The Coalgebraic Class Specification Language CCSL

JAN ROTHE, HENDRIK TEWS
Dep. Theor. Comp. Sci., TU Dresden,
D-01062 Dresden, Germany
{*janr,tews*}*@tcs.inf.tu-dresden.de*

BART JACOBS
Dep. Comp. Sci., Univ. Nijmegen,
P.O. Box 9010, 6500 GL Nijmegen, The Netherlands
bart@cs.kun.nl

April 27, 2000

Abstract. This note presents the *Coalgebraic Class Specification Language* CCSL that is developed within the LOOP project[1]. CCSL allows the (coalgebraic) specification of behavioral types or classes from object-oriented languages. A front-end to the theorem provers PVS [ORR+96] and ISABELLE [Pau94] compiles CCSL specifications into the logic of the theorem provers and and allows to mechanically reason about the specifications.

1. Introduction

The use of *coalgebras* as semantics for object orientation (explicitly proposed in [Rei95]) is for us the most promising approach towards specification and verification of classes. Coalgebras are the categorical duals to algebras (see [JR97]). They are functions Self $\longrightarrow T(\text{Self})$, where T is an interface type (technically, an endofunctor on the category of sets and total functions). The state space Self is seen as a *black box* and the coalgebra comprises all operations that are available to manipulate objects (or elements of Self). As usual in object-oriented programming we use the term *method* for all the operations. The methods can change the internal state of an object or carry out observations on the state space. Thus, properties of elements of the state space can only be *observed* by the operations of the coalgebra. This observational approach to object semantics is contrary to the constructional view on abstract data types. Their semantics, as represented by (initial) algebras, describes every element as *built* via constructor operations. This is not the way a client observes objects.

Coalgebras can naturally model systems with infinite behavior and partial operations. To reason about coalgebras we use the notions of invariance (closure under the application of methods) and bisimulation (indistinguishability via methods). In an appropriate categorical setting both notions can be directly derived from the interface functor T.

The present note sketches the latest version of the coalgebraic class specification language CCSL (see also [HHJT98]). This language supports nested coalgebraic and algebraic specifications. To restrict the behavior of coalgebras we use a higher-order logic enriched with powerful (method-wise) modal operators; see [Jac99, Rot00] for details. The CCSL compiler developed jointly in Dresden and Nijmegen translates CCSL specifications into the logic of the theorem provers PVS or ISABELLE.

We start in Section 2 with a discussion of an example class specification. Section 3 describes the CCSL compiler, especially the notions which it supports. Finally, in Section 4, we summarize the applications of CCSL and conclude.

2. Example

A coalgebraic specification in CCSL consists of a number of sections, describing the class interface, the assertions and the creation conditions. The class interface consists of a definition of the types of methods and constructors of the class and inheritance information. The assertions describe the behavior of the methods. Finally, the creation conditions establish restrictions on the outcome of a class constructor.

[1] See URL http://www.cs.kun.nl/~bart/LOOP/.

As an example, we consider a simple FIFO queue. It contains two methods. The method `put` adds a new element to the end of the queue. The method `top` delivers the first element of the queue together with the successor state, in which this first element is removed.

Figure 1 depicts the specification of the queue in CCSL. The first line declares `Queue` as the name of the class specification. The specification is parametric in the type `A` of the elements of the queue. This type parameter must be instantiated in actual use.

In the declaration of the two methods, $\times$ denotes the Cartesian product and $+$ the coproduct[2] of two types. We use $\mathbf{1}$ to denote the unit type that contains exactly one element $\mathbf{1} = \{*\}$. In the example we use `bot` instead of $\kappa_2(*)$ and $\mathtt{up}(a, x)$ for $\kappa_1(a, x)$.

The method `put` is declared on line 3; its type is clear from the preceding explanations: `put` takes a state and a data element and produces a new state with the element added at the end of the queue[3]. The method `top`, which delivers the first element of the queue, is a partial operation because it fails on empty queues. We can model this operation as a coalgebra into $(\mathtt{A} \times \mathrm{Self}) + \mathbf{1}$. If the result of the method `top` is `bot` then we say that the method fails. Otherwise it (successfully) delivers an element of `A` and a successor state of the queue.

