Invariants, Bisimulations and the Correctness of Coalgebraic Refinements*

Bart Jacobs

Dep. Comp. Sci., Univ. Nijmegen, P.O. Box 9010, 6500 GL Nijmegen, The Netherlands.

Email: bart@cs.kun.nl Tel: ++31 – 24 – 3652236 Fax: ++31 – 24 – 3553450

Abstract. *Coalgebraic specifications are used to formally describe the behaviour of classes in object-oriented languages. In this paper, a general notion of refinement between two such coalgebraic specifications is defined, capturing the idea that one "concrete" class specification realises the behaviour of the other, "abstract" class specification. Two (complete) proof-techniques are given to establish such refinements: one involving an invariant (a predicate that is closed under transitions) on the concrete class, and one involving a bisimulation (a relation that is closed under transitions) between the concrete and the abstract class. The latter can only be used if the abstract class is what we call totally specified. Parts of the underlying theory of invariants and bisimulations in a coalgebraic setting are included, involving least and greatest invariants and connections between invariants and bisimulations. Also, the proof-principles are illustrated in examples (which are fully formalised and verified in* PVS*).*

1 Introduction

This paper is part of a recent research line of applying coalgebraic and coinductive notions and techniques in the formalisation of object-oriented concepts, see [26, 14, 11, 13, 15, 5, 6], building on earlier work [30, 2, 17]. Coalgebras consist of a state space together with a transition function and can be used to describe various kinds of dynamical systems, including automata, transition systems and hybrid systems, see *e.g.* [28, 22, 12] (or [16] for an introduction to the theory of coalgebras). A coalgebraic specification (as developed in [14]) formally captures several crucial aspects of classes in object-oriented languages: it consists of a (hidden) state space (typically written as X) to which a client only has limited access, together with several operations (or methods) which act on X. These operations may be attributes $X \to A$ taking values in a (constant) set A and giving some information about elements of X, or they may be procedures for modifying states. The effect of such modification is not directly visible, but only indirectly via the attributes. Procedures may be of the form $X \times B \to X$, but also of the more complicated form $X \times B \to A + (C \times X)$, where $+$ is a disjoint union (or variant) type constructor. In this case the outcome of a procedure application (with parameter from B) can either be an observable value in A, or a pair consisting of an observable value in C together with a next state (in X). In a typical situation, A is a singleton set so that we have a partially defined operation. Besides these methods, a coalgebraic specification consists of two parts: assertions (or constraints) describing the meaning of the operations, and creation conditions describing constraints for initial states (typically written as new). We refer the reader to [14, 13, 15]

*From: M. Johnson (ed.) *Algebraic Methodology and Software Technology* (Springer LNCS 1349, Berlin 1997) p. 276-291.

for more background information. Hopefully, the example specifications in this paper are self-explanatory.

The work that we are about to describe leans heavily on our earlier paper [11] (and extends the approach of [15]). There we described how to obtain terminal coalgebras satisfying constraints (and proved a "comonadicity" result). It took us some time to get the right perspective on (and see the further importance of) the notions and techniques developed there. The terminal coalgebras satisfying constraints were obtained in [11] via special predicates on state spaces of coalgebras that were closed under transitions of the coalgebra. At the time they were called "mongruences", in analogy with congruences (which are relations that are closed under operations). Only now we realise that they are best called "invariants". The terminal coalgebras satisfying constraints were constructed via greatest invariants implying the constraints (see also Section 4 below). These greatest invariants are in essence greatest fixed points of suitable form. Another recent insight is that these greatest invariants come with a proof-rule, which can be exploited to prove the correctness of refinements. This will be the main topic of the present paper. It will require us to further develop the theory of invariants, and to establish some (standard) connections between invariants and bisimulations, see Section 3.

Invariants have a long tradition in computer science. For example, as loop invariants in correctness proofs in the style of Floyd and Hoare. But also in automata-based verification, see *e.g.* [21]. And they occur in an object-oriented setting in [20] in the form of "representation invariants" for implementations of data abstractions in CLU. There is some variation in what precisely constitutes an invariant: it can be (1) a predicate which is closed under transitions (this is what we shall use); or (2) a predicate which holds for all reachable states; or sometimes also (3) a predicate which is closed under transitions and holds for all initial states. Clearly, (3) implies (2), and (2) implies (1). Invariants are well-established as a proof-method for refinements (or implementations) between automata, see *e.g.* [18] for a discussion. Here we extend this approach to a coalgebraic setting, thereby giving it a firm semantical basis.

The present work on refinement, with its invariant and bisimulation proof-principles, shows that the theory of coalgebraic specifications has reached a certain level of maturity and applicability. The following two points appear as advantages of coalgebraic specifications. (1) On the one hand, their formalism is reasonably close to actual object-oriented programming: it is not hard to recognise attributes $X \to A$ as instance variables (of type A), and methods $X \times B \to X$ (or even $X \times B \to A + (C \times X)$) as procedures (possibly with exceptions). The assertions in a coalgebraic specification describe the observable effect of a state change caused by a procedure invocation in a precondition-effect like style. (2) On the other hand, it is also relatively straightforward to describe coalgebraic specifications in a formal (logical) system, like that of the proof-tool PVS [25]. The main refinement example in this paper (Propositions 5.4) has been fully checked in PVS. A systematic approach to such formal, tool-assisted reasoning about class specifications (and refinements between them) will be described elsewhere.

The paper is organised as follows. We start in Section 2 with an illustration of a refinement between coalgebraic specifications, which is meant to convey the main ideas. Subsequently in Sections 3 and 4 we describe the semantics of coalgebraic specifications involving (polynomial) functors (capturing signatures of methods), natural transformations (corresponding to transformations of signatures), coalgebras (models, consisting of state spaces with methods), terminal coalgebras as minimal realisations, invariants and bisimulations. This will be used for a classification of coalgebraic specifications into three categories: they can be inconsistent, totally specified or underspecified. Part of these Sections 3 and 4 are somewhat technical, and readers may wish to first have a look at the definition of refinement and the associated proof-principles in Section 5.

2 An illustration

We present a very simple example of a "concrete" class specification, refining an "abstract" class specification, see Figure 1. First we explain the abstract specification. It has a value attribute val taking real numbers as values, and an increment procedure inc. The latter is not perfect, as becomes clear from the assertion: for an arbitrary state s, the value $\mathsf{s.inc.val}$ after an increment is not precisely the original value $\mathsf{s.val}$ plus one, but all that we are told is that it lies in the (real number) interval $[\mathsf{s.val} + \frac{3}{4}, \mathsf{s.val} + \frac{5}{4}]_{\mathbb{R}}$. Hence inc can be understood as having tolerance $\frac{1}{4}$ with respect to the perfect increment operation. The initial state new, which is the object that is returned when a new instance of the class is created, satisfies the "creation" condition that its value is between 0 and 1. We clearly have an example of an *underspecified* class, since the behaviour of its objects is not fully determined by the specification.

class spec: Abstract
  methods:
    $\mathsf{val}\colon X \longrightarrow \mathbb{R}$
    $\mathsf{inc}\colon X \longrightarrow X$
  assertions:
    $\mathsf{s.inc.val} \in [\mathsf{s.val} + \frac{3}{4}, \mathsf{s.val} + \frac{5}{4}]_{\mathbb{R}}$
  creation:
    $\mathsf{new.val} \in [0,1]_{\mathbb{R}}$
end class spec
	class spec: Concrete
	  methods:
	    $\mathsf{val}\colon X \longrightarrow \mathbb{R}$
	    $\mathsf{inc}\colon X \longrightarrow X$
	    $\mathsf{dec}\colon X \longrightarrow X$
	  assertions:
	    $\mathsf{s.dec.val} \in [\mathsf{s.val} - \frac{5}{4}, \mathsf{s.val} - \frac{3}{4}]_{\mathbb{R}}$
	    $\mathsf{s.val} < 0 \vdash \mathsf{s.inc.val} \in$
	        $[\mathsf{s.val} + \frac{1}{2}, \mathsf{s.val} + \frac{3}{2}]_{\mathbb{R}}$
	    $\mathsf{s.val} \geq 0 \vdash \mathsf{s.inc.val} \in$
	        $[\mathsf{s.val} + \frac{7}{8}, \mathsf{s.val} + \frac{9}{8}]_{\mathbb{R}}$
	  creation:
	    $\mathsf{new.val} = 2\frac{1}{2}$
	end class spec

Figure 1: An abstract and a concrete class specification

The concrete specification in Figure 1 has one attribute val and two procedures inc and dec for incrementing and decrementing the value. Both of them are imperfect, with the imprecision of the increment operation depending on the value of the state on which it operates. This is also a case of underspecification.

