Exercises in Coalgebraic Specification *

Bart Jacobs

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

18 January 2000

Abstract. An introduction to coalgebraic specification is presented via examples. A coalgebraic specification describes a collection of coalgebras satisfying certain assertions. It is thus an axiomatic description of a particular class of mathematical structures. Such specifications are especially suitable for state-based dynamical systems in general, and for classes in object-oriented programming languages in particular. This paper will gradually introduce the notions of bisimilarity, invariance, component classes, temporal logic and refinement in a coalgebraic setting. Besides the running example of the coalgebraic specification of (possibly infinite) binary trees, a specification of Peterson's mutual exclusion algorithm is elaborated in detail.

Contents

* Draft version, intended for the proceedings of the *Mathematics for Information Technology* spring school, Oxford, April 2000. Comments are welcome.

1 Introduction

This paper presents an introduction to the relatively young area of coalgebraic specification, developed in [33, 10, 12, 11, 13, 14, 6, 7, 3]. It is aimed at a mathematically oriented audience, and therefore it focuses on coalgebraic specifications as axiomatic descriptions of certain mathematical structures, and on how to formulate and prove properties about such structures. The emphasis lies on concrete examples, and not on the meta-theory of coalgebras. Currently, coalgebraic specifications are being used and developed in theoretical computer science, in particular to specify classes in object-oriented languages (such as Java [20]). Much of the motivation, terminology, and many of the examples stem from this area. We shall not emphasise this aspect, and no prior experience in computer science is assumed. It is possible that in the future, coalgebras will find comparable applications in mathematics, for instance in system and control theory. Further, the theory of coalgebras is best formulated and developed using categorical notions and techniques. In this introduction we shall not use any category theory, however, and describe coalgebras at an elementary level. This means that we cover neither homomorphisms of coalgebras, nor the related topic of terminal (also called final) coalgebras.

Coalgebras are simple mathematical structures that can be understood as duals of algebras. We refer to [18] for an introduction to the study of coalgebras. Here we shall simply use coalgebras, without concentrating too much on the difference with algebras, and we hope that readers will come to appreciate their rôle in specification and verification. They are typically used to describe state-based dynamical systems, where the state space (set of states) of the system is considered as a black box, and where nothing is known about the way that the observable behaviour is realised. Coalgebraic specification is thus important for the study of such dynamical systems. In this field one naturally reasons in terms of invariance and bisimilarity. Indeed, these notions are fundamental in the theory of coalgebras[1]. Further, a recent development is the close connection between coalgebras and temporal logic, see [27, 15]. The temporal operators $\Box$ for henceforth and $\Diamond$ for eventually can be defined easily in a coalgebraic setting, in terms of invariants. Their use is quite natural in a state-based setting, namely for reasoning about all/some future states in safety/progress formulas. In this paper we illustrate the use of $\Box$ and $\Diamond$ in coalgebraic specification. We also give an impression of the application of the least and greatest fixed point operators from the μ-calculus [36] in this setting.

As mentioned, coalgebraic specifications give an axiomatic description of mathematical structures. As such they resemble axiomatic descriptions of groups or rings that are common in mathematics. But there are also some differences with standard mathematical approaches, stemming from their use in computer science.

[1] Invariants and bisimilations (or congruences) are also relevant for algebras, see for example [30, 32, 8], and the end of Section 6.

1. Coalgebraic specifications are typically structured, using the object-oriented mechanisms of inheritance (with subclasses inheriting from superclasses) and aggregation (with ambient classes having component classes). This is needed because specifications in computer science tend to become very big, in order to capture all possible scenarios in which a system has to function. Such explicit structuring mechanisms do not exist in axiomatisations in mathematics. Lamport [23, Section 1] writes: "Although mathematicians have developed the science of writing formulas, they haven't turned that science into an engineering discipline. They have developed notations for mathematics in the small, but not for mathematics in the large.".
2. Coalgebraic specifications have a rather precise format. Actually, there is an experimental formal language CCSL, for Coalgebraic Class Specification Language. Specifications in CCSL are thus objects which can be manipulated by a computer. There is a LOOP compiler [7] which translates CCSL specifications into logical theories for a proof tool. The latter is a special program that helps in formal reasoning (*e.g.* via built-in tactics and decision procedures). It is like a calculator for reasoning. Such proof tools are useful for proving the correctness of elaborate system descriptions (*e.g.* as complex coalgebraic specifications), involving many different case distinctions. Humans easily make mistakes, by omitting a case or a precondition, but proof tools are very good at such bureaucratism. The specifications in this paper are presented in a semi-formal style, resembling CCSL. All the results about the examples have been proved with the proof tool PVS [28], using the automatic translation of CCSL specifications into the logic of PVS with (a recent version of) the compiler from [7]. However, here we shall present the proofs of these results in usual mathematical style.

This introductory paper is organised as follows. After some explanations about the notation that will be used, an introduction to formal specification is given in Section 3; it contains a slightly unusual specification of vector spaces, meant as preparation to coalgebraic specification. Section 4 introduces the running example of (possibly infinite) binary trees. It is followed by two sections on bisimulations and on invariants. Section 7 introduces temporal logic in a coalgebraic setting, and applies it to binary trees. Subsequently, also the ingredients of a coalgebraic μ-calculus are introduced, and used to describe (in)finiteness for binary trees. Section 9 then introduces the example of Peterson's mutual exclusion algorithm. It is specified in a structured manner, and its correctness is proved. The final section concentrates on refinement in a coalgebraic setting.

2 Mathematical preliminaries

We will use some standard constructions on sets. For example, the Cartesian product $X \times Y = \{(x, y) \mid x \in X \text{ and } y \in Y\}$, with projection functions $\pi: X \times Y \to X$ and $\pi': X \times Y \to Y$ given by $\pi(x, y) = x$ and $\pi'(x, y) = y$. We shall frequently make use of the fact that functions $Z \to X \times Y$ correspond to pairs of functions $Z \to X$, $Z \to Y$.

The dual of the product is the coproduct (also called sum or disjoint union): $X + Y = \{(x, 0) \mid x \in X\} \cup \{(y, 1) \mid y \in Y\}$, with coprojection functions $\kappa: X \to X + Y$ and $\kappa': Y \to X + Y$, given by $\kappa(x) = (x, 0)$ and $\kappa'(y) = (y, 1)$. Sometimes we write $\kappa x = \kappa(x)$ and $\kappa' y = \kappa'(y)$ without brackets $(-)$, like for projections π, π'. Notice that these coprojections are injective functions, and are "disjoint", in the sense that $\kappa(x) \neq \kappa'(y)$, for all $x \in X, y \in Y$. We shall also often use that functions $X + Y \to Z$ correspond to pairs of functions $X \to Z$, $Y \to Z$. For this we use some special notation: given $f: X \to Z$ and $g: Y \to Z$, there is a corresponding function $X + Y \to Z$, which we shall write as:

$$
\begin{array}{rl}
a \longmapsto & \mathsf{CASES}\ a\ \mathsf{OF} \\
& \quad \kappa x \mapsto f(x) \\
& \quad \kappa' y \mapsto g(y) \\
& \mathsf{ENDCASES}
\end{array}
$$