In an algebraic approach the partiality of `top` would be handled via subsorts—which introduce all kinds of complications, see for instance [GD94]. Coalgebraically, the structured output type of `top` handles this partiality quite naturally. This idea scales to more complicated forms like non-determinism or computations that involve exceptions.

Note that, using currying, the method `put` can be transformed into an coalgebra. Then both methods can be combined into one coalgebra of type $\mathrm{Self} \longrightarrow (\mathrm{Self}^A) \times ((A \times \mathrm{Self}) + \mathbf{1})$ to match the view on coalgebraic specification that we presented in the introduction. But in applications it is better to distinguish different operations and to allow a finite number of coalgebras per class specification.

With coalgebras alone it is not possible to construct new objects (elements of Self) out of nothing. Therefore in CCSL we combine a coalgebraic method signature with a (degenerated) algebraic constructor signature. In our example we have one constructor `new_queue`, which is a constant in Self. It serves as initial state.

The assertions restrict the behavior of the two methods. Technically an assertion is a logical formula with one free variable of type Self. This variable is declared with the keyword `SelfVar` on line 7. The basic propositions of the logic are observational equalities $\underline{\leftrightarrow}$. Observational equality is the relation lifting (see [HJ98]) of bisimilarity to arbitrary types. In the example we only use observational equality for the type $(\mathtt{A} \times \mathrm{Self}) + \mathbf{1}$. Two elements u and v of this type are observational equal if and only if they either both equal `bot` or if $u = \mathtt{up}(a_1, x)$ and $v = \mathtt{up}(a_2, y)$ such that a_1 equals a_2 and x and y are bisimilar (with respect to the interface of `Queue`).

Compared to usual equality, observational equality has several advantages. First, it mirrors the intuition that the state space of a class is seen as black box. Thus, two elements of Self can only be compared up to the observations they yield. Second, as a technical condition the use of observational equality ensures the existence of final models (for consistent specifications).

The first assertion `queue_empty` describes the behavior of an empty queue. A queue is empty, if the `top` method fails, that is, if `x.top = bot`. The implication in lines 9 and 10 describes that, if x denotes an empty queue, then after putting an $a \in \mathtt{A}$ into this queue, the first element of the queue is a and, removing this element, we get a queue that is observationally equal to the original empty one.

The second assertion describes the behavior of nonempty queues. To ensure that x is nonempty, we could have taken the expression `NOT` x`.top = bot`. Because we need the result of this application later in the formula, we bind y and a in the condition of the implication. The assertion describes that the first element of the nonempty queue does not change when adding a new element to its end. Further, for nonempty queues, the two operations of adding an element to the queue and removing the head of the queue commute (up to observational equality).

The final part of a class specification describes properties of objects that are created using the con-

[2] In the category of sets and total functions the coproduct of two sets A and B is given by their disjoint union together with the injections $\kappa_1 : A \longrightarrow A + B$ and $\kappa_2 : B \longrightarrow A + B$.

[3] Note that this style of specification is functional: objects are not handled by reference, but by value; the `put` operation produces a new object.

```
BEGIN Queue[ A : TYPE ] : CLASSSPEC
  METHOD
    put :  Self × A -> Self;
    top :  Self -> (A × Self)+1;
  CONSTRUCTOR
    new_queue:  Self;
  ASSERTION   SelfVar x:  Self
    queue_empty :
      x.top = bot IMPLIES
        FORALL (a:A). x.put(a).top ↔ up(a,x)
    queue_filled:
      FORALL(a:A,y:Self).  x.top ↔ up(a,y) IMPLIES
        FORALL (a':A). x.put(a').top ↔ up(a,y.put(a'))
  CREATION
    queue_cre:
      new_queue.top = bot
END Queue
```

Figure 1: A queue class specification in CCSL.

structor of the class. In our case, there is one creation condition. It asserts, that a queue created by `new_queue` is initially empty.

3. Verification Support

In the LOOP project we are most interested in practical applications. Therefore we did not develop our own theorem prover for coalgebraic class specifications. Instead, we decided to implement a front-end tool (the CCSL compiler) for existing theorem provers (currently ISABELLE and PVS). This section contains an overview of the results of our efforts.