Suppose we are in a situation where we need a class which behaves as described in the "abstract" specification, but all we have is a class satisfying the "concrete" specification. Then we do not really have a problem, since this available concrete class provides us with the required "abstract" behaviour: if we take $\mathsf{r} = \mathsf{new.dec.dec}$ in the concrete class as initial state (for the abstract specification that we are trying to realise), and if we only use the increment operation (of the concrete specification), then we get the required behaviour. This is because by using only the increment operation starting from this state r (with value $\mathsf{r.val} \in [0,1]_{\mathbb{R}}$), the value will always be positive, so that the inc procedure is sufficiently precise. We conclude that the concrete class specification *refines* (sometimes also called *implements*) the abstract specification.

The informal argument that we have just given can be formalised as follows: the operations of the abstract specification also exist in the concrete specification. This is obvious in this situation, by the choice of names, but it may require an actual translation. Further, there is a predicate P on the states of the concrete specification which satisfies the following three properties.

(i) P is closed under the "abstract" procedures (in this case only the inc operation);

this expresses that P is an *invariant*.

(ii) there is a special, "reachable" state r in P which satisfies the creation condition of the abstract specification;

(iii) the states in P satisfy the assertion of the abstract specification.

The predicate P which satisfies these conditions is given by $P(\mathsf{s}) \Leftrightarrow \mathsf{s.val} \geq 0$. These three points will be developed into a (complete) proof-method for refinements in the next few sections.

3 Logical preliminaries: invariants and bisimulations for a polynomial functor

Invariants and bisimulations are special predicates and relations on state spaces (of state-based dynamical systems, in general). We start by recalling some basic facts from the logic of predicates and relations on sets. The main subsequent achievement is the definition for an arbitrary predicate P of the least (or weakest) invariant $\overline{P}$ implied by P and of the greatest (or strongest) invariant $\underline{P}$ implying P. These predicates $\overline{P}$ and $\underline{P}$ will play an important rôle in our proof principles.

We shall be working in the (ordinary) universe of discourse given by the category **Sets** of sets and functions. Of primary importance will be the structure given by Cartesian products $X \times Y$ (with projections $X \stackrel{\pi}{\leftarrow} X \times Y \stackrel{\pi'}{\rightarrow} Y$), coproducts (or disjoint unions) $X + Y$ (with coprojections $X \stackrel{\kappa}{\rightarrow} X + Y \stackrel{\kappa'}{\leftarrow} Y$), and exponents X^Y (with evaluation maps $\mathrm{ev}\colon X^Y \times Y \to X$). We often write 1 for a singleton set (with typical element $*$), and 0 for the empty set.

For an arbitrary set X we shall write $\mathcal{P}(X)$ for the poset (category) of subsets of X, ordered by inclusion. Such subsets will be identified with predicates on X, and the inclusion between them with implication. For a function $f\colon X \to Y$ we get a monotone function (*i.e.* a functor) $f^*\colon \mathcal{P}(Y) \to \mathcal{P}(X)$ by $(Q \subseteq Y) \mapsto (\{x \mid f(x) \in Q\} \subseteq X)$, which is commonly called *substitution along* f. It preserves the propositional connectives, like $\top, \wedge, \bot, \vee, \neg$. A basic observation is that this f^* has both a left and a right adjoint $\coprod_f \dashv f^* \dashv \prod_f$ in a situation:

$$\begin{tikzcd}
\mathcal{P}(X) \arrow[r, bend left=30, "\prod_f"] \arrow[r, bend right=30, "\coprod_f"'] \arrow[r, phantom, bend left=15, "\top"] \arrow[r, phantom, bend right=15, "\top"] & \mathcal{P}(Y) \arrow[l, "f^*" description] \\
X \arrow[r, "f"'] & Y
\end{tikzcd}$$

The adjoints $\coprod_f$ and $\prod_f$ are given by the formulas:

$$\begin{array}{rcl}
\coprod_f(P \subseteq X) & = & (\{y \mid \exists x \in X\ f(x) = y \ \&\ x \in P\} \subseteq Y) \\
\prod_f(P \subseteq X) & = & (\{y \mid \forall x \in X\ f(x) = y \Rightarrow x \in P\} \subseteq Y).
\end{array}$$

The adjunctions $(\coprod_f \dashv f^*)$ and $(f^* \dashv \prod_f)$ are in essence Galois connections, and boil down to the following equivalences:

$$\coprod_f(P) \subseteq Q \;\Leftrightarrow\; P \subseteq f^*(Q) \qquad \text{and} \qquad f^*(Q) \subseteq P \;\Leftrightarrow\; Q \subseteq \prod_f(P).$$

It is not hard to see that for composable functions f, g one has $(g \circ f)^* = f^* \circ g^*$, $\coprod_g \circ \coprod_f = \coprod_{g \circ f}$ and $\prod_g \circ \prod_f = \prod_{g \circ f}$.

We restrict our attention to so-called polynomial (endo)functors $T: \mathbf{Sets} \to \mathbf{Sets}$ on the category $\mathbf{Sets}$. These functors are defined inductively, as the elements of the least class of functors $\mathbf{Sets} \to \mathbf{Sets}$ containing[1]:

1. the identity functor $\mathrm{id}: \mathbf{Sets} \to \mathbf{Sets}$;
2. the constant functors $K_A: \mathbf{Sets} \to \mathbf{Sets}$ (for each set A), mapping $X \mapsto A$;
3. the product $X \mapsto T_1(X) \times T_2(X)$ and coproduct $X \mapsto T_1(X) + T_2(X)$ functors, for functors T_1, T_2 already belonging to the class;
4. the exponent functors $X \mapsto T(X)^A$ (for each set A), for a functor T in the class.

A (T-)*coalgebra* of a (polynomial) functor $T: \mathbf{Sets} \to \mathbf{Sets}$ consists of two things: a set X together with a function $\alpha: X \to T(X)$. The set X is often called the *state space* or *carrier*, and the function α the *operation* or *transition function*. The important point to realise is that the functor T determines the shape of the operation α. It corresponds to what is called a signature in algebra, see [16]. For example, a coalgebra $\alpha: X \to T(X)$ of the polynomial functor $T(X) = A \times X^B$ consists of two operations $\alpha_1: X \to A$ and $\alpha_2: X \to X^B$. This map α_1 may be called an *attribute*, giving us some information (in terms of values in A) about the states in X. And α_2 can be identified with a procedure operation $X \times B \to X$; it maps a state together with a (parameter) element in B to a next state. These two attribute and procedure operations are most common in our investigations, but more general operations, for example of the form $X \to (A + (C \times X))^B$ can be useful (as in [14]).

If we understand functors as describing signatures of operations, then natural transformations between functors can be understood as maps (or translations) between signatures. Indeed, given a natural transformation $\tau: T \Rightarrow S$ between two functors $T, S: \mathbf{Sets} \rightrightarrows \mathbf{Sets}$, one can transform every coalgebra $\alpha: X \to T(X)$ "of shape T" into a coalgebra $\tau_X \circ \alpha: X \to S(X)$ which is "of shape S". An easy example, arising in the setting of [15], involves functors $T(X) = C \times X^D$ and $S(X) = A \times X^B$. Two functions $f: C \to A$ and $g: B \to D$ (called translation functions in [15]) give rise to a natural transformation $T \Rightarrow S$ with components $f \times \mathrm{id}^g: C \times X^D \to A \times X^B$ given by $(c \in C, \varphi \in X^D) \mapsto (f(c) \in A, \varphi \circ g \in X^B)$.