This function checks for an element $a \in X + Y$ if it is of the form $\kappa(x) = (x, 0)$, or $\kappa'(y) = (y, 1)$. In the first case it applies the function f to x, and in the second case g to y. We shall apply such CASES functions also when f, g are predicates, since these fit in for $Z = \{0, 1\}$. Thus, for example, the set

$$
\begin{array}{rl}
\{(x, y) \in \mathbb{N} \times (1 + \mathbb{N}) \mid x > 0 \Rightarrow & \mathsf{CASES}\ y\ \mathsf{OF} \\
& \quad \kappa u \mapsto x = 1 \\
& \quad \kappa' v \mapsto x = y \\
& \mathsf{ENDCASES}\, \}
\end{array}
$$

contains all pairs of the form $(0, \kappa *)$, $(1, \kappa *)$, $(0, \kappa' n)$, $(m, \kappa' m)$, for $n, m \in \mathbb{N}$ with $m > 0$. In general, coproducts are not so frequently used as products, but they play an important rôle for coalgebras, because they occur in many examples.

The empty set will usually be written as 0. Then $X + 0 \cong X$. We write 1 for an arbitrary singleton set, say $1 = \{*\}$. Note that $X \times 1 \cong X$. We recall that $X \times Y \cong Y \times X$, $X \times (Y \times Z) \cong (X \times Y) \times Z$, and also that $X + Y \cong Y + X$, $X + (Y + Z) \cong (X + Y) + Z$.

We shall further write Y^X for the set of (total) functions from X to Y. There is a one-to-one correspondence between functions $Z \to Y^X$ and functions $Z \times X \to Y$. A function $x \mapsto \cdots$ will sometimes be written in lambda notation: $\lambda x.\ \cdots$.

Finally, $X^\star$ will be the set of finite sequences $\alpha = \langle x_1, \ldots, x_n \rangle$ of elements $x_i \in X$. We shall write $|-|$ for the length, so that $|\alpha| = n$. The empty sequence is $\langle\rangle$, and $x_0 \cdot \alpha$ is $\langle x_0, x_1, \ldots, x_n \rangle$. The set $X^\star$ forms the free monoid on X.

What we shall call a *polynomial functor* is a certain mapping[2] from sets to sets, built from primitive operations. The collection of polynomial functors that we consider in this paper is the smallest collection satisfying:

1. the identity mapping $X \mapsto X$ is polynomial, and for each set A, the constant mapping $X \mapsto A$ is polynomial;

[2] These mappings are described here as acting on sets only, but they also act on functions. This "functoriality" aspect will not be relevant in this paper, and will therefore be ignored. See [18] for more information.

2. if T_1 and T_2 are polynomial functors, then so are the product $X \mapsto T_1(X) \times T_2(X)$ and coproduct $X \mapsto T_1(X) + T_2(X)$;
3. if T is a polynomial functor, then so is $X \mapsto T(X)^A$, for each set A.

A typical example of a polynomial functor is a mapping

$$X \longmapsto 1 + (A + X)^B + (X \times X)^C.$$

An *algebra* for a polynomial functor T consists of a set X together with a function $T(X) \to X$. A *coalgebra* for T is a set X with a function $X \to T(X)$ in the reverse direction. Polynomial functors describe the "interfaces" of algebras and coalgebras, capturing the types of the operations. This will be illustrated in many examples below. This paper focuses mostly on coalgebras, assuming that algebras are relatively well-known.

Coalgebras are abstract dynamical systems. A typical example is given by a (deterministic) automaton. Ignoring initial states, such an automaton is usually described as consisting of an alphabet A and a set of states X, together with a transition function $\delta: X \times A \to X$ and a subset $F \subseteq X$ of final states. By massaging this structure a bit, using the above correspondences, we can equivalently describe such an automaton as a coalgebra: the subset $F \subseteq X$ can also be described as a "characteristic" function $X \to \{0, 1\}$. Similarly, the transition function $\delta: X \times A \to X$ corresponds to a function $X \to X^A$. Combining these two functions $X \to \{0, 1\}$ and $X \to X^A$ we get a coalgebra $X \to \{0, 1\} \times X^A$ capturing the original automaton.

For more introductory information on coalgebras, see [18], and on automata as coalgebras, see [34].

Exercises

1. Prove the following "case-lifting" lemma. For functions $f: X \to Z$, $g: Y \to Z$ and $h: Z \to W$, one has for all $a \in X + Y$,

$$h\begin{pmatrix} \mathsf{CASES}\ a\ \mathsf{OF} \\ \quad \kappa x \mapsto f(x) \\ \quad \kappa' y \mapsto g(y) \\ \mathsf{ENDCASES} \end{pmatrix} = \begin{pmatrix} \mathsf{CASES}\ a\ \mathsf{OF} \\ \quad \kappa x \mapsto h(f(x)) \\ \quad \kappa' y \mapsto h(g(y)) \\ \mathsf{ENDCASES} \end{pmatrix}.$$

   This equation is often useful in computing with coproducts.
2. Use the case notation to define the canonical distributivity map $d: (X \times Z) + (Y \times Z) \to (X + Y) \times Z$. Prove that it is an isomorphism, by explicitly constructing a map e in the opposite direction and using the previous exercise to prove that $d \circ e = \mathrm{id}$ and $e \circ d = \mathrm{id}$.

3 Specification of groups and vector spaces

In this section it will be assumed that the reader is (reasonably) familiar with the notions of group, field and vector space. We shall present specifications (or axiomatic descriptions) of (two of) these notions in the same style in which we will specify coalgebras later in this paper. The intention[3] is to make the mathematically oriented reader feel more comfortable with this style of specification. Therefore, semantical subtleties will be ignored at this stage, since the reader is already assumed to be familiar with the structures that are being specified (*i.e.* with their meaning). We will concentrate on vector spaces because they are fairly familiar and because they involve aspects—like extension and parametrisation—which we shall also see in coalgebraic specification.

We start with groups. Actually, for convenience, we start with Abelian (*i.e.* commutative) groups. Figure 1 presents a specification of Abelian groups, using *ad hoc* notation, which is hopefully self-explanatory. We have used the computer science convention of denoting the operations by explicit names, instead of by mathematical symbols, like 0, + and $(-)^{-1}$. Also, assertions have names (unit, assoc, inv, comm), so that it is easy to refer to them. The specification is presented in a certain (pseudo) format, so that it can (in principle) be processed by a machine. The text after the double slashes (//) serves as comment. Comments are valuable in specifications. The underlying set of the structure that we are specifying is written as X (throughout this paper).

