Journal of Universal Computer Science, vol. 7, no. 2 (2001), 175-193
submitted: 1/9/00, accepted: 13/10/00, appeared: 28/2/01 © Springer Pub. Co.

The Coalgebraic Class Specification Language CCSL[1]

Jan Rothe
(Institut Theoretische Informatik
TU Dresden, Germany
janr@tcs.inf.tu-dresden.de)

Hendrik Tews
(Institut Theoretische Informatik
TU Dresden, Germany
tews@tcs.inf.tu-dresden.de)

Bart Jacobs
(Department Computer Science
Univ. Nijmegen, The Netherlands,
bart@cs.kun.nl)

Abstract: This paper presents the *Coalgebraic Class Specification Language* CCSL that is developed within the LOOP project on formal methods for object-oriented languages[2]. CCSL allows the (coalgebraic) specification of behavioral types and classes of object-oriented languages. It uses higher-order logic with universal modal operators to restrict the behavior of objects. A front-end to the theorem provers PVS [ORR+96] and ISABELLE [Pau94] compiles CCSL specifications into the logic of these theorem provers and allows to mechanically reason about the specifications.

Keywords: coalgebra, specification, binary methods, modal logic
Categories: D.2.4 D.2.1 F.3.1 F.4.1 (ACM'98)

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 fit nicely together with algebras (or abstract data types). The associated proof principles coinduction and induction can be used in a nested way [HJ97]. These observations lead to the development of the Coalgebraic Class Specification Language CCSL (see also [HHJT98]) as a joint project of the University of Dresden (Germany) and the University of Nijmegen (The Netherlands). CCSL allows to nest algebraic and coalgebraic specifications. This way the user can choose the appropriate specification technique for his problem: algebras for finitely generated data types and coalgebras for behavioral types.

In this paper we present the specification language CCSL. We focus on the nonstandard, coalgebraic part, extending the introduction from [HHJT98]. The main contribution of [HHJT98] was an application-oriented introduction into

[1] This work was partly funded by the DFG within the Graduiertenkolleg "Specification of discrete processes and systems of processes by operational models and logics".
[2] See URL http://www.cs.kun.nl/~bart/LOOP/.

coalgebraic specification and its semantics in HOL on the basis of a few examples. In the present paper we give a precise definition of polymorphic coalgebraic class specification and its semantics. Further, we show how to define method-wise modal operators for coalgebraic class specification.

The structure of this paper is as follows. In Section 2 we define our notion of *polymorphic coalgebraic class specification* and give its semantics in the universe of sets and total functions. The definitions will be illustrated with a specification of *first-in-first-out* (FIFO) queues. This takes up the main part of the paper. Though the chosen example is a simple one, it demonstrates many aspects of our specification language.

Together with CCSL we developed a compiler that translates specifications into logical theories of a theorem prover. This enables applications and case studies of considerable size in which the compiler and the proof tool relieve the user from many bureaucratic tasks. In Section 3 we present this CCSL compiler and discuss its structure. In Section 4 we present an extension of our logic. Infinitary modal operators based on the underlying coalgebraic structure can be used to formulate and prove safety and liveness properties for objects. Further features of CCSL are presented in Section 5. The conclusions in Section 6 give an overview of applications of the specification language CCSL. The remainder of this introduction explains coalgebras on an intuitive level.

In coalgebraic specification one considers *observations and transitions*. They are functions of the form

$$\mathsf{Self} \longrightarrow \boxed{\cdots\ \mathsf{Self}\ \cdots}$$

where the set Self is the state space of a system (or the set of all states of all objects of a given class). The right hand side is some (type) expression that may contain Self. In contrast, the operations considered in algebraic specification have the converse shape: Algebras are functions $\boxed{\cdots X \cdots} \longrightarrow X$ going *into* a set X. Because of this duality, observation and transition functions are called *coalgebras* in the remainder of this paper. Examples of coalgebras are:

$$\mathsf{Self} \longrightarrow \mathsf{Data} \times \mathsf{Self} \qquad\qquad \mathsf{Self} \longrightarrow \big(\mathsf{Input} \to (\mathbf{1} + (\mathsf{Self} \times \mathsf{Output}))\big)$$

where $\times$ is the Cartesian product, $+$ is disjoint union, $\to$ is the function space, and $\mathbf{1}$ is the singleton set $\{\bot\}$. So a coalgebra c acts on states $x \in \mathsf{Self}$ and produces an output $c(x)$ giving some information about x or producing a next state x' (or both, as in the left example above).

There are also operations that are both algebraic and coalgebraic, for instance the algebra $\mathsf{Self} \times \mathsf{Input} \longrightarrow \mathsf{Self}$ can be equivalently seen as a coalgebra $\mathsf{Self} \longrightarrow (\mathsf{Input} \to \mathsf{Self})$. Following this idea, in CCSL we relax the definition of coalgebra for practical reasons and consider also functions

$$\boxed{\mathsf{Self} \times \cdots} \longrightarrow \boxed{\cdots\ \mathsf{Self}\ \cdots}$$

as coalgebras under suitable restrictions on the types of the argument side.

Algebras, possibly satisfying certain logical formulas, have been successfully applied in computer science for describing various data structures. Coalgebras are best suited for state-based dynamical systems where the internals of the state space are hidden. Important issues for coalgebras are bisimulation, coinduction, invariants, and modal logic. It is a great advantage of coalgebras, that the structure of the coalgebra delivers these notions for free. Some of this will be explained in this paper. For an introduction to coalgebras and coinduction see [JR97].

Directly related to this paper is the work of Cîrstea: In [Cîr99] she uses a restricted version of coalgebraic class signature such that she can obtain a sound and complete deduction system for her flavor of coalgebraic logic. Hennicker and Kurz consider in [HK99] coinductively defined algebraic extensions of coalgebraic signatures. The work on hidden algebra [Roş00] has similar motivations to our work. In hidden algebra signatures contain operations of the form $\mathsf{Self} \times S_1 \times \cdots \times S_n \longrightarrow S_0$, where (typically) Self is a hidden sort and the S_i are arbitrary (hidden or visible) sort. Our notion of coalgebraic class signatubre is single-sorted, but we allow for structured output types (as shown above) and also for structured argument types (for instance $\mathsf{Self} \times (\mathbb{N} \to \mathsf{Self}) \longrightarrow \cdots$). The use of coalgebras makes the treatment of partial operations very easy. In hidden algebra one has to use subsorting to model partiality.

We would like to thank two anonymous referees and Kai Brünnler for their helpful comments.

2 Coalgebraic Class Specifications

In this section we formally define the notion of *coalgebraic class specification* as it is used in the specification language CCSL. The formalization that we use is tailored towards practical application and perhaps not ideally suited for theoretical or categorical investigations in the theory of coalgebras. As a running example we use a specification of a FIFO queue; the complete specification in CCSL syntax is in Figure 1 on page 11.

In CCSL specifications are allowed to contain type parameters. In later use a specification might be instantiated by providing a concrete type for each of the type parameters. Inside the specification the type parameters appear as free type variables in type expressions. We assume an infinite set $\mathbb{V} = \{\alpha_1, \alpha_2, \ldots\}$ of type variables. For the semantics of type expressions, the type variables give rise to an additional level of indexing. Instead of taking a plain set M as semantics we take indexed collections $(M_{U_1,\ldots,U_n})$ where the index sets $U_1, \ldots, U_n$ are the interpretations of the type variables.