A *homomorphism* of (T-)coalgebras, say from $\alpha: X \to T(X)$ to $\beta: Y \to T(Y)$ consists of a function $f: X \to Y$ between the underlying state spaces which commutes with the operations: $\beta \circ f = T(f) \circ \alpha$. One calls this coalgebra β on Y *terminal* (or also *final*) if for each set X and coalgebra α on X there is precisely one such homomorphism f. Terminal coalgebras, if they exist[2], are determined up-to-isomorphism. Therefore one often speaks of *the* terminal coalgebra of a functor. A standard result, due to Lambek, tells us that the transition function $\beta: Y \to T(Y)$ of such a terminal coalgebra is an isomorphism. It is a fixed point of the functor T.

Polynomial functors T as defined above gives rise to two adjoint operations on predicates:

$$\begin{tikzcd} \mathcal{P}(X) \arrow[rr, bend left, "(-)^T"] \arrow[rr, phantom, "\top"] & & \mathcal{P}(T(X)) \arrow[ll, bend left, "(-)_T"] \end{tikzcd}$$

The operation $(-)^T$ is written as $\mathrm{Pred}(T)(-)$ in [8, 11, 9], but here we choose to use a shorter notation with superscripts. Its left adjoint operation $(-)_T \dashv (-)^T$ is new. We shall

[1] There are at least two reasonable additional clauses imaginable, allowing (finite) powersets, and fixed points of existing functors (see [7] for the latter); such extensions are not needed for what follows.

[2] The polynomial functors T on $\mathbf{Sets}$ that we use here, all have a terminal coalgebra, which can be obtained via standard techniques [19, 29] as limit of the chain $1 \leftarrow T(1) \leftarrow T^2(1) \leftarrow \cdots$. See [14] for a concrete description of the terminal coalgebras of the kind of functors used below.

define both $(-)^T$ and $(-)_T$ simultaneously, by induction on the structure of the functor T. For predicates $P \subseteq X$ on X and $Q \subseteq T(X)$ on $T(X)$, we define $P^T \subseteq T(X)$ and $Q_T \subseteq X$ as:

$$\begin{array}{rclrcl} P^{\mathrm{id}} & = & P & Q_{\mathrm{id}} & = & Q \\ P^{K_A} & = & A \ (\subseteq A) & Q_{K_A} & = & \emptyset \ (\subseteq X) \\ P^{T_1 \times T_2} & = & \pi^*(P^{T_1}) \cap \pi'^*(P^{T_2}) & Q_{T_1 \times T_2} & = & (\coprod_\pi Q)_{T_1} \cup (\coprod_{\pi'} Q)_{T_2} \\ P^{T_1 + T_2} & = & \prod_\kappa (P^{T_1}) \cap \prod_{\kappa'} (P^{T_2}) & Q_{T_1 + T_2} & = & (\kappa^* Q)_{T_1} \cup (\kappa'^* Q)_{T_2} \\ P^{T^A} & = & \prod_\pi \mathrm{ev}^*(P^T) & Q_{T^A} & = & (\coprod_{\mathrm{ev}} \pi^* Q)_T. \end{array}$$

Some illustrations of these definitions[3,4] will be given below.

3.1. Lemma. *These newly defined predicates satisfy*
(i) $Q \subseteq P^T \Leftrightarrow Q_T \subseteq P$;
(ii) $(f^* P)^T = T(f)^*(P^T)$ *and* $\coprod_f (Q_T) = (\coprod_{T(f)} Q)_T$ *and* $\prod_f (Q_T) = (\prod_{T(f)} Q)_T$.

Proof. (i) By induction on the structure of the functor T, using the adjunctions described earlier. For example in the coproduct case:

$$\begin{array}{ll} Q \subseteq P^{T_1+T_2} = \prod_\kappa (P^{T_1}) \cap \prod_{\kappa'} (P^{T_2}) & \\ \quad \Leftrightarrow \quad Q \subseteq \prod_\kappa (P^{T_1}) \text{ and } Q \subseteq \prod_{\kappa'} (P^{T_2}) & \\ \quad \Leftrightarrow \quad \kappa^* Q \subseteq P^{T_1} \text{ and } \kappa'^* Q \subseteq P^{T_2} & \\ \quad \Leftrightarrow \quad (\kappa^* Q)_{T_1} \subseteq P \text{ and } (\kappa'^* Q)_{T_2} \subseteq P & \text{by induction hypothesis} \\ \quad \Leftrightarrow \quad Q_{T_1+T_2} = (\kappa^* Q)_{T_1} \cup (\kappa'^* Q)_{T_2} \subseteq P. & \end{array}$$

(ii) The first statement is proved by induction on the structure of T. The other two then follows easily by the adjunctions. □

3.2. Definition. Let $\alpha : X \to T(X)$ be a coalgebra of a polynomial functor T and let $P \subseteq X$ be a predicate on its state space.
(i) Put

$$P^+ \stackrel{\mathrm{def}}{=} \alpha^*(P^T) \subseteq X \qquad \text{and} \qquad P^- \stackrel{\mathrm{def}}{=} (\textstyle\coprod_\alpha P)_T \subseteq X.$$

The way to think of P^+ is as the predicate which holds of states whose "direct successors" satisfy P, and of P^- as the predicate of states which are "direct successors" of states satisfying P. Briefly, P^+ is $\mathsf{NextTime}\, P$ (or $\bigcirc P$), and P^- is $\mathsf{LastTime}\, P$ from temporal logic[5] (see *e.g.* [3]). Notice that we leave the dependence on α implicit.
(ii) Call the predicate $P \subseteq X$ an (α-)*invariant* if $P \subseteq P^+$, or equivalently, if $P^- \subseteq P$.

The equivalence mentioned in (ii) holds because $P \subseteq P^+ = \alpha^*(P^T) \Leftrightarrow \coprod_\alpha P \subseteq P^T \Leftrightarrow (\coprod_\alpha P)_T = P^- \subseteq P$. We illustrate these definitions in two examples.

[3] In earlier work [8, 11, 9] the formulation $P^{T_1+T_2} = \coprod_\kappa (P^{T_1}) \cup \coprod_{\kappa'} (P^{T_2})$ was used instead of the one above. These two formulations are equivalent, and the one we have chosen is easier when we consider the left adjoint $(-)_T$ to $(-)^T$, see the proof of the subsequent lemma.

[4] It is easy to extend these definitions to include (finite) powersets, namely by $P^{\mathcal{P}} = \mathcal{P}(P)$ and $Q_{\mathcal{P}} = \bigcup Q$.

[5] One can also use these definitions to introduce correctness specifications (in the style of Floyd and Hoare) for coalgebras: for predicates $P_1, P_2 \subseteq X$ on the state space of a coalgebra $\alpha : X \to T(X)$ define $\{P_1\}\alpha\{P_2\}$ as $\coprod_\alpha P_1 \subseteq P_2^T$. Then P^+ is by construction the weakest precondition $\mathrm{wpr}(\alpha, P)$, and P^- is the strongest postcondition $\mathrm{spo}(\alpha, P)$.

3.3. Examples. (i) Consider the (polynomial) functor $T(X) = A \times X$, with a coalgebra structure $(\mathsf{head}, \mathsf{tail}): X \to A \times X$. For a predicate $P \subseteq X$ we compute both P^+ and P^-:

$$\begin{array}{rcl}
P^+ & = & \{x \in X \mid (\mathsf{head}(x), \mathsf{tail}(x)) \in P^{K_A \times \mathrm{id}}\} \\
& = & \{x \in X \mid \mathsf{head}(x) \in P^{K_A} \ \&\ \mathsf{tail}(x) \in P^{\mathrm{id}}\} \\
& = & \{x \in X \mid \mathsf{head}(x) \in A \ \&\ \mathsf{tail}(x) \in P\} \\
& = & \{x \in X \mid \mathsf{tail}(x) \in P\}. \\
P^- & = & (\coprod_{(\mathsf{head},\mathsf{tail})} P)_{K_A \times \mathrm{id}} \\
& = & (\coprod_{\pi} \coprod_{(\mathsf{head},\mathsf{tail})} P)_{K_A} \cup (\coprod_{\pi'} \coprod_{(\mathsf{head},\mathsf{tail})} P)_{\mathrm{id}} \\
& = & \emptyset \cup \coprod_{\mathsf{tail}} P \\
& = & \{x \in X \mid \exists y \in P\ x = \mathsf{tail}(y)\} \\
& = & \{\mathsf{tail}(y) \mid y \in P\}.
\end{array}$$

The predicate P is thus an invariant if and only if $\mathsf{tail}(x) \in P$ for all $x \in P$.