BEGIN AbelianGroup
  // 'AbelianGroup' is the name of the specification
  OPERATIONS
    $\mathsf{zero}\colon 1 \longrightarrow X$ // recall, 1 is a singleton set $\{*\}$; so we have a constant
    $\mathsf{add}\colon X \times X \longrightarrow X$
    $\mathsf{inv}\colon X \longrightarrow X$
  ASSERTIONS   // named requirements that are imposed
    unit: $\left[\ \forall x \in X.\, \mathsf{add}(x, \mathsf{zero}(*)) = x \wedge \mathsf{add}(\mathsf{zero}(*), x) = x\ \right]$
    assoc: $\left[\ \forall x, y, z \in X.\, \mathsf{add}(x, \mathsf{add}(y, z)) = \mathsf{add}(\mathsf{add}(x, y), z)\ \right]$
    inv: $\left[\ \forall x \in X.\, \mathsf{add}(x, \mathsf{inv}(x)) = \mathsf{zero}(*)\ \right]$
    comm: $\left[\ \forall x, y \in X.\, \mathsf{add}(x, y) = \mathsf{add}(y, x)\ \right]$
END AbelianGroup

Fig. 1. Specification of Abelian groups

The use of the unit element zero as a function $1 \to X$ instead of as an element $\mathsf{zero} \in X$ is a bit formal. It allows us to treat all methods as functions. Also, it allows us to combine the three methods into a single function:

$$1 + (X \times X) + X \longrightarrow X \tag{1}$$

using the bijective correspondences from Section 2. This single function combines the three functions zero, add, inv into an *algebra* of the polynomial functor $\mathcal{G}$

[3] There is no claim whatsoever that this is how groups and vector spaces should be specified; the presentations in this section only serve as preparation.

given by $\mathcal{G}(X) = 1 + (X \times X) + X$. Indeed the above specification may be called an "algebraic specification". It describes a collection of algebras of the form (1) satisfying the assertions as in Figure 1. These algebras are the models of the specification, and are of course precisely the Abelian groups. Algebraic specification has developed into a field of its own in computer science, see [2] for an up-to-date source of information. It is used for the description of various kinds of datastructures, like lists and stacks, with associated operations.

We move on to vector spaces, taking the specification of fields for granted[4]. We shall simply write $+, 0, \cdot, 1$ for their operations. Fields are used as parameters in the axiomatic description of vector spaces. Also, the underlying set of vectors of a vector space is an Abelian group. This structure determines the specification in Figure 2.

BEGIN VectorSpace[K: FIELD]
  EXTENDS AbelianGroup   // from Figure 1
  OPERATIONS
    $\mathsf{scalar\_mult}\colon K \times X \longrightarrow X$
  ASSERTIONS
    // scalar multiplication and the group structure
    scalar_group_add: $\Big[\ \forall a \in K.\, \forall x, y \in X.\ \mathsf{scalar\_mult}(a, \mathsf{add}(x, y)) = \mathsf{add}(\mathsf{scalar\_mult}(a, x), \mathsf{scalar\_mult}(a, y))\ \Big]$
    // scalar multiplication and the field structure
    scalar_field_add: $\Big[\ \forall a, b \in K.\, \forall x \in X.\ \mathsf{scalar\_mult}(a + b, x) = \mathsf{add}(\mathsf{scalar\_mult}(a, x), \mathsf{scalar\_mult}(b, x))\ \Big]$
    scalar_field_mult: $\Big[\ \forall a, b \in K.\, \forall x \in X.\ \mathsf{scalar\_mult}(a \cdot b, x) = \mathsf{scalar\_mult}(a, \mathsf{scalar\_mult}(b, x))\ \Big]$
    scalar_field_unit: $\Big[\ \forall x \in X.\, \mathsf{scalar\_mult}(1, x) = x\ \Big]$
END VectorSpace

Fig. 2. Specification of vector spaces

Notice that the fact that vector spaces are parametrised by fields K is expressed by the addition [K: FIELD] to the name. Further, we say that vector spaces extend Abelian groups. This means that all the methods and assertions of Abelian groups also exist for vector spaces. Such **EXTENDS** clauses are convenient for structuring specifications. More formally it means that for each field K, the underlying polynomial functor $\mathcal{V}_K$ of the VectorSpace specification contains the polynomial functor $\mathcal{G}$ of the AbelianGroup specification as a coproduct

[4] Such a specification has to deal with partiality, since the division operation of fields is not defined on the zero element.

component:

$$\begin{aligned}\mathcal{V}_K(X) &= \mathcal{G}(X) + (K \times X) \\ &= 1 + (X \times X) + X + (K \times X).\end{aligned} \tag{2}$$

An algebra $\mathcal{V}_K(X) \to X$ of this functor thus combines four operations zero, add, inv and $\mathsf{scalar\_mult}$. A model of this specification (for a field K) is such an algebra satisfying the assertions from Figure 2. It is a vector space over K.

Given a specification as above, there are (at least) two things one can do.

1. Describe models of the specification. For example, for an arbitrary field K, one obtains an obvious model by choosing K itself for X, with its own Abelian group structure, and with scalar multiplication given by its multiplication. Similarly, K^n forms a model, for each $n \in \mathbb{N}$. Knowing that a specification has a (non-trivial) model is important, because it shows its consistency.
2. Develop the theory of the specification. For example, one can derive the following well-known consequences of the assertions in the VectorSpace specification:

$$\begin{aligned}\mathsf{scalar\_mult}(0, x) &= \mathsf{zero}(*) \\ \mathsf{scalar\_mult}(a, \mathsf{zero}(*)) &= \mathsf{zero}(*) \\ \mathsf{scalar\_mult}(a, \mathsf{inv}(x)) &= \mathsf{scalar\_mult}(-a, x).\end{aligned}$$

   Theory development often involves further definitions, like basis or dimension for vector spaces.

A third activity is constructing refinements. These may be understood as "relative models", and will be further investigated in Section 10.

Exercises

1. Write down specifications—in the style of this section—for lattices and for Boolean algebras (see *e.g.* [4] for details about these notions).

4 A first coalgebraic specification: binary trees

In the previous section we have seen specifications whose methods could be described as algebras $T(X) \to X$. An obvious step is to consider specifications based on coalgebras $X \to T(X)$. This is what coalgebraic specification is all about. It allows us to describe an entirely new class of structures, having a state space with associated operations.

We start with an example. Let us consider certain trees with labels from a fixed set A. The trees we wish to consider are possibly infinite binary trees. This means that each node has a label, and has either no successor trees (also called children), or has two successor trees. The characteristic coalgebraic operations for such trees are $\mathsf{label}\colon X \to A$, giving the label at the top of the tree, and $\mathsf{children}\colon X \to 1 + (X \times X)$ giving the left and right children, if any. Note that these two functions can be combined into a single coalgebra of the form:

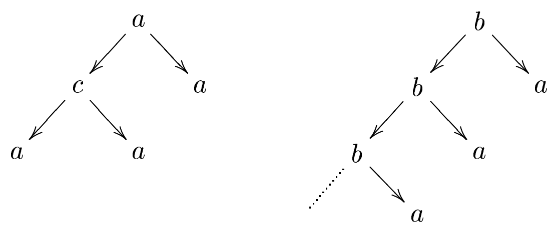
$$X \longrightarrow A \times (1 + (X \times X))$$

For a "tree" $x \in X$ the result $\mathsf{children}(x)$ is either of the form $\kappa *$ in the left $+$-component 1, telling that x has no children, or of the form $\kappa'(x_1, x_2)$ in the right component $X \times X$, with x_1 as first (or left) and x_2 as second (right) child. From here one can continue unfolding the tree x by inspecting $\mathsf{children}(x_1)$ and $\mathsf{children}(x_2)$ (and their labels). If all of these paths stop at some stage, x behaves like a finite tree, but if one of them continues indefinitely we get an infinite tree. Note that each such tree has at least one label.