When developing a specification one rarely starts from scratch. Usually one expects an environment containing some primitive types (like the natural numbers) and type constructors (like $\mathsf{List}[-]$). Further one expects standard functions like addition or concatenation of lists. One also wants to use the types and the operations that have been defined by earlier specifications. All this will be captured by a *ground signature*.

Let us start with types. We use a specific instance of the polymorphic λ–calculus $\lambda\!\!\to$ over a polymorphic signature; see [Jac99a] for the general theory behind. The *types* depend on the set of allowed type constructors (which will be given by the ground signature). Type constructors have an arity assigned (the number of type arguments they take). For instance the type constructor $\mathbb{N}$, for the natural numbers, has arity 0, it is also called a type constant; the type constructor List has arity 1, it takes an arbitrary type σ and returns the type $\mathsf{List}[\sigma]$ of lists over σ. So assume we have an indexed set (C_n) for $n \in \mathbb{N}$ of type constructors, where each C_n contains the type constructors of arity n. The set of types $\mathbb{T}$ over (C_n) is generated by the following grammar.

$$\mathbb{T} ::= \mathbb{V} \mid \mathsf{Self} \mid \mathbb{C}_n[\mathbb{T}_1, \ldots, \mathbb{T}_n] \mid \mathbb{T} \times \mathbb{T} \mid \mathbb{T} + \mathbb{T} \mid \mathbb{T} \to \mathbb{T}$$

Here the meta-symbol $\mathbb{V}$ stands for the type variables and $\mathbb{C}_n$ stands for the type constructors of arity n from C_n (so type constructors must always be applied to the right number of arguments). The last three constructions are for the Cartesian product, the disjoint union, and the exponent type (the function space), respectively.

A type τ is called a *constant type* if it does not contain Self, it is called a *(strictly) covariant type* if τ contains Self only in strictly covariant positions (i.e., for all subexpressions $\rho \to \sigma$ occurring in τ the type ρ must be a constant type). For instance $(\alpha \to \mathsf{Self}) \times \mathsf{Self}$ is a covariant type, but $(\alpha \to \mathsf{Self}) \to \alpha$ is not.

A *ground signature* Ω is a pair $\langle (C_n), (\Omega_\sigma) \rangle$ of two indexed sets, such that (C_n) for $n \in \mathbb{N}$ is an indexed set of type constructors and (Ω_σ) is a set of constant (or function) symbols for each constant type σ over the collection (C_n). For $f \in \Omega_\sigma$ we say that f has type σ, also denoted by $f : \sigma$. If σ contains type variables, then f is a polymorphic constant or function. Throughout this paper we assume that the ground signature contains the type constant bool, which will be interpreted by the booleans, the constants $\mathsf{true}, \mathsf{false} \in \Omega_{\mathsf{bool}}$, the projection functions $\pi_1 : \alpha_1 \times \alpha_2 \to \alpha_1$ and $\pi_2 : \alpha_1 \times \alpha_2 \to \alpha_2$ and the injections $\kappa_1 : \alpha_1 \to \alpha_1 + \alpha_2$ and $\kappa_2 : \alpha_2 \to \alpha_1 + \alpha_2$. In the running example of queues we further assume the type constructor Lift of arity 1 that adds an error element $\perp$ to its argument.[3] We have two constants connected with Lift: the error element $\perp : \mathsf{Lift}[\alpha]$ and the injection $\mathsf{up} : \alpha \to \mathsf{Lift}[\alpha]$.

A class declaration (or a class interface) in an object-oriented programming language consists of a finite number of typed methods. They all have an additional hidden argument of type Self (which has to be given explicitly in CCSL). This motivates the following definition.

Definition 1. Assume a ground signature $\Omega = \langle (C_n), (\Omega_\sigma) \rangle$.

1. A type over the collection (C_n) of the form $(\mathsf{Self} \times \sigma_1 \times \cdots \times \sigma_n) \to \sigma$ (or of form $\mathsf{Self} \to \sigma$ for n=0) is called a *method type*. It is called a *binary method type* if one of the σ_i is not a constant type (i.e., σ_i does contain Self) or if σ is not a strictly covariant type.

2. The types of the form $\sigma \to \mathsf{Self}$, for any constant type σ, together with the type Self[4], form the set of *constructor types*.

3. A *coalgebraic class signature* is a pair $\langle \Sigma_M, \Sigma_C \rangle$, where Σ_M is a finite set of method declarations $m_i : \sigma_i$, for method types σ_i, and Σ_C is a finite

[3] Alternatively one can consider Lift as an abbreviation $\mathsf{Lift}[\alpha] = \mathbf{1} + \alpha$ and assume the one-element (or unit) type $\mathbf{1}$ together with $\perp : \mathbf{1}$ in the ground signature.

[4] The type Self is equivalent to $\mathbf{1} \to \mathsf{Self}$. So we include Self in the constructor types only for convenience.

set of constructor declarations $c_i : \tau_i$ for constructor types τ_i. The set of type variables occurring in the σ_i and the τ_i are the *type parameters* of the signature. If one of the method declarations involves a binary method type then the signature is said to contain binary methods.

Example 1. Consider a FIFO queue. It supports two operations, one for enqueuing elements (put) and one for removing elements from the head (top). Removing the first element from a queue is a partial operation, which fails if the queue is empty. Therefore the signature Σ_{Queue} contains the method declarations $\mathsf{put} : \mathsf{Self} \times \alpha \to \mathsf{Self}$ and $\mathsf{top} : \mathsf{Self} \to \mathsf{Lift}[\alpha \times \mathsf{Self}]$. So for any element x of Self either $\mathsf{top}(x) = \bot$ (signaling an empty queue) or $\mathsf{top}(x) = \mathsf{up}(a, x')$, where a is the first element of the queue and x' is the successor state of x with a removed. Notice that top is typically coalgebraic. To achieve the same in an algebraic setting would require two separate operations in partial algebra, one for displaying the top element and one for removing it; these operations need to be defined on the same subset of Self. All this is far less concise and natural.

For the creation of new queues we add the constructor declaration $\mathsf{new} : \mathsf{Self}$ to Σ_{Queue}. We could also use a constructor $\mathsf{new\_from\_list} : \mathsf{List}[\alpha] \to \mathsf{Self}$ that takes the elements of a list to initialize the queue.

Example 2 Expressions in Java. Coalgebras give a convenient mathematical representation of classes in Java (or in similar object-oriented programming languages), see [JvdBH+98]. To each class a coalgebraic class signature can be associated, reflecting the types of the fields, methods and constructors of the class. A model of such a signature (see Definition 3 below), i.e. a coalgebra, can then be understood as an implementation of the class.