(ii) We consider a more complicated functor $X \mapsto (A + (C \times X))^B$ with a coalgebra $\alpha: X \to (A + (C \times X))^B$. For a predicate $P \subseteq X$ we compute:

$$\begin{array}{rcl}
P^+ & = & \prod_{\pi} \mathrm{ev}^*(\prod_{\kappa}(A) \cap \prod_{\kappa'}(\pi^*(C) \cap \pi'^* P)) \\
& = & \{x \in X \mid \forall b \in B\ \forall c \in C\ \forall y \in X\ \alpha(x)(b) = \kappa'(c, y) \Rightarrow y \in P\} \\
P^- & = & \coprod_{\pi'} \kappa'^* \coprod_{\mathrm{ev}} \pi^* \coprod_{\alpha} P \\
& = & \{x \in X \mid \exists y \in P\ \exists b \in B\ \exists c \in C\ \alpha(y)(b) = \kappa'(c, x)\}.
\end{array}$$

The power of these definitions $(-)^T$ and $(-)_T$ is that they lead to a uniform notion of invariant, in which the (polynomial) functor T is a parameter. This ensures a wide applicability. Also, the abstract formulations in terms of $\coprod, \prod$ and $(-)^*$ allow us to reason conveniently with the adjointness (or Galois) properties. Moreover, these definitions can be given in an arbitrary (but sufficiently expressive) logical system of first order logic (containing $\exists, \forall$ and equality $=$), using induction on the functor T.

3.4. Fact. It can be shown that a predicate $P \subseteq X$ on the state space of a coalgebra $\alpha: X \to T(X)$ is an invariant (as defined above) if and only if P, as a subset of X, carries a *subcoalgebra* structure. The latter means that there is a (unique) transition function $\alpha_P: P \to T(P)$ making the inclusion function $P \hookrightarrow X$ a homomorphism of coalgebras:

$$\begin{tikzcd}
T(P) \arrow[r, hook] & T(X) \\
P \arrow[u, "\alpha_P"] \arrow[r, hook] & X \arrow[u, "\alpha"']
\end{tikzcd}$$

(Uniqueness follows from the fact that polynomial functors (on **Sets**) preserve injections.)

This subcoalgebra property seems simpler than the property $P \subseteq P^+$ (or equivalently, $P^- \subseteq P$) as used above to define invariants. The earlier definition however has the big advantage that it can be given (by induction) in an arbitrary predicate logic, as used for example by a proof-tool. The subcoalgebra property depends on the fact that predicates on sets can be identified with subsets (and thus are sets themselves). In general, predicates are not identified with subsets.

The following results follow directly from the adjointness properties.

3.5. Lemma. *Invariants are closed under conjunctions $\wedge$ and under universal quantification $\forall$.* □

3.6. Lemma. *The function $P \mapsto P^+$ is monotone and preserves meets $\bigcap_n P_n$ of descending chains ($P_0 \supseteq P_1 \supseteq P_2 \supseteq \cdots$). And the function $P \mapsto P^-$ is also monotone and preserves joins $\bigcup_n P_n$ of ascending chains ($P_0 \subseteq P_1 \subseteq P_2 \subseteq \cdots$).* □

3.7. Proposition. *For a predicate $P \subseteq X$ on the state space of a coalgebra $\alpha: X \to T(X)$ we define a descending chain $(P_n)_{n\in\mathbb{N}}$ of predicates by:*

$$P_0 = P, \qquad P_{n+1} = (P_n)^+ \cap P_n \qquad \text{and} \qquad \underline{P} = \bigcap_n P_n.$$

And similarly, an ascending chain $(P^n)_{n\in\mathbb{N}}$ by:

$$P^0 = P, \qquad P^{n+1} = (P^n)^- \cup P^n \qquad \text{and} \qquad \overline{P} = \bigcup_n P^n.$$

Then: $\underline{P}$ is the greatest invariant contained in P and $\overline{P}$ is the least invariant containing P.

Sometimes we write $\underline{P}_\alpha$ for $\underline{P}$ and $\overline{P}^\alpha$ for $\overline{P}$ if we wish to see the dependence on α explicitly. The predicate $\overline{P}$ captures the states which are reachable from P (via α). In [28] the existence of $\underline{P}$ and $\overline{P}$ (written as $\langle P\rangle$ and $[P]$ there) follows from the observation that invariants (or: subsystems, as used there) form a complete lattice. Here we show how $\underline{P}$ and $\overline{P}$ can be obtained by iteration.

Proof. By construction we have $\underline{P} \subseteq P \subseteq \overline{P}$. The predicate $\underline{P}$ is an invariant since $(\underline{P})^+ = (\bigcap_n P_n)^+ = \bigcap_n (P_n)^+ \supseteq \bigcap_n P_{n+1} = \underline{P}$. And if Q is an invariant with $Q \subseteq P$, then $Q \subseteq P_n$, by induction on n, and so $Q \subseteq \bigcap_n P_n = \underline{P}$. A similar argument shows that $\overline{P}$ is an invariant and that every invariant Q with $P \subseteq Q$ satisfies $\overline{P} \subseteq Q$. □

It is not hard to show that $\underline{P}$ is the greatest fixed point X in $X = P \cap X^+$. Hence we may write $\underline{P}$ as future-tense necessity $\Box P$, like in temporal logic, see [3]. Similarly, $\overline{P}$ is the least fixed point X in $X = P \cup X^-$, so that we can understand $\overline{P}$ as the past-tense eventuality $\Diamond P$. These connections with temporal logic are not fully explored yet.

In the sequel we shall meet the following situation.

3.8. Proposition. *Consider two polynomial functors S, T with a natural transformation $\tau: S \Rightarrow T$ between them. Let $\beta: Y \to S(Y)$ and $\alpha: X \to T(X)$ be two coalgebras with a homomorphism of T-coalgebras $f: Y \to X$ from the translation $(\tau_Y \circ \beta)$ of β to α. Assume further that $Q \subseteq Y$ is a $(\tau_Y \circ \beta)$-invariant and that $P \subseteq X$ is an arbitrary subset. Then: if $f(y) \in P$ for all $y \in Q$, then $f(y) \in \underline{P}_\alpha$ for all $y \in Q$.*

Proof. We may assume $Q \subseteq Q^+$ and $\coprod_f Q \subseteq P$ and have to show that $\coprod_f Q \subseteq \underline{P}_\alpha$. This follows if we can show that the predicate $\coprod_f Q \subseteq X$ is an α-invariant, *i.e.* if it satisfies $\coprod_f Q \subseteq (\coprod_f Q)^+$. Therefore we compute:

$$(\textstyle\coprod_f Q)^+ = (\coprod_\alpha \coprod_f Q)_T = (\coprod_{T(f)} \coprod_{\tau_Y \circ \beta} Q)_T = \coprod_f (\coprod_{\tau_Y \circ \beta} Q)_T = \coprod_f (Q^+) \supseteq \coprod_f (Q).$$

□

We conclude this section with bisimulations with respect to a polynomial functor T. Along the lines described above (and in line with [11, 9]) we define a bisimulation on two coalgebras $\alpha: X \to T(X)$, $\beta: Y \to T(Y)$ of the same functor as a relation $R \subseteq X \times Y$ on their sets of states such that $R \subseteq (\alpha \times \beta)^*(R^T)$, where the relation R^T (also written as

$\mathrm{Rel}(T)(R))$ on $T(X) \times T(Y)$ is defined by induction on the structure of T as:

$$\begin{array}{rcl} R^{\mathrm{id}} & = & R \\ R^{K_A} & = & \mathrm{Eq}(A) = \{(a, a) \mid a \in A\} \\ R^{T_1 \times T_2} & = & (\pi \times \pi)^*(R^{T_1}) \cap (\pi' \times \pi')^*(R^{T_2}) \\ & = & \{(\langle x_1, x_2\rangle, \langle y_1, y_2\rangle) \mid R^{T_1}(x_1, y_1) \ \&\ R^{T_2}(x_2, y_2)\} \\ R^{T_1 + T_2} & = & \prod_{\kappa \times \kappa}(R^{T_1}) \cap \prod_{\kappa' \times \kappa'}(R^{T_2}) \ = \ \coprod_{\kappa \times \kappa}(R^{T_1}) \cup \coprod_{\kappa' \times \kappa'}(R^{T_2}) \\ & = & \{\langle \kappa x_1, \kappa y_1\rangle \mid R^{T_1}(x_1, y_1)\} \cup \{\langle \kappa' x_2, \kappa' y_2\rangle \mid R^{T_2}(x_2, y_2)\} \\ R^{T^A} & = & \prod_{\pi} \langle \mathrm{ev} \circ \pi \times \mathrm{id}, \mathrm{ev} \circ \pi' \times \mathrm{id}\rangle^*(R^T) \\ & = & \{(f_1, f_2) \mid \forall a \in A \ R^T(f_1(a), f_2(a))\}. \end{array}$$