For a fixed set A, we say, by analogy with vector space, that a *binary tree space* over A consists of a set U together with a coalgebra $U \to A \times (1 + (U \times U))$, which we shall typically write as a pair of functions $\mathsf{label}\colon U \to A$ and $\mathsf{children}\colon U \to 1 + (U \times U)$. An example of a binary tree space over $A = \{a, b, c\}$ is the set of states $U = \{0, 1, 2, 3, 4\}$ with functions:

$$\begin{array}{ll} \mathsf{label}(0) = a & \mathsf{children}(0) = \kappa'(3,2) \\ \mathsf{label}(1) = b & \mathsf{children}(1) = \kappa'(1,2) \\ \mathsf{label}(2) = a & \mathsf{children}(2) = \kappa * \\ \mathsf{label}(3) = c & \mathsf{children}(3) = \kappa'(2,2) \\ \mathsf{label}(4) = b & \mathsf{children}(4) = \kappa'(4,2) \end{array} \tag{3}$$

The unfoldings (or behaviours) of the states $0 \in U$ and $1 \in U$ then look as follows.



Such binary trees can occur as the (possibly infinite) behaviour of a process, where a process may be understood as a program that is supposed to be running forever, like an operating system on a computer. For terminating processes one gets finite behaviour.

In a next step we like to include a size method in the specification of these binary trees. An immediate problem is that the size of such trees need not yield a natural number. Therefore we choose the "extended natural numbers" $1 + \mathbb{N}$ as range of the size function, where $\kappa *$ denotes infinity. Addition of infinity to an extended natural number always yields infinity. These two methods $\mathsf{children}$ and size form the two main ingredients of our first coalgebraic specification of binary trees in Figure 3.

In this specification we use parametrisation by the type (or set) of labels, like in Figure 2. An important point is that the types of the methods in the specification determines a polynomial functor, which, in this case, is

$$\mathcal{BT}(X) = A \times (1 + (X \times X)) \times (1 + \mathbb{N}).$$

It is obtained by putting a Cartesian product $\times$ between the result types of the individual methods. A crucial observation is that the three methods label,

```
BEGIN BinaryTreeSpace[A: TYPE]
  METHODS   // alternative terminology for operations
    label: X ⟶ A
    children: X ⟶ 1 + (X × X)
    size: X ⟶ 1 + ℕ
  ASSERTIONS
    size_def: [ ∀x ∈ X. size(x) = CASES children(x) OF
                                     κu ↦ κ'1
                                     κ'(v1, v2) ↦ CASES size(v1) OF
                                                    κu ↦ κ*
                                                    κ'n ↦ CASES size(v2) OF
                                                            κu ↦ κ*
                                                            κ'm ↦ κ'(1 + n + m)
                                                          ENDCASES
                                                  ENDCASES
                                   ENDCASES ]
END BinaryTreeSpace
```

Fig. 3. Specification of binary trees, version 1.

children, size correspond to a coalgebra $X \to \mathcal{BT}(X)$ of the polynomial functor $\mathcal{BT}$. The specification may be understood as describing a collection of coalgebras of $\mathcal{BT}$ satisfying an assertion. Thus, coalgebraic specification is much like algebraic specification as described in the previous section.

The assertion 'size_def' for the size method is non-trivial; it says that empty trees have size 1 (since they do have a label), and that if one of the successors of a non-empty tree has size infinity ($\kappa *$), then the whole tree has size infinity. Otherwise the size is the sum of the sizes of the successors plus one. With this definition we can compute for the earlier example (3) that:

$$
\begin{array}{ll}
\mathsf{size}(2) = \kappa' 1 & \text{since } \mathsf{children}(2) = \kappa * \\
\mathsf{size}(3) = \kappa'(1+1+1) = \kappa' 3 & \text{since } \mathsf{children}(3) = \kappa'(2,2) \\
\mathsf{size}(0) = \kappa'(1+3+1) = \kappa' 5 & \text{since } \mathsf{children}(0) = \kappa'(3,2) \\
\mathsf{size}(1) = \kappa * & \\
\mathsf{size}(4) = \kappa * &
\end{array}
$$

The latter two equations holds since if $\mathsf{size}(1)$ were $\kappa' n$, for some $n \in \mathbb{N}$, then assertion 'size_def' in Figure 3 yields $\kappa' n = \mathsf{size}(1) = \kappa'(n+1+1)$, which is impossible. Hence $\mathsf{size}(1)$ must be $\kappa *$. Similarly for $\mathsf{size}(4)$.

For those readers who are not so familiar with the style of specification as in Figure 3 we shall give a reformulation of the binary tree space specification as an axiomatic description in mathematical style:

> Let A be an arbitrary set. A *binary tree space* over A consists of a set X, elements of which will be called (binary) trees, together with three

operations $\mathsf{label}: X \to A$, $\mathsf{children}: X \to 1 + (X \times X)$ and $\mathsf{size}: X \to 1 + \mathbb{N}$ satisfying, for all $x \in X$,

1. If $\mathsf{children}(x) = \kappa *$, then $\mathsf{size}(x) = \kappa' 1$;
2. If $\mathsf{children}(x) = \kappa'(x_1, x_2)$ and either $\mathsf{size}(x_1) = \kappa *$ or $\mathsf{size}(x_2) = \kappa *$, then $\mathsf{size}(x) = \kappa *$;
3. If $\mathsf{children}(x) = \kappa'(x_1, x_2)$ and $\mathsf{size}(x_1) = \kappa' n_1$ and $\mathsf{size}(x_2) = \kappa' n_2$, then $\mathsf{size}(x) = \kappa'(n_1 + n_2 + 1)$.

In case $\mathsf{children}(x) = \kappa'(x_1, x_2)$ we shall call x_1 and x_2 the successor trees or children of the tree x.

We consider some examples of binary tree spaces (*i.e.* of models of the specification BinaryTreeSpace) over an arbitrary set A.