We give a sketch of how this works, by concentrating on an example method in Java of the form:

```
int m(bool b) { ... }
```

Such a Java method can have three possible output modes: it may hang (e.g. through an infinite loop), terminate normally and produce an integer result together with a successor state, or terminate abruptly by throwing an exception (e.g. caused by a division by zero). These three different output options are captured in the following method type for the method `m`:

$$\mathsf{Self} \times \mathsf{bool} \longrightarrow 1 + (\mathsf{int} \times \mathsf{Self}) + (\mathsf{Excp} \times \mathsf{Self})$$

where Excp is an appropriate type for exceptions. One sees how the different possible outputs are naturally captured by using a structured codomain type—typical for coalgebras.

As the examples showed already, in CCSL we model methods and constructors in a *functional way*. This means that we do not assume a global state that is available to every method. Everything to which a method should have access must be passed as an argument (including the element of Self on which the method is invoked). Further, a method cannot change the current object. Instead it has to return a successor state as shown in the example above.

In object-oriented programming a method which acts on two (or more) objects is called a binary method [BCC+95]. There are different ways how to model binary methods in CCSL. One can, for instance, pass a reference of the second object (i.e., an natural number) as an additional argument to the method. However, the most natural way is to pass a second argument of type Self, like in equal : Self × Self → bool. Operations like the preceding equal do not fit into the traditional categorical definition of coalgebras. In CCSL we allow these operations (and also more complicated method types like $\mathsf{Self} \times (\mathbb{N} \to \mathsf{Self}) \to \mathbb{N}$) but the theory behind is complicated [Tew00b]. For the remainder of this paper we restrict ourselves to coalgebraic class signatures without binary methods, but see also Section 5.3.

The constructors that can occur in a coalgebraic class signature are quite restricted. They are merely parameterized initial states. For instance the *copy constructors* of C++, which create a copy of the current object, should be modeled as method $\mathsf{Self} \to \mathsf{Self} \times \mathsf{Self}$ in a coalgebraic class signature.

2.1 Semantics of class signatures

The start of the semantic development is an interpretation of the type constructors. So assume a fixed ground signature Ω in the following and let C be a type constructor of arity n. An interpretation of C is an indexed collection of sets $(\llbracket \mathsf{C} \rrbracket_{U_1,\ldots,U_n})$ where the indices $U_1, \ldots, U_n$ are ordinary sets. So for C we have, for each n–tuple of sets $U_1, \ldots, U_n$, a set $\llbracket \mathsf{C} \rrbracket_{U_1,\ldots,U_n}$ that gives us the semantics of the type constructor C when applied to the argument sets U_i. The interpretation of the type constructors expands in a straightforward way to an interpretation of all types.

Definition 2 Interpretation of types. Let Ω be a ground signature and assume we have an interpretation for each type constructor of Ω.

1. Let τ be a type containing at most k type variables $\alpha_1, \ldots, \alpha_k$. The interpretation of τ is an indexed collection of sets $(\llbracket \tau \rrbracket^X_{U_1,\ldots,U_k})$, where the set X interprets the type Self and the sets U_i interpret the type variables. It is defined by induction on the structure of τ.
   - If $\tau = \alpha_i$ for a type variable α_i, then $\llbracket \tau \rrbracket^X_{U_1,\ldots,U_k} = U_i$
   - If $\tau = \mathsf{Self}$ then $\llbracket \tau \rrbracket^X_{U_1,\ldots,U_k} = X$
   - If $\tau = \mathsf{C}[\sigma_1, \ldots, \sigma_n]$ for a type constructor C of arity n, then $\llbracket \tau \rrbracket^X_{U_1,\ldots,U_k} = \llbracket \mathsf{C} \rrbracket_{V_1,\ldots,V_n}$, where $V_i = \llbracket \sigma_i \rrbracket^X_{U_1,\ldots,U_k}$
   - If $\tau = \sigma_1 \odot \sigma_2$ then $\llbracket \tau \rrbracket^X_{U_1,\ldots,U_k} = \llbracket \sigma_1 \rrbracket^X_{U_1,\ldots,U_k} \odot \llbracket \sigma_2 \rrbracket^X_{U_1,\ldots,U_k}$ for $\odot \in \{\times, +, \to\}$. Here $\times$ denotes the Cartesian product, $+$ the disjoint union, and $M \to N$ denotes the set of all (total) functions between M and N.

   If τ is a constant type we omit the superscript X and write $(\llbracket \tau \rrbracket_{U_1,\ldots,U_k})$ for its interpretation.

2. A *model of the ground signature* consists of an interpretation of all type constructors from Ω and for each $f \in \Omega_\sigma$ for some type σ, an indexed collection of elements $(\llbracket f \rrbracket_{U_1,\ldots,U_k} \in \llbracket \sigma \rrbracket_{U_1,\ldots,U_k})$, where the U_i interpret the type variables α_i in σ. In the rest of the paper we identify the type constructors from

the ground signature with their interpretation, so we write $\mathsf{Lift}[X]$ for the set X enriched with a bottom element.

Using both parts of the previous definition, we can now define the semantics of a class signature.

Definition 3. Assume a model of the ground signature Ω and let $\Sigma = \langle \Sigma_M, \Sigma_C \rangle$ be a coalgebraic class signature with k type parameters. A *model for Σ* is an indexed collection $(\langle X, M, C \rangle_{U_1,\ldots,U_k})$ such that for each interpretation $U_1, \ldots, U_k$ of the type parameters we have

- a state space X,
- a set M of coalgebras that, for each symbol $m_i : \mathsf{Self} \times \sigma_1 \times \cdots \times \sigma_n \to \sigma$ from Σ_M, contains a function $[\![m_i]\!] : [\![\mathsf{Self} \times \sigma_1 \times \cdots \times \sigma_n]\!]^X_{U_1,\ldots,U_k} \to [\![\sigma]\!]^X_{U_1,\ldots,U_k}$.
- a set C of algebras containing one function $[\![c_i]\!] : [\![\sigma]\!]_{U_1,\ldots,U_k} \to X$ for each constructor symbol $c_i : \sigma \to \mathsf{Self}$ from Σ_C.

Remark. Consider a signature Σ without binary methods and without type parameters. The signature and its models can be more abstractly described: Σ corresponds to a pair (T_Σ, F_Σ) of endofunctors on the category **Set**. A model for Σ corresponds to a pair of functions $F_\Sigma(X) \to X \to T_\Sigma(X)$, that is to a T_Σ–coalgebra and a F_Σ–algebra on the same carrier.

To see how this works, recall first that the two types $\sigma_1 \times \sigma_2 \to \sigma_3$ and $\sigma_1 \to \sigma_2 \to \sigma_3$ are equivalent in the sense, that their interpretations are always isomorphic sets of functions. Further two method types $\mathsf{Self} \to \sigma_1$ and $\mathsf{Self} \to \sigma_2$ can be combined into one type $\mathsf{Self} \to \sigma_1 \times \sigma_2$. And for two constructor types $\sigma_1 \to \mathsf{Self}$ and $\sigma_2 \to \mathsf{Self}$ one can equivalently use $(\sigma_1 + \sigma_2) \to \mathsf{Self}$. So any signature Σ can be transformed into an equivalent signature Σ' with just one method type $\mathsf{Self} \to \sigma_\Sigma$ and one constructor type $\rho_\Sigma \to \mathsf{Self}$. So we set $T_\Sigma(X) = [\![\sigma_\Sigma]\!]^X$ and $F_\Sigma(X) = [\![\rho_\Sigma]\!]^X$. This extends to an endofunctor on **Set**.