Bisimulations have received a lot of attention in process theory (see *e.g.* [24]), but also in coalgebra (*e.g.* in [1, 27, 28]). The greatest bisimulation on two coalgebras is usually written as $\underline{\leftrightarrow}$. It captures behavioural indistinguishability. A standard result is that elements of the state spaces of two arbitrary T-coalgebras are bisimilar if and only if they are mapped to the same element of the terminal T-coalgebra. If we speak about bisimulation on a single coalgebra $\alpha: X \to T(X)$, then we mean bisimulation as above with $(Y, \beta) = (X, \alpha)$.

An aside: let δ be the diagonal function $\delta(x) = (x, x)$, so that for a predicate P, $\coprod_\delta(P)$ is the relation $\{(x, y) \mid P(x) \wedge x = y\}$. It is not hard to see that $P^T = \delta^*((\coprod_\delta(P))^T)$, where $(-)^T$ on the left hand side applies to a predicate, and $(-)^T$ on the right hand side to a relation. Hence the lifting $(-)^T$ of the functor T to predicates can be obtained from the lifting of T to relations. Also, P is an invariant, if and only if $\coprod_\delta(P)$ is a bisimulation (see *e.g.* [9, 28]). Hence one can argue that the lifting to relations is more fundamental. But our explicit treatment of predicates will be convenient in many situations.

The following technical result will be of importance later; it provides (folklore) connections between invariants and bisimulations.

3.9. Lemma. *Consider two coalgebras $\alpha: X \to T(X)$ and $\beta: Y \to T(Y)$ of the same polynomial functor T, together with a bisimulation $R \subseteq X \times Y$ and an invariant $Q \subseteq Y$ (for β). Then:*

(i) $\pi'^*(Q) \cap R = \{(x, y) \mid Q(y) \ \&\ R(x, y)\} \subseteq X \times Y$ *is a bisimulation;*

(ii) $\coprod_\pi(R) = \{x \mid \exists y \in Y \ R(x, y)\} \subseteq X$ *is an invariant (for α);*

(iii) $\coprod_\pi(\pi'^*(Q) \cap R) = \{x \mid \exists y \in Y \ Q(y) \ \&\ R(x, y)\} \subseteq X$ *is an invariant (for α).*

Proof. The last point follows from the first two. For (i) and (ii) the main steps are to prove $(\pi'^*(Q) \cap R)^T = \pi'^*(Q^T) \cap R^T$ and $(\coprod_\pi R)^T = \coprod_\pi(R^T)$, by induction[6] on the structure of T. The results then follow by calculation. □

Bisimulations of the form $\pi'^*(Q) \cap R$ as in (i) are called "weak" in [21].

4 The terminal model of a coalgebraic specification

In this section we describe terminal models of coalgebraic specifications via invariants, following [11]. In terms of these models we can define what it means for a class to be *underspecified*, *totally specified* or *inconsistent*.

Typical coalgebraic specifications are described in Figures 1, 2 and 3. They consist of three sections, describing the methods (or operations), assertions (or constraints) and

[6] The easiest way is simply to write out the definitions given earlier. Using the adjunctions is not straightforward, since some additional laws of categorical logic, like Beck-Chevalley and Frobenius, have to be used. In (ii), the case where T is an exponent functor requires the Axiom of Choice (which we assume to hold in the logic of sets, and which can be proved in the logic of PVS).

the creation conditions which are required to hold for an initial state new. We write s (for state of self) for an arbitrary inhabitant of the state space X. What is typically coalgebraic about these specifications is that nothing is told about what is inside X (or how it is built up, like in algebra). We are only given some information about the observable behaviour of elements of X. Using the bisimilarity sign $\underline{\leftrightarrow}$ in assertion or creation clauses is justified, since $\underline{\leftrightarrow}$ is determined by the functor associated with the signature of the (earlier given) methods of the specification, see the explanation in (i) below.

class spec: Stack(A)
  methods:
    $\mathsf{push}\colon X \times A \longrightarrow X$
    $\mathsf{pop}\colon X \longrightarrow X$
    $\mathsf{top}\colon X \longrightarrow 1 + A$
  assertions:
    $\mathsf{s.push}(a)\mathsf{.top} = a$
    $\mathsf{s.push}(a)\mathsf{.pop} \mathrel{\underline{\leftrightarrow}} \mathsf{s}$
    $\mathsf{s.top} = * \vdash \mathsf{s.pop} \mathrel{\underline{\leftrightarrow}} \mathsf{s}$
  creation:
    $\mathsf{new.top} = *$
end class spec
	class spec: FreshStack(A)
	  inherts from: Stack(A)
	  methods:
	    $\mathsf{fresh\_push}\colon X \times A \longrightarrow X$
	    $\mathsf{occurs?}\colon X \times A \longrightarrow \{0, 1\}$
	  assertions:
	    $\exists n\ \mathsf{s.pop}^{(n)}\mathsf{.top} = a \vdash \mathsf{s.occurs?}(a) = 1$
	    $\neg\exists n\ \mathsf{s.pop}^{(n)}\mathsf{.top} = a \vdash \mathsf{s.occurs?}(a) = 0$
	    $\mathsf{s.occurs?}(a) = 0 \vdash \mathsf{s.fresh\_push}(a) \mathrel{\underline{\leftrightarrow}} \mathsf{s.push}(a)$
	    $\mathsf{s.occurs?}(a) = 1 \vdash \mathsf{s.fresh\_push}(a) \mathrel{\underline{\leftrightarrow}} \mathsf{s}$
	end class spec

Figure 2: Two stack specification Stack and FreshStack, where the latter has an extra method $\mathsf{fresh\_push}$ which only pushes fresh (not already occurring) elements on the stack

A model of such a class specification consists basically of two parts: a set S serving as state space, together with suitable functions acting on S which interpret the methods; this interpretation should be such that (a) the assertions are satisfied, and (b) the subset of S consisting of those states for which the creation conditions hold is non-empty. Below we shall construct a particular model for a class specification, namely the terminal one. This construction involves three steps, which we shall illustrate for the stack specification in Figure 2.

(i) A functor T is extracted from the specification, which captures the signature of the methods. This functor is given by the Cartesian product of the output types of the methods, where the output type of an operation $X \times B \longrightarrow S(X)$ is defined as the exponent type $S(X)^B$. For example the functor associated with the stack methods is $T(X) = X^A \times X \times (1 + A)$. (The first two factors X^A and X resulting from push and pop can be combined into a single factor $X^{(A+1)}$, since $X^A \times X \cong X^{(A+1)}$, leading to further simplication.)

Coalgebras $X \to T(X)$ are models of the methods in the specification. Among these models the terminal coalgebra $Z \overset{\cong}{\to} T(Z)$ will play a special rôle. In the case of the stack specification, this terminal coalgebra can be identified with the set $Z = (1 + A)^{(A+1)^\star}$ of functions from $(A + 1)^\star$ to $1 + A$, with methods acting on $\varphi \in Z$ as:

$$\begin{aligned}
\mathsf{push}(\varphi, a) &= \lambda\sigma \in (A + 1)^\star\ \varphi(\sigma \cdot (\kappa a)) \\
\mathsf{pop}(\varphi) &= \lambda\sigma \in (A + 1)^\star\ \varphi(\sigma \cdot (\kappa' *)) \\
\mathsf{top}(\varphi) &= \varphi(\langle\rangle).
\end{aligned}$$

(ii) Once we have interpretations of the methods of a coalgebraic specification in some coalgebra $X \to T(X)$, we can consider the assertions. They give rise to a subset[7] $E \subseteq$

[7] We get a subset (instead of a relation, as in algebra), because the assertions are required to contain only one variable ranging over states (which is typically written as s), see [11].