1. A trivial example is the empty set 0 with obvious operations given by the empty functions. If the set A is non-empty, the singleton set 1 is also a binary tree space.
2. We can turn the set of all infinite trees with labels from A into a binary tree space. This set, call it $\mathsf{InfTree}(A)$, can be defined as the set of functions $A^{2^\star}$ of all functions from the free monoid $2^\star$ of finite sequences of elements from $2 = \{0, 1\}$ to A. For $\phi \in \mathsf{InfTree}(A) = A^{2^\star}$ we define

$$\begin{aligned} \mathsf{label}(\phi) &= \phi(\langle\rangle) \\ \mathsf{children}(\phi) &= \kappa'(\lambda\alpha \in 2^\star.\, \phi(0 \cdot \alpha), \lambda\alpha \in 2^\star.\, \phi(1 \cdot \alpha)) \\ \mathsf{size}(\phi) &= \kappa * . \end{aligned}$$

3. Next consider the set of both finite and infinite trees:

$$\begin{aligned} \mathsf{FinInfTree}(A) \stackrel{\text{def}}{=} \{(a, \phi): A \times (1 + (A \times A))^{2^\star} \mid {} & \forall \alpha \in 2^*.\, \forall b \in \{0, 1\}. \\ & \phi(\alpha) = \kappa * \Rightarrow \phi(b \cdot \alpha) = \kappa * \} \end{aligned}$$

with operations:

$$\begin{aligned} \mathsf{label}(a, \phi) &= a \\ \mathsf{children}(a, \phi) &= \mathsf{CASES}\ \phi(\langle\rangle)\ \mathsf{OF} \\ &\qquad \kappa x \mapsto \kappa * \\ &\qquad \kappa'(a_0, a_1) \mapsto \kappa'\big((a_0, \lambda\alpha \in 2^\star.\, \phi(0 \cdot \alpha)), \\ &\qquad\qquad\qquad (a_1, \lambda\alpha \in 2^\star.\, \phi(1 \cdot \alpha))\big) \\ &\quad \mathsf{ENDCASES} \\ \mathsf{size}(a, \phi) &= \ldots \end{aligned}$$

The definition of size on $\mathsf{FinInfTree}(A)$ is non-trivial. One can of course say that size is determined by equation 'size_def' from Figure 3, but one has still has to show that such a function exists. One way to go about is to first define $\mathsf{FinTree}(A) \subseteq \mathsf{FinInfTree}(A)$ to be the smallest subset F satisfying: $(a, \phi) \in F$ if either and $\mathsf{children}(a, \phi) = \kappa *$ or $\mathsf{children}(a, \phi) = \kappa'((a_0, \phi_0), (a_1, \phi_1))$ and both $(a_0, \phi_0) \in F$ and $(a_1, \phi_1) \in F$. The elements of $\mathsf{FinTree}(A)$ are the "finite trees", and their size is determined by the computation rules in equation 'size_def' from Figure 3. For elements not in $\mathsf{FinTree}(A)$, the size is $\kappa *$[5].

[5] We will elaborate on this "μ-calculus" style definition in Section 8.

4. The subset $\mathsf{FinTree}(A) \subseteq \mathsf{FinInfTree}(A)$ defined above (in 3) with inherited operations.

Thus we have seen that binary tree spaces exist as non-trivial mathematical structures. We conclude this section with a warning that coalgebraic specification is a subtle matter.

4.1 Elements of binary trees

Suppose that we would like to add to the BinTreeSpace specification in Figure 3 a method

$$\mathsf{elem}\colon X \times A \longrightarrow \{\mathsf{false}, \mathsf{true}\}$$

telling whether a tree $x \in X$ contains an element $a \in A$ or not. The associated assertion that probably first comes to mind is:

$$\begin{array}{l}\mathsf{elem\_def} : \big[\, \forall x \in X.\, \forall a \in A.\, \mathsf{elem}(x, a) = \\ \qquad (\mathsf{label}(x) = a) \vee \mathsf{CASES}\ \mathsf{children}(x)\ \mathsf{OF} \\ \qquad\qquad\qquad \kappa u \mapsto \mathsf{false} \\ \qquad\qquad\qquad \kappa'(v_1, v_2) \mapsto \mathsf{elem}(v_1, a) \vee \mathsf{elem}(v_2, a) \\ \qquad\qquad\quad \mathsf{ENDCASES}\,\big]\end{array}$$

But this assertion is not good enough, in the sense that it allows interpretations which are probably unwanted. For example, on the binary tree space $\mathsf{InfTree}(A)$ introduced above, we can define $\mathsf{elem}(\phi) = \mathsf{true}$, for all $\phi \in \mathsf{InfTree}(A)$. Then the assertion elem_def holds, although we do not have a meaningful membership method.

Later, in Section 7 we shall introduce a temporal logic for coalgebras which allows us to express an appropriate assertion for the elem method. Then, $\mathsf{elem}(x, a)$ will be equal to: "there is a (not-necessarily direct) successor tree y of x with a as label"[6].

Exercises

1. Check that the subset $\mathsf{FinTree}(A) \subseteq \mathsf{FinInfTree}(A)$ is closed under the children operation defined on $\mathsf{FinInfTree}(A)$.
2. Characterise the subset $\mathsf{InfTree}(A) \subseteq \mathsf{FinInfTree}(A)$ in term of the children operation, by analogy with the above definition of $\mathsf{FinTree}(A) \subseteq \mathsf{FinInfTree}(A)$.
[See also the definitions of Fin and Inf in Section 8.]

5 Bisimulations and bisimilarity

Suppose we have an arbitrary coalgebra $c\colon X \to T(X)$ for a polynomial functor T. At this level of abstraction, we often a call the elements of X *states*, and call

[6] This definition yields the "smallest" function elem satisfying the above assertion elem_def with the first '=' replaced by '$\Leftarrow$'.

X itself the *state space*. The function c provides X with certain operations—like label, children and size in the previous section. These operations give us certain access to the state space X. They may allow us to observe certain things (like the current label, or the size), and they may allow us to "modify states", or to "move to next states" (like the successor trees). Typically for coalgebras, we can observe and modify, but we have no means for constructing new states. The behaviour of a state $x \in X$ is all that we can observe about x, either directly or indirectly (via its successor states).

In this situation it may happen that two states have the same behaviour. In that case we cannot distinguish them with the operations (of the coalgebra) that we have at our disposal. The two states need not be equal then, since the operations may only give limited access to the state space, and certain aspects be may unobservable. When two states x, y are observationally indistinguishable, they are called *bisimilar*. This is written as $x \underline{\leftrightarrow} y$. For example, the states 1 and 4 from the binary tree space $U = \{0, 1, 2, 3, 4\}$ in the previous section are bisimilar; they have the same unfoldings.

In this section we shall formally introduce the notion of bisimilarity $\underline{\leftrightarrow}$, and show how it can be used in (coalgebraic) specification. Therefore we shall first introduce what is called *relation lifting*. Recall that a polynomial functor T is a mapping $X \mapsto T(X)$ of sets to sets. We shall define an associated mapping, called $\mathrm{Rel}(T)$, which maps a relation $R \subseteq X \times X$ to a new relation $\mathrm{Rel}(T)(R) \subseteq T(X) \times T(X)$. This mapping $\mathrm{Rel}(T)$ will be defined by induction on the structure of the polynomial functor T.