Example 3 Model for Queue. The signature Σ_{Queue} from Example 1 has one type parameter α. A model for this signature consists of a set X_U for every set U and two coalgebras $\mathsf{put}_U : X_U \times U \to X_U$ and $\mathsf{top}_U : X_U \to \mathsf{Lift}[U \times X_U]$, and a constant $\mathsf{new}_U : X_U$. Let $\mathbb{N}^+$ be the natural numbers including infinity ∞ and take $X_U = \mathbb{N}^+ \times (\mathbb{N} \to U)$, so a state in X_U is a pair $\langle n, f \rangle$, consisting of the number n of elements in the queue and a function $f : \mathbb{N} \to U$ that gives the elements in the queue for arguments lesser than n. We set

$$\begin{aligned}
\mathsf{top}(\langle n, f \rangle) &= \begin{cases} \text{if } n = 0 : & \bot \\ \text{otherwise} : & \mathsf{up}(f(0),\ \langle n-1, \lambda n \,.\, f(n+1) \rangle) \end{cases} \\
\mathsf{put}(\langle n, f \rangle, u) &= \begin{cases} \text{if } n = \infty : & \langle n, f \rangle \\ \text{otherwise} : & \langle n+1,\ \lambda i \,.\, \text{if } i = n \text{ then } u \text{ else } f(i) \rangle \end{cases} \\
\mathsf{new} &= \langle 0, f_0 \rangle
\end{aligned}$$

where f_0 is an arbitrary function $\mathbb{N} \to U$. Note that at this stage there is nothing that restricts the behaviour of these methods: There exist models of the Queue signature that contain infinite queues (as shown) and there are also models that do not resemble FIFO queues at all.

2.2 Coalgebraic Specification

A coalgebraic class specification consists of a coalgebraic class signature and a set of formulae that restrict the behavior of the methods and constructors of the signature. The models of the specification are those models of the signature that fulfill all formulae. For CCSL we use a standard higher-order logic where the basic propositions are *behavioral equalities*. Intuitively, two terms are behaviorally equal, if they are pointwise equal for constant and parameter types and pointwise bisimilar for the type Self.

The definition of bisimulation works on the inductive structure of the types. The nonconstant type constructors from the ground signature are difficult to handle. For them we need specially constructed fixedpoints in a lattice of relations. The details are explained in [HJ97]. In this paper we want to concentrate on the essentials, therefore we assume in the following a restricted ground signature Ω_r that contains the type constructor Lift, arbitrary type constants but no other type constructors (so $C_1 = \{\mathsf{Lift}\}$ and $C_n = \emptyset$ for $n > 1$). This restriction is harmless, the ground signature Ω_r is sufficient for a large number of applications including the queue example.

Consider a type τ over Ω_r, choose a set X to interpret Self, and fix an interpretation $U_1, \ldots, U_n$ of the type variables in τ. Now a relation $R \subseteq X \times X$ gives rise to a relation

$$\mathrm{Rel}(\tau)(R) \quad \subseteq \quad [\![\tau]\!]^X_{U_1,\ldots,U_n} \times [\![\tau]\!]^X_{U_1,\ldots,U_n}$$

defined by induction on the structure of τ. For suitable x and y the relation $x \; \mathrm{Rel}(\tau)(R) \; y$ holds,

- in case $\tau = \alpha_i$ or $\tau = \mathsf{C}$, for a type constant C, if $x = y$,
- in case $\tau = \mathsf{Lift}[\sigma]$ if either $x = y = \bot$ or if $x = \mathsf{up}(x')$ and $y = \mathsf{up}(y')$ for $x', y' \in [\![\sigma]\!]^X_{U_1,\ldots,U_n}$ and $x' \; \mathrm{Rel}(\sigma)(R) \; y'$
- in case $\tau = \mathsf{Self}$ if $x \, R \, y$
- in case $\tau = \sigma_1 \times \sigma_2$ if $(\pi_1 x) \; \mathrm{Rel}(\sigma_1)(R) \; (\pi_1 y)$ and $(\pi_2 x) \; \mathrm{Rel}(\sigma_2)(R) \; (\pi_2 y)$
- in case $\tau = \sigma_1 + \sigma_2$ if one of the two cases hold
  - $x = \kappa_1 x_1$ and $y = \kappa_1 y_1$ for $x_1, y_1 \in [\![\sigma_1]\!]^X_{U_1,\ldots,U_n}$ and $x_1 \; \mathrm{Rel}(\sigma_1)(R) \; y_1$
  - $x = \kappa_2 x_2$ and $y = \kappa_2 y_2$ for $x_2, y_2 \in [\![\sigma_2]\!]^X_{U_1,\ldots,U_n}$ and $x_2 \; \mathrm{Rel}(\sigma_2)(R) \; y_2$
- in case $\tau = \sigma_1 \to \sigma_2$ if for all $a, b \in [\![\sigma_1]\!]^X_{U_1,\ldots,U_n}$ it holds that $a \; \mathrm{Rel}(\sigma_1)(R) \; b$ implies $(x\,a) \; \mathrm{Rel}(\sigma_2)(R) \; (y\,b)$

The relation $\mathrm{Rel}(\tau)(R)$ is called the relation lifting of R [HJ98]. Note that $x \; \mathrm{Rel}(\tau)(R) \; y \iff x = y$ for constant types τ.

Definition 4. Let Σ be a coalgebraic class signature and assume a model $\mathcal{M} = \langle X, M, C \rangle$ of Σ for a fixed interpretation $U_1, \ldots, U_n$ of the type parameters.

1. A relation $R \subseteq X \times X$ is a Σ–bisimulation for $\mathcal{M}$ if for all methods $m \in M$ of method type σ we have $m \; \mathrm{Rel}(\sigma)(R) \; m$.

2. *Bisimilarity* $\sim_{\mathcal{M}}$ is the largest Σ–bisimulation for $\mathcal{M}$.
3. For a type τ, two elements $x, y \in [\![\tau]\!]^X$ are *behaviorally equivalent* $x \cong_{\mathcal{M}} y$ if $x \ \mathrm{Rel}(\tau)(\sim_{\mathcal{M}}) \ y$ holds.

These definitions of bisimulation and bisimilarity are equivalent to the definitions in [Rut00, JR97] (recall that we consider only signatures without binary methods). This follows for example from Proposition 5.8 in [Tew00b].

Proposition 5. *Consider a coalgebraic class signature Σ and a model $\mathcal{M}$ of Σ. The $\mathcal{M}$–bisimulations are closed under union and intersection. Further, bisimilarity on $\mathcal{M}$ is an equivalence relation. It is given by the union of all bisimulations.*

Proof (Sketch). Prove that relation lifting commutes with union, intersection, and relational composition by induction on types. Then the result is immediate.

Example 4. Let us unroll Definition 4 for the queue example. Assume a model $\langle X, M, C\rangle$ of Σ_{Queue} for a fixed interpretation U of the type parameter α. In the following we use top and put for the functions in M that interpret the queue operations. A relation $R \subseteq X \times X$ is a Σ_{Queue}–bisimulation if for all $x, y \in X$ with $x \, R \, y$ the following two points hold.