X describing the states which statisfy the assertions. We can then simply say that the assertions hold in this coalgebra if this subset $E \subseteq X$ is all of X.

In the stack example we get the following subset E of the terminal coalgebra[8].

$$E = \{\varphi \in Z \mid (\forall a \in A\ \varphi(\langle \kappa a \rangle) = \kappa' a \ \&\ \lambda\sigma\ \varphi(\sigma \cdot (\kappa a) \cdot (\kappa' *)) = \varphi) \\ \&\ \varphi(\langle\rangle) = \kappa * \Rightarrow \lambda\sigma\ \varphi(\sigma \cdot (\kappa' *)) = \varphi\}.$$

This subset E does not carry a coalgebra structure, but in the previous section we have seen how it can be provided with such structure, by taking $\underline{E}$ or $\overline{E}$. The set $\underline{E}$ is what we are interested in: it is the terminal model which satisfies the assertions (since $\underline{E} \subseteq E$).

We describe the set $\underline{E}$ explicitly for the stack example with E as above.

$$\underline{E} = \{\varphi \in Z \mid \forall a \in A\, \forall \sigma, \sigma' \in (A+1)^\star\ \varphi(\sigma \cdot (\kappa a)) = \kappa' a \\ \&\ \varphi(\sigma \cdot (\kappa a) \cdot (\kappa' *) \cdot \sigma') = \varphi(\sigma \cdot \sigma') \ \&\ \varphi(\sigma) = \kappa * \Rightarrow \varphi(\sigma \cdot (\kappa' *) \cdot \sigma') = \varphi(\sigma \cdot \sigma')\}$$

On close inspection we see that functions $\varphi \in \underline{E}$ are completely determined by their values $\varphi((\kappa' *)^n)$, where $(\kappa' *)^n$ is the sequence $(\kappa' *) \cdots (\kappa' *)$ of length n. Furthermore, these values $\varphi((\kappa' *)^n)$ are such that if $\varphi((\kappa' *)^n) = \kappa *$, then $\varphi((\kappa' *)^{n+1}) = \kappa *$. Hence such a $\varphi \in \underline{E}$ may be identified with a finite or infinite sequence of elements from A. Thus $\underline{E} \cong A^\infty$, where $A^\infty = A^\star + A^{\mathbb{N}}$ is the set of both finite and infinite sequences of elements of A. The induced methods on $\sigma \in A^\infty$ are

$$\sigma.\mathsf{push}(a) = a \cdot \sigma \qquad \sigma.\mathsf{pop} = \begin{cases} \langle\rangle & \text{if } \sigma = \langle\rangle \\ \sigma' & \text{if } \sigma = a \cdot \sigma' \end{cases} \qquad \sigma.\mathsf{top} = \begin{cases} \kappa * & \text{if } \sigma = \langle\rangle \\ \kappa' a & \text{if } \sigma = a \cdot \sigma' \end{cases}$$

It is not hard to verify that the stack assertions hold in this coalgebra. By construction, A^∞ (with these operations) is the terminal coalgebra satisfying the assertions, see [11]. It is the *minimal realisation* (in automata theoretic terminology) of the specified behaviour[9].

(iii) In a final step, the creation conditions in the specification determine a subset $I \subseteq \underline{E}$ of "initial" states that can be used as interpretations of new. In the stack example,

$$I = \{\sigma \in A^\infty \mid \sigma.\mathsf{top} = \kappa *\} = \{\langle\rangle\}.$$

so that the interpretation of new is determined as the empty sequence. This corresponds to our intuition.

Recapitulating, we obtain the terminal model of a coalgebraic specification by successively carving out appropriate subsets $I \subseteq \underline{E} \subseteq E \subseteq Z$ of the terminal coalgebra $Z \overset{\cong}{\Rightarrow} T(Z)$ of the functor T capturing the signature of methods.

The set $I \subseteq A^\infty$ of initial states in the terminal coalgebra A^∞ satisfying the assertions is a singleton. This need not be the case in general, and this leads us to the following classification of coalgebraic specifications.

4.1. Definition. Consider an arbitrary coalgebraic specification $\mathcal{A}$, for which we determine the terminal model as $I_{\mathcal{A}} \subseteq \underline{E}_{\mathcal{A}}$ as above, with set of initial states $I_{\mathcal{A}}$. We call $\mathcal{A}$

(i) *inconsistent* if $I_{\mathcal{A}}$ is the empty set;
(ii) *totally specified* if $I_{\mathcal{A}}$ is a singleton;
(iii) *underspecified* if $I_{\mathcal{A}}$ is neither empty nor a singleton.

[8] On the terminal coalgebra, the bisimilarity relation $\underline{\leftrightarrow}$ is simply equality; on an arbitrary coalgebra we have to use the bisimilarity relation on that coalgebra as interpretation of $\underline{\leftrightarrow}$ in the specification.

[9] The occurrence of the set $A^\star$ of finite sequences in the minimal Stack realisation is probably clear. We also get the infinite sequences $A^{\mathbb{N}}$ in this model since nothing in the specification tells us that the sequence of observations s.top, s.pop.top, s.pop.pop.top, ... must eventually lead to an element $*$ (indicating that the stack s is empty).

Hence, an inconsistency means that there is no (initial) state displaying the required behaviour; total specification means that there is precisely one such state, and underspecification means that there may be many such states[10]. The stacks in Figure 2 are totally specified, as we have seen. (The 'fresh' stacks are also totally specified, since they have the same models as stacks: the additional methods fresh_push and occurs? are defined in terms of the stack methods.) The store specification in Figure 3 gives an example of underspecification. Intuitively this may be clear, since it is not layed down after how many delete_one invocations a stored element is actually deleted.

class spec: Store(A)
  methods:
    $\mathsf{occurs?}\colon X \times A \longrightarrow \{0,1\}$
    $\mathsf{put}\colon X \times A \longrightarrow X$
    $\mathsf{delete\_one}\colon X \longrightarrow X$
  assertions:
    $\mathsf{s.occurs?}(a) = 1 \vdash$
      $\exists n\ \mathsf{s.delete\_one}^{(n)}.\mathsf{occurs?}(a) = 0$
    $a_1 = a_2 \vdash \mathsf{s.put}(a_1).\mathsf{occurs?}(a_2) = 1$
    $a_1 \neq a_2 \vdash \mathsf{s.put}(a_1).\mathsf{occurs?}(a_2) = \mathsf{s.occurs?}(a_2)$
    $\exists a\ \mathsf{s.occurs?}(a) = 1 \vdash$
      $\exists a\ (\mathsf{s.occurs?}(a) = 1 \wedge \mathsf{s.delete\_one.occurs?}(a) = 0$
      $\wedge\, \forall b \neq a\ \mathsf{s.delete\_one.occurs?}(b) = \mathsf{s.occurs?}(b))$
    $\neg\exists a\ \mathsf{s.occurs?}(a) = 1 \vdash$
      $\mathsf{s.delete\_one.occurs?}(b) = 0$
  creation:
    $\mathsf{new.occurs?}(a) = 0$
end class spec

Figure 3: A specification of a class in which one can store elements from A, which reappear at some unspecified later stage

5 Refinements with invariant and bisimulation proofs

A refinement establishes a relation between two specifications. Usually one says that an *abstract* specification is refined by a *concrete* one, see [10, 23, 4] (and also Section 2). This means that the behaviour that is specified in the abstract specification can be realised by the concrete specification. The latter often contains more details about how to realise this behaviour. Typically, it is more deterministic (*i.e.* less underspecified) than the abstract specification (as stressed in [6]), see the stack specification refining the store specification later in this section. We start with the definition of refinement, and with two associated proof principles, involving invariants and bisimulations.

5.1. Definition. Consider two coalgebraic specifications: $\mathcal{A}$ for abstract, and $\mathcal{C}$ for concrete, with associated (polynomial) "signature" functors $T_{\mathcal{A}}$ and $T_{\mathcal{C}}$. We say that $\mathcal{C}$ *refines* $\mathcal{A}$ (or equivalently, that $\mathcal{A}$ is *refined by* $\mathcal{C}$) if

(i) there is a translation of methods from the abstract to the concrete specification in the form of a natural transformation $\tau\colon T_{\mathcal{C}} \Rightarrow T_{\mathcal{A}}$;

[10]The initial state new occurring in the specification may be understood in this case as an arbitrary element of this set.

