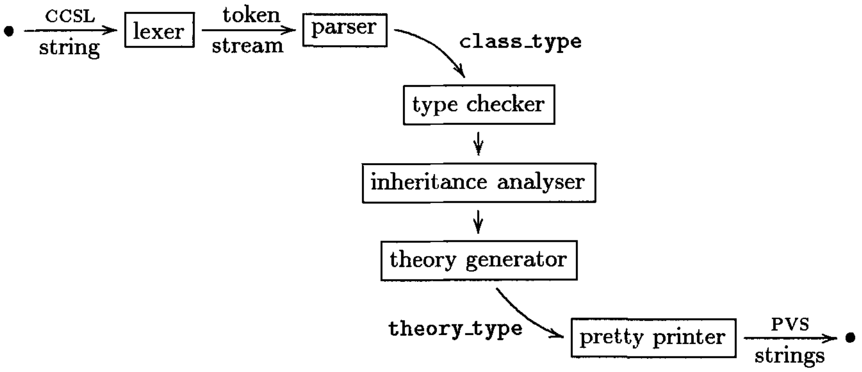


Fig. 5. Tool architecture

The compiler basically consists of the input modules lexer and parser, the internal modules (the vertical part in Figure 5), and the pretty printer. The lexer and parser are generated by the OCAML tools OCAMLLEX and OCAMLYACC which resemble the well-known LEX and YACC from C programming environments. Parsing a (CCSL) string yields an internal symbolic class represented as a value of the complicated, inductively defined OCAML type `class_type`. The parser can be replaced by any other function which generates values of `class_type`. All internal passes have input and output values in this type. The real work is carried out at a symbolic level. Extra steps can easily be inserted. After type checking and performing several semantic checks (for instance to determine the full inheritance tree of a class) the final internal pass produces symbolic theories and proofs as values of the OCAML type **theory_type**. This latter pass is the workhorse of the whole system. Finally, a target specific pretty printer converts the symbolic representation for PVS (or another proof assistant).

Currently, the compiler accepts CCSL class specifications in a file *name*.**beh** and generates the corresponding theories and proofs as described in the previous sections. For instance, compilation of a file **register.beh** containing the simple specification from Figure 1 will generate the files **register.pvs** and **register.prf**. The file **register.pvs** can then be loaded, parsed, and type checked in PVS. Before filling out the theory frames as described above the user can prove automatically all the standard lemmas with the **proof-file** command.

Conclusions and future work

We have elaborated a way to model object-oriented class specifications in higher order logic in such detail that it is amenable to tool support. Future work, as already mentioned at various points in this paper, involves: elaboration of the formal definition of CCSL (including *e.g.* visibility modifiers and late bindings), completion of the implementation of the LOOP tool, definition of appropriate tactics, stepwise refinement, development of various extensions to the tool and of course: use of the tool in reasoning about various object-oriented systems.

Acknowledgements

We thank David Griffioen for helpful discussions.

References

1. M. Abadi and L. Cardelli. *A Theory of Objects.* Monographs in Comp. Sci. Springer, 1996.
2. M. Archer and C. Heitmeyer. TAME: A specialized specification and verification system for timed automata. In A. Bestavros, editor, *Work In Progress (WIP) Proceedings of the 17th IEEE Real-Time Systems Symposium (RTSS'96)*, pages 3–6, Washington, DC, December 1996. The WIP Proceedings is available at http://www.cs.bu.edu/pub/ieee-rts/rtss96/wip/proceedings.
3. K. Bruce, L. Cardelli, G. Castagna, The Hopkins Objects Group, G. Leavens, and B. Pierce. On binary methods. *Theory & Practice of Object Systems*, 1(3), 1996.
4. C. Cîrstea. Coalgebra Semantics for Hidden Algebra: parametrised objects and inheritance. To appear in: *Workshop on Algebraic Development Techniques* (Springer LNCS), 1998.
5. A. Dold, F.W. von Henke, H. Pfeifer, and H. Rueß. Formal verification of transformations for peephole optimization. In *FME '97: Formal Methods: Their Industrial Application and Strengthened Foundations*, Lecture Notes in Computer Science.
6. 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.

7. J. Gosling, B. Jay, and G. Steele. *The Java Language Specification.* Addison-Wesley, 1996.
8. R. Hennicker, M. Wirsing, and M. Bidoit. Proof systems for structured specifications with observability operators. *Theor. Comp. Sci.*, 173(2):393–443, 1997.
9. C. Hermida and B. Jacobs. Structural induction and coinduction in a fibrational setting. *Information & Computation* (to appear).
10. B. Jacobs. Inheritance and cofree constructions. In P. Cointe, editor, *European Conference on Object-Oriented Programming*, number 1098 in Lect. Notes Comp. Sci., pages 210–231. Springer, Berlin, 1996.
11. 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.
12. B. Jacobs. Behaviour-refinement of coalgebraic specifications with coinductive correctness proofs. In M. Bidoit and M. Dauchet, editors, *TAPSOFT'97: Theory and Practice of Software Development*, number 1214 in Lect. Notes Comp. Sci., pages 787–802. Springer, Berlin, 1997.
13. 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, Berlin, 1997.
14. B. Jacobs and J. Rutten. A tutorial on (co)algebras and (co)induction. *EATCS Bulletin*, 62:222–259, 1997.
15. J. Knappman. A PVS based tool for developing programs in the refinement calculus. Master's thesis, Inst. of Comp. Sci. & Appl. Math., Christian-Albrechts-Univ. of Kiel, 1996. `http://www.informatik.uni-kiel.de/inf/deRoever/DiplJKm.html`
16. B. Meyer. *Object-Oriented Software Construction.* Prentice Hall, 2^{nd} rev. edition, 1997.
17. S. Owre, S. Rajan, J.M. Rushby, N. Shankar, and M. Srivas. PVS: Combining specification, proof checking, and model checking. In R. Alur and T.A. Henzinger, editors, *Computer Aided Verification*, number 1102 in Lect. Notes Comp. Sci., pages 411–414. Springer, Berlin, 1996.
18. 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.
19. L.C. Paulson. *Isabelle: A Generic Theorem Prover.* Number 828 in Lect. Notes Comp. Sci. Springer, Berlin, 1994.
20. C.H. Pratten. An Introduction to Proving AMN Specifications with PVS and the AMN-Proof Tool. `http://www.dsse.ecs.soton.ac.uk/~chp/amn_proof/papers.html`.
21. H. Reichel. An approach to object semantics based on terminal co-algebras. *Math. Struct. in Comp. Sci.*, 5:129–152, 1995.
22. D. Rémy and J. Vouillon. Objective ML: A simple object-oriented extension of ML. In *Princ. of Progr. Lang.*, pages 40–53. ACM Press, 1997.
23. J.U. Skakkebæk. *A Verification Assistant for a Real Time Logic.* PhD thesis, Dep. of Computer Science, Techn. Univ. Denmark, 1994.