1. If T is the identity mapping, then so is $\mathrm{Rel}(T)$.
2. If T is the constant mapping $X \mapsto A$, then $\mathrm{Rel}(T)$ is the constant mapping $R \mapsto =_A$, where $=_A \subseteq A \times A$ is the equality relation consisting of $\{(a, a) \mid a \in A\}$.
3. If T is the product functor $X \mapsto T_1(X) \times T_2(X)$, then
$$\begin{aligned}\mathrm{Rel}(T)(R) &= \{((x_1, x_2), (y_1, y_2)) \mid \mathrm{Rel}(T_1)(R)(x_1, y_1) \wedge \mathrm{Rel}(T_2)(R)(x_2, y_2)\} \\ &\subseteq \big(T_1(X) \times T_2(X)\big) \times \big(T_1(X) \times T_2(X)\big)\end{aligned}$$
4. If T is the coproduct functor $X \mapsto T_1(X) + T_2(X)$, then
$$\begin{aligned}\mathrm{Rel}(T)(R) &= \{(\kappa x_1, \kappa y_1) \mid \mathrm{Rel}(T_1)(R)(x_1, y_1)\} \\ &\quad \cup \{(\kappa' x_2, \kappa' y_2) \mid \mathrm{Rel}(T_2)(R)(x_2, y_2)\} \\ &\subseteq \big(T_1(X) + T_2(X)\big) \times \big(T_1(X) + T_2(X)\big)\end{aligned}$$
5. If T is the function space functor $X \mapsto T_1(X)^A$, then
$$\begin{aligned}\mathrm{Rel}(T)(R) &= \{(f, g) \mid \forall a \in A.\, \mathrm{Rel}(T_1)(R)(f(a), g(a))\} \\ &\subseteq \big(T_1(X)^A\big) \times \big(T_1(X)^A\big)\end{aligned}$$

For example, for a functor $T(X) = A + (X \times X)$ and a relation $R \subseteq X \times X$, the relation $\mathrm{Rel}(T)(R) \subseteq T(X) \times T(X)$ contains all pairs $(\kappa a, \kappa a)$, for $a \in A$, and all pairs $(\kappa'(x_1, x_2), \kappa'(y_1, y_2))$ with $R(x_1, y_1)$ and $R(x_2, y_2)$. Informally, $\mathrm{Rel}(T)(R)$ contains all pairs $(z, w) \in T(X) \times T(X)$ whose constituents in a constant set are equal, and whose constituents in X are related by R.

Definition 1. *Let T be a polynomial functor, with relation lifting Rel(T) as defined above. Further, let $c: X \to T(X)$ be a coalgebra for T.*

1. *A relation $R \subseteq X \times X$ is a bisimulation (with respect to the coalgebra c) if forall $x, y \in X$,*
$$R(x,y) \Longrightarrow \mathit{Rel}(T)(R)(c(x), c(y)).$$
2. *The bisimilarity relation $\underline{\leftrightarrow} \subseteq X \times X$ (with respect to c) is defined as the greatest bisimulation; that is,*
$$x \mathrel{\underline{\leftrightarrow}} y \Longleftrightarrow \exists R \subseteq X \times X.\, R \text{ is a bisimulation, and } R(x,y).$$

Bisimilarity is one of the fundamental notions in the theory of coalgebras. Notice that the above definition gives for each polynomial functor and for each coalgebra thereof an appropriate tailor-made definition of bisimilarity. The definition via relation lifting is convenient in a logical setting, because it is based on induction (on the structure of the polynomial functor). Therefore it can easily be translated in the logical language of proof tools—see the discussion in point 2. in Section 1. One can prove that bisimilarity is an equivalence relation, but this will not be done here. The proof relies on some basic properties of relation lifting, which can be proved by induction on the structure of the functor involved.

The notion of bisimilarity comes alive via examples. Let us return to the binary tree space specification from Figure 3 in the previous section. We have already seen that the functor involved is $\mathcal{BT}(X) = A \times (1 + (X \times X)) \times (1 + \mathbb{N})$. We shall elaborate what a bisimulation relation is for an arbitrary coalgebra $c = \langle \mathsf{label}, \mathsf{children}, \mathsf{size} \rangle : X \to \mathcal{BT}(X)$ of this functor. Therefore we first compute the relation lifting operation $\mathrm{Rel}(\mathcal{BT})$. It turns a relation $R \subseteq X \times X$ into a relation $\mathrm{Rel}(\mathcal{BT})(R) \subseteq \mathcal{BT}(X) \times \mathcal{BT}(X)$, in the following way. For a pair $((x_1, x_2, x_3), (y_1, y_2, y_3)) \in \mathcal{BT}(X) \times \mathcal{BT}(X)$,

$$\begin{array}{l}
\mathrm{Rel}(\mathcal{BT})(R)((x_1, x_2, x_3), (y_1, y_2, y_3)) \\
\;\Leftrightarrow \mathrm{Rel}(A)(x_1, y_1) \wedge \mathrm{Rel}(1 + (X \times X))(x_2, y_2) \wedge \mathrm{Rel}(1 + \mathbb{N})(x_3, y_3) \\
\;\Leftrightarrow (x_1 = x_2) \wedge \\
\qquad \big((x_2 = y_2 = \kappa *) \vee \\
\qquad\; (x_2 = \kappa'(x_{20}, x_{21}) \wedge y_2 = \kappa'(y_{20}, y_{21}) \wedge R(x_{20}, y_{20}) \wedge R(x_{21}, y_{21})) \big) \wedge \\
\qquad (x_3 = y_3).
\end{array}$$

Such $R \subseteq X \times X$ is a bisimulation for a coalgebra $c = \langle \mathsf{label}, \mathsf{children}, \mathsf{size} \rangle : X \to \mathcal{BT}(X)$ if for all $x, y \in X$,

$$R(x,y) \Longrightarrow \mathrm{Rel}(\mathcal{BT})(R)(c(x), c(y))$$

i.e. if,

$$R(x,y) \Longrightarrow \begin{cases}
\mathsf{label}(x) = \mathsf{label}(y) \wedge \\
\big((\mathsf{children}(x) = \mathsf{children}(y) = \kappa *) \vee \\
\quad (\mathsf{children}(x) = \kappa'(x_0, x_1) \wedge \mathsf{children}(y) = \kappa'(y_0, y_1) \wedge \\
\quad R(x_0, y_0) \wedge R(x_1, y_1))\big) \wedge \\
\mathsf{size}(x) = \mathsf{size}(y)
\end{cases}$$

This means that elements which are related by a bisimulation R have equal labels and sizes, and either both have no children, or both have children, which are again pairwise related by R. Bisimilarity $\underline{\leftrightarrow}$ for binary trees is the greatest such bisimulation. Since it is an equivalence relation (and is closed under the operations), it behaves very much like an equality relation. In fact, bisimilarity is an important ingredient of assertions in coalgebraic specifications. There, one generally wishes to avoid using actual equality between states, and so one uses bisimilarity instead. The reason for not wanting equality between states is that it severely restricts the possible models[7] of the specification. But also, if states are indistinguishable, then they can be considered as equal from the outside.