(ii) for each model $M_{\mathcal{C}}$ of the concrete specification $\mathcal{C}$, there is a reachable (in $\mathcal{C}$) state r, such that

(a) r satisfies the creation conditions from $\mathcal{A}$ (when they are suitably translated via τ);

(b) the predicate $RS = \overline{\{r\}}^{\mathcal{A}}$ of reachable states from r (with respect to the translated methods from $\mathcal{A}$) implies the assertions from $\mathcal{A}$ (translated via τ). This requirement makes sense, because RS, considered as subset of the model $M_{\mathcal{C}}$, carries an $\mathcal{A}$-coalgebra structure by Fact 3.4 (using τ again).

The translation τ allows us to re-interpret the abstract assertions and creation conditions in the induced $T_{\mathcal{A}}$-structure on the concrete state space. The intuition behind the definition is the following: in order to realise the behaviour of the abstract class specification $\mathcal{A}$ we need first of all a translation τ of the methods of $\mathcal{A}$ to the methods of some other class specification $\mathcal{C}$. Then, in each model of $\mathcal{C}$, there should be a reachable state r which can play the rôle of the initial state new in $\mathcal{A}$, and all the states which are reachable from r via the translated methods from $\mathcal{A}$ should satisfy the assertions from $\mathcal{A}$. Hence on this subset RS of reachable states from r we realise the behaviour of $\mathcal{A}$. This suffices since by working in $\mathcal{C}$ (via τ) we will never get beyond the reachable states. In brief: $\mathcal{C}$ refines $\mathcal{A}$ if every model of $\mathcal{C}$ contains some reachable part which is a model of $\mathcal{A}$ (after suitable translation).

5.2. Theorem. *Consider abstract and concrete class specifications $\mathcal{A}$ and $\mathcal{C}$ as in the previous definition, with translation $\tau : T_{\mathcal{C}} \Rightarrow T_{\mathcal{A}}$. Then: $\mathcal{C}$ refines $\mathcal{A}$ if and only if for each model $M_{\mathcal{C}}$ of $\mathcal{C}$ there is a reachable state r satisfying (a) above, and a predicate P on $M_{\mathcal{C}}$ which satisfies:*

(i) *P is an $\mathcal{A}$-invariant,* i.e. *is closed under the translated operations of $\mathcal{A}$;*

(ii) *P contains r;*

(iii) *the (translated) assertions from $\mathcal{A}$ hold on P.*

Proof. The (only-if) part is easy, since by definition $\overline{\{\mathsf{r}\}}^{\mathcal{A}}$ satisfies the conditions in the theorem. Conversely, if P is predicate as described in (i)–(iii), then $\overline{\{\mathsf{r}\}}^{\mathcal{A}}$ is contained in P, by definition of $\overline{(-)}$. Hence all the assertions of $\mathcal{A}$ also hold on $\overline{\{\mathsf{r}\}}^{\mathcal{A}}$. □

The advantage of reasoning with invariants is that they reduce global arguments to local ones which only involve reasoning with single steps.

There is an alternative proof method involving bisimulations in case the abstract class $\mathcal{A}$ is totally specified (see Definition 4.1). This proof technique is used in [15].

5.3. Theorem. *Consider again the abstract and concrete class specifications $\mathcal{A}$ and $\mathcal{C}$ with $\tau : T_{\mathcal{C}} \Rightarrow T_{\mathcal{A}}$ as in the previous theorem, but this time assume that the abstract class $\mathcal{A}$ is totally specified. Then: $\mathcal{C}$ refines $\mathcal{A}$ if and only if for all models $M_{\mathcal{C}}$ of $\mathcal{C}$ and $M_{\mathcal{A}}$ of $\mathcal{A}$ there is a reachable state r in $M_{\mathcal{C}}$ (satisfying (a) in Definition 5.1) and a bisimulation relation R on $M_{\mathcal{C}} \times M_{\mathcal{A}}$ with $R(\mathsf{r}, \mathsf{new})$.*

The bisimulation R in this situation is associated with the functor $T_{\mathcal{A}}$ describing the abstract signature. The states of the concrete specification carry a $T_{\mathcal{A}}$-coalgebra structure via the translation τ.

Proof. First assume that $\mathcal{C}$ refines $\mathcal{A}$, and that models $M_{\mathcal{C}}$ and $M_{\mathcal{A}}$ are given, with a reachable state r in $M_{\mathcal{C}}$. We may assume an invariant P on $M_{\mathcal{C}}$ as in Theorem 5.2. Then $R = \{(\mathsf{s}, \mathsf{t}) \in M_{\mathcal{C}} \times M_{\mathcal{A}} \mid P(\mathsf{s}) \;\&\; \mathsf{s} \underline{\leftrightarrow} \mathsf{t}\}$ is our required relation. It is a bisimulation by Lemma 3.9 (i), and it satisfies $R(\mathsf{r}, \mathsf{new})$ because $P(\mathsf{r})$ and because $\mathcal{A}$ is totally specified: both r and new are mapped to the same element of the terminal $\mathcal{A}$-model, so that $\mathsf{r} \underline{\leftrightarrow} \mathsf{new}$.

In the reverse direction, assume we have an arbitrary model $M_{\mathcal{C}}$ of $\mathcal{C}$ with an appropriate reachable state r. Consider the terminal mode $M_{\mathcal{A}}$ of $\mathcal{A}$. By assumption, there is a bisimulation R on $M_{\mathcal{C}} \times M_{\mathcal{A}}$ with $R(\mathsf{r}, \mathsf{new})$. Now we can define a predicate P on $M_{\mathcal{C}}$ as $P(\mathsf{s}) \Leftrightarrow \exists \mathsf{t} \in \overline{\{\mathsf{new}\}}\ R(\mathsf{s}, \mathsf{t})$. It is an invariant by Lemma 3.9 (iii), and clearly satisfies $P(\mathsf{r})$. Further, all the assertions from $\mathcal{A}$ hold on P by the following argument. If $P(\mathsf{s})$, say with $R(\mathsf{s}, \mathsf{t})$ for $\mathsf{t} \in \overline{\{\mathsf{new}\}}$, then $\mathsf{s} \leftrightarrow \mathsf{t}$ because R is a bisimulation. Since the assertions hold for t, they also hold for s. □

We conclude with an illustration of these notions and techniques in an example, involving a refinement of the Store specification (in Figure 3) by the FreshStack specification (in Figure 2). This refinement will be proved via an invariant. Examples of bisimulation refinement proofs occur in [15].

A store object, as characterised by the class specification Store, can store elements from some data set A. The $\mathsf{occurs?}$ method (or attribute) with interface $X \times A \longrightarrow \{0, 1\}$ tells us of a particular element $a \in A$ if it is stored in a state (or object) s, namely if $\mathsf{s.occurs?}(a) = 1$. The procedure $\mathsf{put}\colon X \times A \longrightarrow X$ allows us to put elements from A into a store, and the procedure $\mathsf{delete\_one}\colon X \longrightarrow X$ will delete an unspecified element from the store, provided the store is not empty. The first assertion of the specification tells us that if an element is stored, then it will be deleted after a finite number of invocations of the delete_one method. Notice that, as a result, only denumerably many elements from A can be stored in a single state.

There is an obvious way to realise the behaviour required for stores via stacks. Putting an element into a store-as-stack means pushing it onto the stack, where some care is needed to ensure that only elements which are not already on the stack are actually pushed; others are discarded. The fresh_push procedure in the FreshStack specification does precisely this. Deleting an element from a store-as-stack is simply popping an element off the stack. Notice that we hereby make a specific choice which element is deleted, namely the one which was last put on the stack. This illustrates the point that refinement reduces implementation freedom. But the store specification allows us to make such decisions. The invariant that has to be maintained for such stores-as-stacks is that there are no multiply occuring elements on the stack.