The CCSL compiler is written in OCAML [L+99]. An execution for some class specification performs four major steps: first, the specification is parsed. In this step, an internal representation of the specification is generated. Next, the tool checks type correctness for the logical formulae that appear in the assertions and creation conditions. The third pass produces an internal representation of the theories that have to be generated. Finally, the pretty-printing pass writes these theories into source files for the desired theorem prover.

Supported Notions

After running the CCSL compiler on a source file, you receive the generated theories as an output. Among others, you find theories that define the interface (signature) of the class specification, the notions of bisimilarity and invariance, and the notion of morphisms between coalgebras for the class signature. These definitions together with associated lemmas serve as tools for the following more complex concepts.

Models

A model of a specification is developed in two steps: First, one has to construct a state space and a coalgebra for the signature of the specification. Second, one has to prove that the assertions hold on this coalgebra. The CCSL compiler supports this by generating suitable predicates for proving whether a coalgebra is a model of the specification.

To check whether a class specification matches ones intuition, one can examine the final model (in the category of all models) of the specification. The final model of a class specification comprises all possible behaviors of all models of the specification. Nicely enough it allows to directly access the elements of the state space since bisimilarity boils down to equality on final models. Thus, by showing that one's intended model is final, it is easy to determine whether the specification is too loose in the sense that it allows unintended behavior.

Inheritance

The CCSL compiler supports (multiple) inheritance of specifications. Besides the sections for methods, constructors, assertions, and creation conditions, a class specification can contain an inheritance section like

```
INHERIT FROM Queue[A]
        RENAMING put AS enqueue
```

This has the effect that, provided A is a valid type, the current specification is extended with the methods and the assertions of the queue specification. During this extension the original method put is renamed into enqueue and the CCSL compiler ensures that the inherited assertions queue_empty and queue_filled restrict the behavior of enqueue instead of put.

Components

While inheritance models an *is-a* relation, it is also possible to model the *contains-a* relation between different systems. Imagine a specification with a method declaration like

```
queues :  [Self, nat] -> Queue[A]
```

Then models of this specification are required to *contain* an infinite array of queues over type A. For components we assume a standard semantics. Currently the user can choose between *loose semantics*, where an arbitrary model of the queue specification is used as semantics for the component, or *final semantics*, where the final model is used.

Modal Operators

Modal operators are well suited for the verification of properties of potentially nonterminating systems. The third author recognized in [Jac99] the connection between greatest invariants and the future time necessity operator. This was worked out for CCSL in [Rot00].

The CCSL compiler generates theories that define (method wise) future time necessity and eventuality operators. Using these operators in our example, it is for instance possible to prove that every queue, that is reached from an empty queue, can be emptied again.

Refinement

Refinement is a relation between two specifications. A concrete specification $\mathcal{C}$ *refines* an abstract specification $\mathcal{A}$ if all models of $\mathcal{C}$ can be turned (in a generic way) into models of $\mathcal{A}$. Refinements can be concatenated in a number of steps, starting from a high level abstract specification and leading to an implementation. The properties of refinement ensure that the models of the most concrete specification still fulfill the assertions of the original abstract specification.

For coalgebraic class specifications we have two notions of refinement. *Model theoretic refinement* is a generalizations of [Jac97] and is based on the notion of models. In contrast, *behavioral refinement* is based on bisimilarity of corresponding initial states. Behavioral refinement is more general than model theoretic refinement. Behavioral refinement can also be used if only a part of the abstract class interface (for instance the public part) is refined. Under mild assumptions on the logic used in the assertions, it can be shown that, in a situation where both notions of refinement can be applied, behavioral refinement implies model theoretic refinement.

Binary Methods

The theory of coalgebras is usually developed for (covariant) endofunctors on an arbitrary category. Consider the example of a binary method

```
equal :  [Self, Self] -> bool
```

To model such an operation coalgebraically one would need the functor $F(X) = 2^X$, where 2 is the semantics of type bool. But on the categories we are interested in, this F cannot be turned into a covariant endofunctor.

In [Tew00b] the second author shows how the notions of coalgebra and bisimulation can be generalized to *extended polynomial functors* $\mathbf{Set}^{\mathrm{op}} \times \mathbf{Set} \longrightarrow \mathbf{Set}$. Coalgebras for extended polynomial functors can model binary methods for practical applications. The CCSL compiler accepts signatures that correspond to extended polynomial functors.