- $\forall a \in U \,.\, (\mathsf{put}(x, a)) \ R \ (\mathsf{put}(y, a))$
- either $\mathsf{top}(x) = \mathsf{top}(y) = \bot$ or there exist $x', y' \in X$ and $a \in U$ such that $\mathsf{top}(x) = \mathsf{up}(a, x')$ and $\mathsf{top}(y) = \mathsf{up}(a, y')$ and $x' \, R \, y'$.

Let us denote the greatest queue bisimulation with $\sim_Q$ in the following. Two terms $u, v \in \mathsf{Lift}[U \times X]$ are behavioral equal $u \cong v$ if either of the following cases hold.

- $u = v = \bot$
- there exists $x, y \in X$ and $a \in U$ such that $u = \mathsf{up}(a, x)$ and $v = \mathsf{up}(a, y)$ and $x \sim_Q y$.

For a logic over our coalgebraic class signatures we use an entirely standard higher-order logic over a polymorphic type theory (see for instance [Jac99a]). So for a signature Σ we have the following.

Ground terms are all constants $c : \tau$ from the ground signature and all method symbols $m : \sigma$ and constructor symbols $c : \rho$ from Σ.

Variables We assume an collection of typed variables $(\mathcal{V}_\sigma)$. For any element $x \in \mathcal{V}_\sigma$ we have a term $x : \sigma$.

Equality For any type τ and terms $t_1, t_2 : \tau$ we have the behavioral equality $t_1 \cong t_2 : \mathsf{bool}$ and if τ is a constant type also (as syntactic sugar) $t_1 = t_2 : \mathsf{bool}$

Constructions Terms are closed under abstraction ($\lambda x : \tau . t : \tau \to \sigma$ for $t : \sigma$ and $x \in \mathcal{V}_\tau$), application ($t_1 t_2 : \sigma$ for $t_1 : \tau \to \sigma$ and $t_2 : \tau$), tupeling ($(t_1, \ldots, t_n) : \sigma_1 \times \cdots \times \sigma_n$ for $t_1 : \sigma_1, \ldots, t_n : \sigma_n$), case distinction ($\mathsf{cases}\ t\ \mathsf{of}\ \kappa_1 x : r,\ \kappa_2 y : s : \tau$ for $t : \sigma_1 + \sigma_2$ and $r : \tau$ with free variable $x : \sigma_1$ and $s : \tau$ with free variable $y : \sigma_2$), the boolean connectives $\wedge, \vee, \Rightarrow, \neg$ (for instance $t_1 \vee t_2 : \mathsf{bool}$ for $t_1, t_2 : \mathsf{bool}$) and existential and universal quantification ($\forall x : \tau . t : \mathsf{bool}$ for $x \in \mathcal{V}_\tau$ and $t : \mathsf{bool}$).

Formulae The terms of type bool are called formulae and denoted by $\mathcal{L}(\Sigma)$.

In CCSL and in examples we use the common object-oriented notation and write $t.m(t_1, \ldots, t_n)$ instead of $m(t, t_1, \ldots, t_n)$ for $t : \mathsf{Self}$. The interpretation of terms and formulae is straightforward. Fix a model $\mathcal{M}$ with state space X of Σ. The interpretation of a term $t : \sigma$ with free variables $x_1 : \tau_1, \ldots, x_n : \tau_n$ and k type variables in $\tau_1, \ldots, \tau_n$ and σ is a collection of functions

$$\left(\ [\![t]\!]^X_{U_1,\ldots,U_k} \ : \ [\![\tau_1]\!]^X_{U_1,\ldots,U_k} \times \cdots \times [\![\tau_n]\!]^X_{U_1,\ldots,U_k} \longrightarrow [\![\sigma]\!]^X_{U_1,\ldots,U_k} \ \right)$$

The ground terms are interpreted by the ground signature and by $\mathcal{M}$. The model $\mathcal{M}$ gives rise to behavioral equality $\cong_{\mathcal{M}}$ that is used for $\cong$. The interpretation of a term t in the model $\mathcal{M}$ is denoted with $[\![t]\!]^{\mathcal{M}}$.

Formulae from $\mathcal{L}(\Sigma)$ that contain method or constructor symbols state properties of these operations. We distinguish between *assertions* and *creation conditions*. Assertions are formulae that restrict the behavior of the methods of the class. They apply to all objects of the class (i.e., the assertion should hold for all elements of the state space). Formally an assertion is a formula that contains one free variable of type Self and no constructor symbols from Σ. Creation conditions restrict the behavior of the constructors of a class. Typically a specification contains one creation condition per constructor symbol. Formally a creation condition is a closed formula.

Definition 6. Let Σ be a coalgebraic class signature.

1. A coalgebraic class specification is a triple $\langle \Sigma, \mathcal{A}, \mathcal{C} \rangle$, where Σ is a coalgebraic class signature, $\mathcal{A}$ is a finite set of assertions, and $\mathcal{C}$ is a finite set of creation conditions, both being sets of formulae from $\mathcal{L}(\Sigma)$.

2. A model $\mathcal{M} = \langle X, M, C \rangle$ of Σ is a model of the specification $\langle \Sigma, \mathcal{A}, \mathcal{C} \rangle$ if for all interpretations of the type parameters of Σ it holds that for all $F \in \mathcal{A}$ we have $\forall x : X . [\![F]\!]^{\mathcal{M}}(x)$ and additionally $[\![G]\!]^{\mathcal{M}}$ is true for all $G \in \mathcal{C}$.

Note that assertions and creation conditions can be combined via conjunction without changing the semantics. But in applications it is nicer to capture different properties in different formulae.

Example 5. In Example 1 we described the Queue–signature. To specify the behavior of FIFO queues we use two assertions and one creation condition. The first assertion tells something about empty queues, a queue q is considered to be empty if the top methods fails on it (i.e., if $q.\mathsf{top} \cong \bot$).

$$F_{\mathsf{empty}}(q) \quad = \quad \left[\quad q.\mathsf{top} \cong \bot \ \ \Rightarrow \ \ \forall a : \alpha \,.\, q.\mathsf{put}(a).\mathsf{top} \cong \mathsf{up}(a, q) \quad \right]$$

```
BEGIN Queue[ A : TYPE ] : CLASSSPEC
  METHOD
    put : [Self, A] -> Self;
    top : Self -> Lift[[A,Self]];

  CONSTRUCTOR
    new : Self;

  ASSERTION SELFVAR x : Self
    q_empty : x.top ≅ ⊥ IMPLIES
      FORALL(a : A) . x.put(a).top ≅ up(a,x);

    q_filled :
      FORALL(a1:A, y : Self) . x.top ≅ up(a1,y) IMPLIES
        FORALL(a2 : A) . x.put(a2).top ≅ up(a1, y.put(a2));

  CREATION
    q_new : new.top ≅ ⊥;
END Queue
```

Figure 1: *The* CCSL *specification for queues*

So if the queue is empty we demand that $q.\mathsf{put}(a).\mathsf{top}$ is always successful (i.e., it never equals $\bot$) and that it returns a pair (b, q') where $a = b$ and q' is an empty queue again. The second assertion restricts nonempty queues.