In the previous paragraph we have informally described a translation from the signature of Store to the signature of FreshStack. According to Definition 5.1, we formally have to define a natural transformation from the functor T_{FS} describing the signature of the FreshStack methods to the functor T_{St} capturing the signature of the Store methods. These (polynomial) functors are given by

$$T_{\mathrm{FS}}(X) = \{0, 1\}^A \times X^A \times X^A \times X \times (1 + A) \quad \text{and} \quad T_{\mathrm{St}}(X) = \{0, 1\}^A \times X^A \times X$$

where the first X^A component in $T_{\mathrm{FS}}(X)$ is the output type of fresh_push, and the second component X^A comes from push. The required natural transformation $\tau\colon T_{\mathrm{FS}} \Rightarrow T_{\mathrm{St}}$ is $\tau(f, g, h, x, y) = (f, g, x)$. It formally captures the translation of methods from Store to FreshStack, so that every T_{FS}-coalgebra gets translated into a T_{St}-coalgebra (via τ) in which delete_one is pop, put is fresh_push and occurs? is occurs?. Usually it is clear enough to describe such a translation at an informal level.

5.4. Proposition. *The Store specification in Figure 3 is refined by the FreshStack specification in Figure 2 via the invariant P (on each model of FreshStack) given by*

$$P(\mathsf{s}) \stackrel{\text{def}}{\Leftrightarrow} \forall n \in \mathbb{N}\ \mathsf{s.pop}^{(n)}.\mathsf{top} \neq * \Rightarrow \forall m > 0\ \mathsf{s.pop}^{(n)}.\mathsf{top} \neq \mathsf{s.pop}^{(n+m)}.\mathsf{top}$$

and via the initial stack new *as representation of the initial store.*

Proof. Essentially this is straightforward reasoning. First, one has to show that $P(\mathsf{new})$ holds, and that P is an invariant: $P(\mathsf{s})$ implies both $P(\mathsf{s.pop})$ and $P(\mathsf{s.fresh\_push}(a))$ (for all $a \in A$). Then that the Store assertions follow from P. We shall do the first one. Assume therefore that $P(\mathsf{s})$ and that $\mathsf{s.occurs?}(a) = 1$. We have to show that for some $n \in \mathbb{N}$, $\mathsf{s.pop}^{(n)}.\mathsf{occurs?}(a) = 0$. The assumption $\mathsf{s.occurs?}(a) = 1$ means that we can pick an n with $\mathsf{s.pop}^{(n)}.\mathsf{top} = a$. We claim that $n + 1$ does what we want. If not, *i.e.* if $\mathsf{s.pop}^{(n+1)}.\mathsf{occurs?}(a) = 1$, say via m with $a = \mathsf{s.pop}^{(n+1)}.\mathsf{pop}^{(m)}.\mathsf{top} = \mathsf{s.pop}^{(n+1+m)}.\mathsf{top}$, then we have a contradiction with $P(\mathsf{s})$. □

6 Concluding remarks

We have developed a general theory of refinement in a coalgebraic setting, with associated proof principles based on invariants and bisimulations. This theory aims at proving the correctness of refinements between classes in object-oriented languages, using modern proof tools. We expect that this theory scales up smoothly to the hybrid setting of [12], since the underlying concepts are of the same (coalgebraic) nature.

Current research (in collaboration with Hensel, Huisman and Tews) focusses on the development of a (front-end) tool for reasoning about coalgebraic class specifications. The idea is that the tool should automatically translate such specifications into appropriate PVS theories, giving us a setting for PVS-assisted reasoning about the specifications. Proving refinements via invariants (as described here) forms an essential ingredient.

Acknowlegments

Thanks are due to Andrea Corradini, Joseph Goguen, Ulrich Hensel, Marieke Huisman, Jan Rutten and Frits Vaandrager for helpful comments and discussions.

References

1. P. Aczel and N. Mendler. A final coalgebra theorem. In D.H. Pitt, A. Poigné, and D.E. Rydeheard, editors, *Category Theory and Computer Science*, number 389 in Lect. Notes Comp. Sci., pages 357–365. Springer, Berlin, 1989.
2. M.A. Arbib and E.G. Manes. Parametrized data types do not need highly constrained parameters. *Inf. & Contr.*, 52:139–158, 1982.
3. E.A. Emerson. Temporal and modal logic. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B, pages 997–998. Elsevier/MIT Press, 1990.
4. J.A. Goguen. An algebraic approach to refinement. In D. Bjørner, C.A.R. Hoare, and H. Langmaack, editors, *VDM '90. VDM and Z—Formal Methods in Software Development*, number 428 in Lect. Notes Comp. Sci., pages 12–28. Springer, Berlin, 1990.
5. J.A. Goguen and G. Malcolm. Proof of correctness of object representations. In A.W. Roscoe, editor, *A Classical Mind. Essays in honour of C.A.R. Hoare*, pages 119–142. Prentice Hall, 1994.
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. U. Hensel and B. Jacobs. Proof principles for datatypes with iterated recursion. In *Category Theory and Computer Science*, Lect. Notes Comp. Sci., Springer, Berlin, 1997.
8. C. Hermida and B. Jacobs. An algebraic view of structural induction. In L. Pacholski and J. Tiuryn, editors, *Computer Science Logic 1994*, number 933 in Lect. Notes Comp. Sci., pages 412–426. Springer, Berlin, 1995.
9. C. Hermida and B. Jacobs. Structural induction and coinduction in a fibrational setting. Full version of [8], 1996.
10. C.A.R. Hoare. Proof of correctness of data representations. *Acta Informatica*, 1:271–281, 1972.

11. B. Jacobs. Mongruences and cofree coalgebras. In V.S. Alagar and M. Nivat, editors, *Algebraic Methods and Software Technology*, number 936 in Lect. Notes Comp. Sci., pages 245–260. Springer, Berlin, 1995.
12. B. Jacobs. Coalgebraic specifications and models of deterministic hybrid systems. In M. Wirsing and M. Nivat, editors, *Algebraic Methods and Software Technology*, number 1101 in Lect. Notes Comp. Sci., pages 520–535. Springer, Berlin, 1996.
13. 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.
14. 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.
15. 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.
16. B. Jacobs and J. Rutten. A tutorial on (co)algebras and (co)induction. *EATCS Bulletin*, 62:222–259, 1997.
17. S. Kamin. Final data types and their specification. *ACM Trans. on Progr. Lang. and Systems*, 5(1):97–123, 1983.
18. L. Lamport. Verification and specification of concurrent programs. In J.W. de Bakker, W.P. de Roever, and G. Rozenberg, editors, *A Decade of Concurrency*, number 803 in Lect. Notes Comp. Sci., pages 347–374. Springer, Berlin, 1994.
19. D.J. Lehmann and M.B. Smyth. Algebraic specification of data types: A synthetic approach. *Math. Systems Theory*, 14:97–139, 1981.
20. B. Liskov and J. Guttag. *Abstraction and Specification in Program Development.* The MIT Press, Cambridge, MA, 1986.
21. N. Lynch and F. Vaandrager. Forward and backward simulations. I. Untimed systems. *Inf. & Comp.*, 121(2):214–233, 1995.
22. E.G. Manes and M.A. Arbib. *Algebraic Appoaches to Program Semantics.* Texts and Monogr. in Comp. Sci.,. Springer, Berlin, 1986.
23. R. Milner. An algebraic definition of simulation between programs. In *Sec. Int. Joint Conf. on Artificial Intelligence*, pages 481–489. British Comp. Soc. Press, London, 1971.
24. R. Milner. *A Calculus of Communicating Systems.* Lect. Notes Comp. Sci. Springer, Berlin, 1989.
25. 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.
26. H. Reichel. An approach to object semantics based on terminal co-algebras. *Math. Struct. in Comp. Sci.*, 5:129–152, 1995.
27. J. Rutten and D. Turi. On the foundations of final semantics: non-standard sets, metric spaces and partial orders. In J.W. de Bakker, W.P. de Roever, and G. Rozenberg, editors, *Semantics: Foundations and Applications*, number 666 in Lect. Notes Comp. Sci., pages 477–530. Springer, Berlin, 1993.
28. J.J.M.M. Rutten. Universal coalgebra: a theory of systems. CWI Report CS-R9652, 1996.
29. M.B. Smyth and G.D. Plotkin. The category theoretic solution of recursive domain equations. *SIAM Journ. Comput.*, 11:761–783, 1982.
30. M. Wand. Final algebra semantics and data type extension. *Journ. Comp. Syst. Sci*, 19:27–44, 1979.