4. Conclusion

In this note we presented the coalgebraic specification language CCSL and sketched its possible applications. CCSL relies on coalgebras and naturally uses the notions of invariance and behavioral equivalence. Other people from the LOOP project apply the same coalgebraic ideas to reason about JAVA programs [JvdBH+98]. In this work they also develop a front-end to PVS and ISABELLE to use a theorem prover to reason about JAVA programs.

The specification language CCSL has been successfully used in several non-trivial case studies. Meyer specifies in [Mey99] the MSMIE (multiprocessor shared information exchange) protocol in CCSL. He refines this initial specification in three steps and proves the correctness of a Java implementation of the protocol.

In [Tew00a], the second author uses CCSL to formalize parts of the memory management of the micro-kernel FIASCO. He then examines the C++ sources of FIASCO. The case study revealed some hidden assumptions about the internal interface of the memory management of FIASCO. The case also study proves that the specification language CCSL together with the theorem prover PVS can well be applied to *real* software.

Currently CCSL admits coalgebraic class specification (as described in Section 2) and ADT definitions. One ongoing project is to investigate the possibility to include algebraic specifications in CCSL.

5. References

[GD94] J.A. Goguen and R. Diaconescu. An Oxford survey of order sorted algebra. *Mathematical Structures in Computer Science*, 4:363–392, 1994.

[HHJT98] U. Hensel, M. Huisman, B. Jacobs, and H. Tews. Reasoning about classes in object-oriented languages: Logical models and tools. In *Programming Languages and Systems*, number 1381 in Lecture Notes in Computer Science, pages 105–121. Springer, 1998.

[HJ98] C. Hermida and B. Jacobs. Structural induction and coinduction in a fibrational setting. *Inf. & Comp.*, pages 107–152, 1998.

[Jac97] B. Jacobs. Invariants, bisimulations and the correctness of coalgebraic refinements. In M. Johnson, editor, *Algebraic Methodology and Software Technology*, volume 1349 of *Lecture Notes in Computer Science*, pages 276–291. Springer, 1997.

[Jac99] B. Jacobs. The temporal logic of coalgebras via Galois algebras. Technical Report CSI-R9906, Computing Science Institute, University of Nijmegen, 1999.

[JR97] B. Jacobs and J. Rutten. A tutorial on (co)algebras and (co)induction. *EATCS Bulletin*, 62:222–259, 1997.

[JvdBH+98] B. Jacobs, J. van den Berg, M. Huisman, M. van Berkum, U. Hensel, and H. Tews. Reasoning about classes in Java (preliminary report). In *Object–Oriented Programming, Systems, Languages and Applications*, pages 329–340. ACM Press, 1998.

[L+99] X. Leroy et al. *The Objective Caml system release 2.02*. Insitut National de Recherche en Informatique et Automatique, March 1999.

[Mey99] Dion Meyer. A case study in object oriented specification and verification: The MSMIE protocol. Master's thesis, Katholieke Universiteit Nijmegen, January 1999.

[ORR+96] 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 Lecture Notes in Computer Science, pages 411–414. Springer, Berlin, 1996.

[Pau94] L. C. Paulson. *Isabelle: A Generic Theorem Prover*. Number 828 in Lecture Notes in Computer Science. Springer, Berlin, 1994.

[Rei95] H. Reichel. An approach to object semantics based on terminal coalgebras. *Mathematical Structures in Computer Science*, 5:129–152, 1995.

[Rot00] J. Rothe. Modal logics for coalgebraic class specification. Master's thesis, Department of Theoretical Computer Science, TU Dresden, D-01062 Dresden, Germany, 2000.

[Tew00a] H. Tews. A case study in coalgebraic specification: Memory management in the FIASCO microkernel. Technical Report TPG2/1/2000, SFB 358, April 2000. available at URL http://wwwtcs.inf.tu-dresden.de/~tews/vfiasco/.

[Tew00b] H. Tews. Coalgebras for binary methods. In H. Reichel, editor, *Coalgebraic Methods in Computer Science*, volume 33 of *ENTCS*. Elsevier, Amsterdam, 2000.