$$F_{\mathsf{filled}}(q) \quad = \quad \left[\begin{array}{l} \forall a_1 : \alpha, q' : \mathsf{Self}\,.\, q.\mathsf{top} \cong \mathsf{up}(a_1, q') \;\Rightarrow \\ \qquad\qquad \forall a_2 : \alpha\,.\, q.\mathsf{put}(a_2).\mathsf{top} \;\cong\; \mathsf{up}(a_1, q'.\mathsf{put}(a_2)) \end{array} \right]$$

This says that if q is nonempty, then the two operations of adding an element (at the end!) and of removing the first element are interchangeable. In the creation condition we demand that the constructor new delivers an empty queue

$$F_{\mathsf{new}} \quad = \quad \left[\;\; \mathsf{new.top} \cong \bot \;\; \right]$$

The full queue specification is $\langle \Sigma_{\mathsf{Queue}}, \{F_{\mathsf{empty}}, F_{\mathsf{filled}}\}, \{F_{\mathsf{new}}\}\rangle$, it is shown in CCSL syntax in Figure 1 on page 11.

3 The Specification Language CCSL and its Compiler

In this section, we introduce the Coalgebraic Class Specification Language CCSL. This specification language allows to specify classes in a coalgebraic way as formally described in Definition 6. We also introduce a compiler for CCSL that translates those class specifications into logical theories for theorem provers and thus allows the mechanical verification of large class specifications. This verification may involve the construction of models and refinements, and theory development.

Coalgebraic specifications according to Definition 6 form the most important ingredient of CCSL. A coalgebraic class specification needs only few syntactic changes and annotations to be a valid CCSL specification. The concrete syntax of CCSL is very close to PVS. Figure 1 contains the CCSL version of Example 5. In CCSL the type parameters must be declared after the name of the specification (line 1). The keyword **CLASSSPEC** tells the compiler that the following is a coalgebraic class specification. In CCSL the product of two types is written with brackets: $[\tau_1, \tau_2]$, so Lift$[[A, \mathbf{Self}]]$ is Lift applied to $A \times$ Self. The declaration of method symbols starts with the keyword **METHOD** on line 2, then the constructor symbols follow and finally we have the two assertions and the creation condition. The variable of type Self that occurs freely in the assertions is declared with the keyword **SELFVAR** before the first assertion.

To give the user support for verifying specifications, 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 (ISABELLE and PVS at the moment). The compiler is written in OCAML [LDG+00]. OCAML is the French dialect of the functional programming language ML enriched with object-oriented features. Besides CCSL, the tool does also support JAVA [JvdBH+98] and JML(annotated JAVA) [LBR99]. For input files the compiler generates appropriate theories for PVS and ISABELLE. We shall focus on CCSL input and PVS output.

The front-end tool is, following standard compiler construction techniques, organized in several passes. These passes act on intermediate internal data structures. In the first pass, the compiler parses the source code and generates an internal representation of the input. Internally, classes and methods are stored in OCAML classes. The types that appear in the class signatures are encoded as elements of an abstract data type `top_types` in OCAML. For instance the type Self $\times A \rightarrow$ Self is internally represented as

```
Function(Product(Self,TypeVariable("A")), Self)
```

Here `Function`, `Product`, `Self` and `ConstantType` are some of the type constructors of the OCAML type `top_types`. They represent the corresponding type constructors from CCSL. Similarly, formulae are represented as members of an abstract data type `top_formulae`.

In the second pass, the tool checks the type correctness of formulae that appear in the assertions. The current version of this pass is a simple recursive type checker that can check the type of a term if the types of all its sub-terms are given. Ground terms that have an ambiguous type (like the injections κ_i) must be annotated with the correct type by the user. A future version of the compiler will contain a better type checker.

After type checking, the tool generates an internal representation of the output theories. In the fourth pass, these theories are written into the output files in the desired format (that is ISABELLE or PVS). This separation from pass three makes it easy to add support for other theorem provers for higher-order logic.

The most basic theory that the compiler generates from a class specification is the interface theory. Figure 2 depicts the PVS-output for our queue example. In the first line, you find the name of the theory and its parameters. As Definition 3

```
QueueInterface[Self : TYPE , A : TYPE] : THEORY
BEGIN
  IMPORTING Lift[[A , Self]]

  QueueSignature : TYPE = [#
      put : [[Self , A] -> Self] ,
      top : [Self -> Lift[[A , Self]]]
    #]
  QueueConstructors : TYPE =  [# new : Self #]
END QueueInterface
```

Figure 2: *The* PVS *theory for the queue interface*

suggests the type Self is semantically treated as a special type variable. Therefore the PVS theory QueueInterface is parametric in two types. In PVS the type constructor Lift is formalized as a parametric abstract data type. This type is imported in line 3. Lines 5-8 capture the interface of the methods of the queue and Line 9 describes the interface of its constructors.

Based on the interface theory, the compiler also generates definitions of invariants, homomorphisms between coalgebras for a class specification, bisimilarity, and behavioral equalities. The theories that describe the assertions use these notions, especially behavioral equality. To increase the verification support also standard lemmas and axioms about invariants and bisimilarity are generated. They are very useful when one tries to prove properties of a specified class. These definitions also serve as tools for the concepts of model and refinement between class specifications.

Altogether the theories that the compiler generates have about 30 times the size of the original CCSL specification. It would be infeasible to do the translation of CCSL into higher-order logic by hand.

4 Modal Operators

Modal operators[5] are well suited for the verification of properties of potentially nonterminating systems. For this reason, we incorporated these operators into our logic. In addition to the constructions in Section 2 we allow the following:

Constructions (cont.) Terms are further closed under modal operators $\Box\ P : \mathsf{Self} \to \mathsf{bool}$ and $\Box^{\{m_1,\ldots,m_n\}}\ P : \mathsf{Self} \to \mathsf{bool}$, where $P : \mathsf{Self} \to \mathsf{bool}$ and $m_1, ..., m_n$ are method symbols from the coalgebraic class signature Σ.

The construction $\Box\ P$ shall be interpreted as a predicate that holds in those states, for which all reachable successor states fulfill P. This corresponds to AGP from CTL [Eme90]. In the second version reachability is restricted to transitions via specific methods $m_1, ..., m_n$. This is especially useful if in a specification some

[5] Some people distinguish between modal and temporal logic. Following a long tradition, we consider temporal operators as special cases of modal operators.

methods $m_1, ..., m_n$ maintain a property and others do not. The corresponding dual to $\Box$ is $\Diamond$, which is defined as syntactic sugar

$$\Diamond^M P(x) \quad = \quad \neg \Box^M \big(\lambda y : \mathsf{Self}.\, \neg P(y)\big)(x)$$

where $M = \{m_1, \ldots, m_n\}$. Observe that we can not use the usual $\neg \Box^M \neg P(x)$, as negation is defined for boolean values only and not for predicates.