BEGIN BinaryTreeSpace[A: TYPE]
METHODS
$\mathsf{label}\colon X \longrightarrow A$
$\mathsf{children}\colon X \longrightarrow 1 + (X \times X)$
$\mathsf{size}\colon X \longrightarrow 1 + \mathbb{N}$
$\mathsf{mirror}\colon X \longrightarrow X$
ASSERTIONS
size_def: $\Big[$ see Figure 3 $\Big]$
label_mirror: $\Big[\ \forall x \in X.\, \mathsf{label}(\mathsf{mirror}(x)) = \mathsf{label}(x)\ \Big]$
children_mirror: $\Big[\ \forall x \in X.$ CASES $\mathsf{children}(x)$ OF

$$\begin{array}{l}
\kappa u \mapsto \mathsf{children}(\mathsf{mirror}(x)) = \kappa * \\
\kappa'(v_1, v_2) \mapsto \text{CASES } \mathsf{children}(\mathsf{mirror}(x)) \text{ OF} \\
\qquad \kappa w \mapsto \mathsf{false} \\
\qquad \kappa'(z_1, z_2) \mapsto (z_1 \underline{\leftrightarrow} \mathsf{mirror}(v_2)) \wedge \\
\qquad\qquad (z_2 \underline{\leftrightarrow} \mathsf{mirror}(v_1)) \\
\quad \text{ENDCASES} \\
\text{ENDCASES} \Big]
\end{array}$$

END BinaryTreeSpace

Fig. 4. Specification of binary trees, version 2.

In Figure 4 we continue the specification of binary tree spaces. What we add is a mirror method which swaps the children of trees (if any), leaving the labels unaffected. The assertion children_mirror use bisimilarity to express that the left and right children of a mirrored tree with children are bisimilar to the mirrored right and left children.

The reader might have expected an additional assertion stating that mirroring does not change the size. In fact, this can be derived. Also that mirroring is its own inverse, up to bisimilarity.

[7] As a typical example, consider a specification of a system with a "do" and an "undo" operation. The equation $\mathsf{undo}(\mathsf{do}(x)) = x$ will not hold in "history" models where one internally keeps track of the operations that have been applied.

Proposition 1. *From the binary tree space assertions in Figure 4 one can derive, for all $x \in X$,*

1. $\mathit{size}(\mathit{mirror}(x)) = \mathit{size}(x)$
2. $\mathit{mirror}(\mathit{mirror}(x)) \mathrel{\underline{\leftrightarrow}} x$.

Proof. We only give a sketch, leaving details to the reader.

1. First, one can prove by well-founded induction,

$$\forall n \in \mathbb{N}.\, \mathsf{size}(x) = \kappa' n \Leftrightarrow \mathsf{size}(\mathsf{mirror}(x)) = \kappa' n$$

   The result then follows easily.
2. According to the definition of bisimilarity $\underline{\leftrightarrow}$, it suffices to find a bisimulation $R \subseteq X \times X$ with $R(\mathsf{mirror}(\mathsf{mirror}(x)), x)$. Because $\underline{\leftrightarrow}$ is the greatest bisimulation, we then get $R \subseteq \underline{\leftrightarrow}$. One can use

$$R = \{(x, y) \mid x \mathrel{\underline{\leftrightarrow}} \mathsf{mirror}(\mathsf{mirror}(y))\}.$$

□

Exercises

1. Let T be an arbitrary polynomial functor. Prove the following basic properties about the associated relation lifting operation $\mathrm{Rel}(T)$.
   (a) $\mathrm{Rel}(T)(=_X) = =_{T(X)}$;
   (b) $\mathrm{Rel}(T)(R^{op}) = \big(\mathrm{Rel}(T)(R)\big)^{op}$, where $R^{op} = \{(y, x) \mid (x, y) \in R\}$;
   (c) $\mathrm{Rel}(T)(R \circ S) = \mathrm{Rel}(T)(R) \circ \mathrm{Rel}(T)(S)$;
   (d) $R \subseteq S$ implies $\mathrm{Rel}(T)(R) \subseteq \mathrm{Rel}(T)(S)$;
   (e) R is symmetric implies $\mathrm{Rel}(T)(R)$ is symmetric;
   (f) For a collection of relations $(R_i \subseteq X \times X)_{i \in I}$ one has $\mathrm{Rel}(T)(\bigcap_{i \in I} R_i) = \bigcap_{i \in I} \mathrm{Rel}(T)(R_i)$.
   Conclude that for a coalgebra $c\colon X \to T(X)$, the union $\underline{\leftrightarrow}$ of all bisimulations on X is itself a bisimulation, and is also an equivalence relation.
2. Prove that the relation R in the proof of Proposition 1.2 is a bisimulation.
3. Check that the above definition of relation lifting $\mathrm{Rel}(T)$ sending a relation $R \subseteq X \times X$ to $\mathrm{Rel}(T)(R) \subseteq T(X) \times T(X)$ in fact also works on relations $R \subseteq X \times Y$, with different carrier sets, and then yields a relation $\mathrm{Rel}(T)(R) \subseteq T(X) \times T(Y)$. For two coalgebras $X \xrightarrow{c} T(X)$ and $Y \xrightarrow{d} T(Y)$ we then call a relation $R \subseteq X \times Y$ a bisimulation (w.r.t. (c, d)) if $R(x, y) \Rightarrow \mathrm{Rel}(T)(R)(c(x), d(y))$ for all $x \in X$ and $y \in Y$. And we say that a function $f\colon X \to Y$ is a homomorphism (of coalgebras) if its graph $\mathcal{G}(f) = \{(x, f(x)) \mid x \in X\}$ is a bisimulation.
   Prove that the identity function is a homomorphism, and also that the composition of two homomorphisms is again a homomorphism. Thus one can form a category $\mathrm{CoAlg}(T)$ of coalgebras and homomorphisms between them.
   [Remark: what we present is an alternative to the standard definition, which uses the action of polynomial functors on functions (which we have not mentioned here, see *e.g.* [18]). It says that a function $f\colon X \to Y$ as above is a homomorphism if and only if the following diagram commutes.

$$\begin{tikzcd}
T(X) \arrow[r, "T(f)"] & T(Y) \\
X \arrow[u, "c"] \arrow[r, "f"'] & Y \arrow[u, "d"']
\end{tikzcd}$$

The reader familiar with this '$T(f)$' may wish to prove the equivalence of these two definitions.]

6 Invariants

Bisimulations are binary predicates on the state space of a coalgebra, which are closed under the operations. We will now introduce invariants, which are unary predicates that are closed under the operations. They are important for several reasons.

1. An invariant expresses a property which is maintained by all operations. If it also holds in initial states, then it holds in all reachable states. Thus, intended "safety" properties of a specification—like: the value of the integer (attribute) i will always be positive—are expressed via invariants.
2. Invariants are used for defining the temporal operators $\Box$ (henceforth) and $\Diamond$ (eventually) in a coalgebraic setting, see Section 7.
3. Invariants are crucial for refinements between coalgebraic specifications, see Section 10.

Bisimulations were introduced in Section 5 via relation lifting. Similarly, we shall introduce invariants via what is called predicate lifting. For a polynomial functor T we define a mapping $\mathrm{Pred}(T)$ which sends a predicate $P \subseteq X$ on X to a predicate $\mathrm{Pred}(T)(P)$ on $T(X)$, by induction on the structure of T. This goes as follow.