Example 6. Again, consider our queue class specification. Using the modal operators, we can express the property, that every queue that was built from an empty queue can be made empty again:

$$\forall x : \mathsf{Self}.\, x.\mathsf{top} \cong \bot \quad \Rightarrow \quad \Box \big(\Diamond\, (\lambda y : \mathsf{Self}.\, y.\mathsf{top} \cong \bot)\big)(x)$$

This property can easily be proved by observing that every queue that was built from an empty queue can be emptied by a number of top steps. This number must be equal to the number of elements in the queue.

The semantics of $\Box$ P is defined in terms of greatest invariants contained in the interpretation of P. (This connection was first recognized in [Jac99b]. A more detailed elaboration of this topic can be found in [Rot00].) For defining invariants, we need an analogue to relation lifting from Section 2 for predicates.

Consider a type τ over the restricted ground signature Ω_r, choose a set X to interpret Self, and fix an interpretation $U_1, \ldots, U_n$ of the type variables in τ. Now a predicate $P \subseteq X$ gives rise to a predicate $\mathrm{Pred}(\tau)(P) \subseteq [\![\tau]\!]^X_{U_1,\ldots,U_n}$ defined by induction on the structure of τ. For an element $x \in [\![\tau]\!]^X_{U_1,\ldots,U_n}$ it holds that $x \in \mathrm{Pred}(\tau)(P)$

- always in case $\tau = \mathsf{C}$ or $\tau = \mathsf{C}$
- in case $\tau = \mathsf{Self}$ if $x \in P$
- in case $\tau = \mathsf{Lift}[\sigma]$, if $x = \bot$ or $x = \mathsf{up}(x')$ and $x' \in \mathrm{Pred}(\sigma)(P)$.
- in case $\tau = \sigma_1 \times \sigma_2$ if $(\pi_1 x) \in \mathrm{Pred}(\sigma_1)(P)$ and $(\pi_2 x) \in \mathrm{Pred}(\sigma_2)(P)$
- in case $\tau = \sigma_1 + \sigma_2$ we have either $x = \kappa_1 x_1$ or $x = \kappa_2 x_2$ for $x_i \in [\![\sigma_i]\!]^X_{U_1,\ldots,U_n}$. In the first case it must hold that $x_1 \in \mathrm{Pred}(\sigma_1)(P)$ and in the second case that $x_2 \in \mathrm{Pred}(\sigma_2)(P)$.
- in case $\tau = \sigma_1 \to \sigma_2$ if for all $a \in [\![\sigma_1]\!]^X_{U_1,\ldots,U_n}$ it holds that $a \in \mathrm{Pred}(\sigma_1)(P)$ implies $x(a) \in \mathrm{Pred}(\sigma_2)(P)$

The predicate $\mathrm{Pred}(\tau)(P)$ is called the predicate lifting of τ.

Definition 7. Assume a coalgebraic class signature Σ and let $\mathcal{M} = \langle X, M, C\rangle$ be a model of Σ. A predicate $P \subseteq X$ is a *Σ–invariant for $\mathcal{M}$* if for all methods $m \in M$ of method type σ we have $m \in \mathrm{Pred}(\sigma)(P)$.

Proposition 8. *Consider a coalgebraic class signature Σ and a model $\mathcal{M}$. The Σ–invariants in $\mathcal{M}$ are closed under union and intersection. Especially the empty and the full set are Σ–invariants.*

Proof. Again, by induction prove that predicate lifting commutes with union and intersection. Then the desired result imediately follows. The proposition about full and empty sets are straight-forward.

So there exists always a greatest invariant contained in P. We shall denote this greatest invariant contained in some P by $\underline{P}$.

Definition 9 Semantics of $\Box$. Assume a coalgebraic class signature Σ and a model $\mathcal{M} = \langle X, M, C \rangle$. As announced above, the type of $\Box\ P$ is $\mathsf{Self} \to \mathsf{bool}$ for $P : \mathsf{Self} \to \mathsf{bool}$. For free variables $x_1 : \tau_1, \ldots, x_n : \tau_n$ in P, such that there are at most k type variables in $\tau_1, \ldots, \tau_n$, the interpretation of $\Box\ P$ is a collection of functions

$$\Big(\ [\![\Box\ P]\!]^X_{U_1,\ldots,U_k}\ :\ [\![\tau_1]\!]^X_{U_1,\ldots,U_k} \times \cdots \times [\![\tau_n]\!]^X_{U_1,\ldots,U_k} \longrightarrow X \longrightarrow [\![\mathsf{bool}]\!]\ \Big)$$

indexed by the possible interpretations for the type variables $U_1, \ldots, U_k$. It is defined by

$$[\![\Box\ P]\!]^X_{U_1,\ldots,U_k}(x_1, \ldots, x_n) \quad = \quad \Big\{ x \in X \text{ such that } x \in \underline{[\![P]\!]^X_{U_1,\ldots,U_k}(x_1, \ldots, x_n)} \Big\}$$

with $(x_1, \ldots, x_n) \in [\![\tau_1]\!]^X_{U_1,\ldots,U_k} \times \cdots \times [\![\tau_n]\!]^X_{U_1,\ldots,U_k}$. That is, the interpretation of $\Box\ P$ (with respect to some free variables and type variables) is the greatest invariant contained in the interpretation of P (with respect to the interpretation of those free variables and type variables).

For modal operators that are restricted to some methods, the interpretation requires suitable restrictions in the definition above. The main idea is to construct a sub-signature of the class signature that is determined by the methods $m_1, \ldots, m_n$. This signature gives rise to its own (sub-)lifting and, thus, own (greatest) (sub-)invariants. These are used to define the semantics of $\Box^{\{m_1,\ldots,m_n\}}\ P$ in the same way as above. To find a fully worked-out version, see [Rot00]. There you also find some calculus rules for $\Box$, that characterize it as "S4" operator, and a characterization result of $\Box$ via computation paths.

As you may expect, our front-end tool generates appropriate theories with the definitions of the modal operators for a class specification. It further generates lemmas that help when reasoning about modal properties.

5 Advanced CCSL

In this section we informally sketch some other features of CCSL and/or coalgebraic specification.

5.1 Inheritance and Late Binding

The notions of inheritance, overriding and late binding are usually seen as the key concepts of object orientation. However it is not completely clear what these concepts should mean for a specification language. For instance the term *late* *binding* refers to the choice of method bodies at runtime depending on the dynamic type of the objects. But a specification language abstracts from implementation details and has no method bodies. For CCSL we decided to support all possible flavors of inheritance, overriding and late binding on an abstract level instead of following one specific language.

Multiple inheritance of specifications can be modeled by inclusion of class signatures. In CCSL we provide the syntax

INHERIT FROM Queue[bool] **RENAMING** top **AS** get

to include the method declarations from the Queue specification (thereby instantiating it to a queue of booleans and renaming the method top). For more details about overriding and late binding we refer the reader to [HHJT98].

5.2 Components and Abstract Data Types

Via components objects can contain other objects (of different classes). After processing the Queue specification the compiler enriches the actual ground signature with a type constructor Queue of arity one (because this specification has only one type parameter). In a specification that follows the queue specification types can therefore contain Queue. So that

queues : [**Self**, nat] –> Queue[A];

is then a valid method declaration in CCSL (describing a method with which the user can access an infinite array of queues in each object). For the semantics of the type constructor Queue the user can choose between *loose semantics* and *final semantics*. For loose semantics the interpretation of Queue will be an arbitrary model of the Queue specification. For final semantics it will be the final model[6].

Besides coalgebraic class specifications CCSL allows also abstract data type definitions (which have not been discussed yet). The type of binary trees with different labels on branches and leaves, for instance, looks in CCSL like

```
BEGIN BTree[A, B : TYPE] : ADT
  CONSTRUCTOR
    leaf : A -> CARRIER;
    branch : [CARRIER, B, CARRIER] -> CARRIER;
END BTree
```

The keyword **CARRIER** stands (similar to Self) for the type being defined. For abstract data types we use an initial semantics. So after processing the binary trees, the ground signature is enriched with a type constructor BTree of arity 2. Its semantics is the carrier of the initial algebra for the BTree signature. The type constructor BTree can be used subsequently to declare methods that take or deliver binary trees. In proving properties for such methods one has to nest

[6] The final model can be understood as the union of all possible models factorized by bisimilarity. It can mimic every possible behavior and further no two different states of the final model show the same behavior. For class signatures without binary methods there exists always a final model [Rut00]. Under additional assumptions on the assertions, it can be shown that every consistent specification without binary methods has a final model. (The use of *final* refers here to the final object in the category of all models of Queue.)

an induction proof inside a coinduction. Currently one cannot add assertions to an abstract data type specification in CCSL.

5.3 Binary Methods

Binary methods are operations that receive two (or more) objects of the current class instead of only one [BCC+95]. A typical example is $\mathsf{equal} : \mathsf{Self} \times \mathsf{Self} \longrightarrow \mathsf{bool}$. For CCSL we use the more general definition (compared to [Rut00]) of coalgebras from [Tew00b]. In the definition of types and class signatures in this paper we also follow the more general approach that allows for binary methods. But for bisimilarity and invariants we gave simpler definitions that do not work properly if the signature contains binary methods. However, the CCSL compiler 'knows' the proper definitions, so the user can specify classes with binary methods. In the presence of binary methods the notions of bisimilarity and invariants behave not as nicely as for traditional coalgebras. For instance bisimulations are not closed under union and a final model does usually not exist. For details about bisimulations and bisimilarity see [Tew00b], more results about invariants will appear elsewhere.

5.4 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 are usually 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 fulfills the assertions of the original abstract specification.

For coalgebraic class specifications we have two notions of refinement. *Model theoretic refinement* is a generalization 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. Details will appear elsewhere.

6 Conclusion

In this paper we presented the coalgebraic class specification language CCSL and sketched its possible applications. CCSL allows to mix class specifications with abstract data-type specifications. For the class specifications it relies on coalgebras and the notions of behavioral equality and invariance.

The specification language CCSL has been successfully used in several nontrivial case studies. Meyer specifies in [Mey99] the MSMIE (multi-processor 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. Lambooij studies in [Lam00] the Y–chart Application Programmers Interface (YAPI) protocol for Kahn processing networks. He developed a specification in CCSL and proves with PVS the correctness of the data transfer and the absence of deadlock for the YAPI protocol.

In [Tew00a], the second author uses CCSL to formalize parts of the memory management of the micro-kernel operating system 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. This case study proves that the specification language CCSL together with the theorem prover PVS can well be applied to *real* software.

In the future we plan a public release of the CCSL compiler. In our current research we try to improve the support of CCSL for the specification of imperative object-oriented programs. Further we try to apply the theory of traces to coalgebras to model systems of objects.

References

[BCC+95] K. Bruce, L. Cardelli, G. Castagna, The Hopkins Object Group, G. T. Leavens, and B. Pierce. On binary methods. *Theory and Practice of Object Systems*, 1(3):221–242, 1995.

[Cîr99] C. Cîrstea. A coalgebraic equational approach to specifying observational structures. In B. Jacobs and J. Rutten, editors, *Coalgebraic Methods in Computer Science*, volume 19 of *ENTCS*. Elsevier, 1999. Available via http://www.elsevier.nl/locate/entcs/.

[Eme90] E. A. Emerson. Temporal and modal logic. In Jan van Leeuwen, editor, *Handbook of Theoretical Computer Science, Volume B: Formal Models and Semantics*, pages 995–1072. Elsevier Science Publishers, 1990.

[HHJT98] 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*, number 1381 in LNCS, pages 105–121. Springer, 1998.

[HJ97] U. Hensel and B. Jacobs. Proof principles for datatypes with iterated recursion. In E. Moggi and G. Rosolini, editors, *Category Theory and Computer Science*, number 1290 in LNCS, pages 220–241. Springer, 1997.

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

[HK99] R. Hennicker and A. Kurz. (Ω, Ξ)–Logic: On the algebraic extension of coalgebraic specifications. In B. Jacobs and J. Rutten, editors, *Coalgebraic Methods in Computer Science*, volume 19 of *ENTCS*, pages 195–212. Elsevier, 1999. Available via http://www.elsevier.nl/locate/entcs/.

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

[Jac99a] B. Jacobs. *Categorical Logic and Type Theory.* Number 141 in Studies in Logic and the Foundations of Mathematics. North Holland, Elsevier, 1999.

[Jac99b] B. Jacobs. The temporal logic of coalgebras via galois algebras. Technical Report CSI-R9906, Computing Science Institute, University of Nijmegen, 1999. To appear in Mathematical Structures in Computer Science.

[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 (OOPSLA)*, pages 329–340. ACM Press, 1998.

[Lam00] P. Lambooij. The YAPI protocol for buffered data transfer. Techn. Rep. CSI-R9923, Comput. Sci. Inst., Univ. of Nijmegen. Available at URL http://www.cs.kun.nl/csi/reports/info/CSI-R9923.html, 2000.

[LBR99] G.T. Leavens, A.L. Baker, and C. Ruby. JML: A notation for detailed design. In H. Kilov and B. Rumpe, editors, *Behavioral Specifications of Business and Systems*, pages 175–188. Kluwer, 1999.

[LDG+00] X. Leroy, D. Doligez, J. Garrigue, D. Rémy, and J. Vouillon. *The Objective Caml system, release 3.00*, April 2000. available via http://caml.inria.fr/ocaml/.

[Mey99] Dion Meyer. A case study in object oriented specification and verification: The MSMIE protocol. Master's thesis, University of Nijmegen, 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 LNCS, pages 411–414. Springer, 1996.

[Pau94] L. C. Paulson. *Isabelle: A Generic Theorem Prover.* Number 828 in LNCS. Springer, 1994.

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

[Roşu00] Grigore Roşu. *Hidden Logic.* PhD thesis, University of California at San Diego, 2000.

[Rot00] J. Rothe. Modal logics for coalgebraic class specification. Master's thesis, TU Dresden, Germany, 2000. Available at URL: http://wwwtcs.inf.tu-dresden.de/~janr/diplom.ps.gz.

[Rut00] J. J. M. M. Rutten. Universal coalgebra: a theory of systems. *Theoretical Computer Science*, 249(1):3–80, October 2000.

[Tew00a] H. Tews. A case study in coalgebraic specification: Memory management in the FIASCO microkernel. 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, 2000. Available via http://www.elsevier.nl/locate/entcs/.