1. If T is the identity mapping, then so is $\mathrm{Pred}(T)$.
2. If T is the constant mapping $X \mapsto A$, then $\mathrm{Pred}(T)$ is the constant mapping $P \mapsto A$, where A is considered as the "truth" predicate $A \subseteq A$.
3. If T is the product functor $X \mapsto T_1(X) \times T_2(X)$, then
$$\begin{aligned}\mathrm{Pred}(T)(P) &= \{(x, y) \mid \mathrm{Pred}(T_1)(P)(x) \wedge \mathrm{Pred}(T_2)(P)(y)\} \\ &\subseteq T_1(X) \times T_2(X)\end{aligned}$$
4. If T is the coproduct functor $X \mapsto T_1(X) + T_2(X)$, then
$$\begin{aligned}\mathrm{Pred}(T)(P) &= \{\kappa x \mid \mathrm{Pred}(T_1)(P)(x)\} \cup \{\kappa' y \mid \mathrm{Pred}(T_2)(P)(y)\} \\ &\subseteq T_1(X) + T_2(X)\end{aligned}$$
5. If T is the function space functor $X \mapsto T_1(X)^A$, then
$$\begin{aligned}\mathrm{Pred}(T)(P) &= \{f \mid \forall a \in A.\, \mathrm{Pred}(T_1)(P)(f(a))\} \\ &\subseteq T_1(X)^A\end{aligned}$$

For the polynomial functor $T(X) = A + (X \times X)$, a predicate $P \subseteq X$ is lifted to the relation $\mathrm{Pred}(T)(P) \subseteq T(X)$ containing all elements κa and those elements $\kappa'(x_1, x_2)$ where both $P(x_1)$ and $P(x_2)$. In general, $\mathrm{Pred}(T)(P)$ contains all elements from $T(X)$ where P holds on all X-positions in $T(X)$.

Definition 2. *Let $c: X \to T(X)$ be a coalgebra of a polynomial functor T, and $P \subseteq X$ be a predicate on its state space.*

1. *A new predicate $NextTime(c)(P) \subseteq X$ is defined as*

$$NextTime(c)(P)(x) \Longleftrightarrow Pred(T)(P)(c(x)).$$

   It expresses that P holds for all successor states of x.
2. *The predicate $P \subseteq X$ is then an invariant for c if it satisfies, for all $x \in X$,*

$$P(x) \Longrightarrow NextTime(c)(P)(x).$$

The requirement in the definition expresses that if an invariant P holds in a state x, then it holds on each successor state of x. Using some elementary properties of predicate lifting, one can show that invariants are closed under conjunction $\wedge$. Also, the predicate which is always true is an invariant. Further, if we have an arbitrary collection $(P_i)_{i \in I}$ of invariants, then $\forall i \in I.\, P_i$ is an invariant, see the exercises below.

Let us consider the binary tree space specification from Figure 4. The polynomial functor that captures the interface of this specification is:

$$\mathcal{BT}_2(X) = A \times (1 + (X \times X)) \times (1 + \mathbb{N}) \times X.$$

For a predicate $P \subseteq X$, the lifting $\mathrm{Pred}(\mathcal{BT}_2)(P) \subseteq \mathcal{BT}_2(X)$ on a 4-tuple $(x_1, x_2, x_3, x_4) \in \mathcal{BT}_2(X)$ is:

$$\begin{array}{l} \mathrm{Pred}(\mathcal{BT}_2)(P)(x_1, x_2, x_3, x_4) \\ \Leftrightarrow \mathrm{Pred}(A)(P)(x_1) \wedge \mathrm{Pred}(1 + (X \times X))(P)(x_2) \wedge \\ \qquad \mathrm{Pred}(1 + \mathbb{N})(P)(x_3) \wedge \mathrm{Pred}(X)(P)(x_4) \\ \Leftrightarrow \big(x_2 = \kappa * \vee (x_2 = \kappa'(x_{20}, x_{21}) \wedge P(x_{20}) \wedge P(x_{21}))\big) \wedge P(x_4). \end{array}$$

An invariant on a coalgebra $c = \langle \mathsf{label}, \mathsf{children}, \mathsf{size}, \mathsf{mirror} \rangle : X \to \mathcal{BT}_2(X)$ is a predicate $P \subseteq X$ with

$$P(x) \Longrightarrow \mathrm{Pred}(\mathcal{BT}_2)(P)(c(x))$$

i.e. with

$$P(x) \Longrightarrow \begin{cases} \big(\mathsf{children}(x) = \kappa'(x_0, x_1) \Rightarrow P(x_0) \wedge P(x_1)\big) \wedge \\ P(\mathsf{mirror}(x)) \end{cases}$$

An invariant P thus holds on all successor states of an $x \in P$, produced either as children of x, or as mirror of x.

A concrete example of an invariant for the binary tree specification is the predicate stating that the size is finite: $\mathrm{FinSize}(x) \Leftrightarrow (\mathsf{size}(x) \neq \kappa *)$. Clearly, if $\mathrm{FinSize}(x)$, then none of the successor trees can have infinite size; and also $\mathrm{FinSize}(\mathsf{mirror}(x))$ holds because mirroring does not change the size, see Proposition 1 (1).

As we have seen, invariants and bisimulations on a coalgebra are unary and binary predicates which are closed under the operations. The more operations the coalgebra has, the more closure requirements there are. The approach that we follow—via predicate and relation lifting—produces tailor-made requirements, following the structure of the interface (as given by a polynomial functor) of a coalgebra. Since these requirements follow from the structure, they can be generated automatically. That is precisely what is done by the LOOP tool (from [7]).

As a final remark we add that predicates and relations which are closed under the operations of an algebra (instead of a coalgebra) also make sense, see [30, 32, 8]. A relation $R \subseteq X \times X$ on the carrier set of an algebra $a: T(X) \to X$ is closed under the operations if:

$$\mathrm{Rel}(T)(R)(a(x), a(y)) \Longrightarrow R(x, y).$$

In this case one may call R a *congruence*. But note that such an R need not be an equivalence relation. There is no special name in mathematics for a predicate $P \subseteq X$ which is closed under the operations of an algebra a, *i.e.* which satisfies:

$$\mathrm{Pred}(T)(P)(a(x)) \Longrightarrow P(x).$$

Such a P is called an *inductive predicate* in [8] (since such predicates form the assumptions of induction arguments), but *invariant* would be a reasonable name also in this algebraic case.

Exercises

1. Prove for an arbitrary polynomial functor T that predicate lifting $\mathrm{Pred}(T)$ satisfies:
   (a) $P \subseteq Q$ implies $\mathrm{Pred}(T)(P) \subseteq \mathrm{Pred}(T)(Q)$;
   (b) $\mathrm{Pred}(T)(X \subseteq X) = (T(X) \subseteq T(X))$;
   (c) $\mathrm{Pred}(T)(\bigcap_{i \in I} P_i) = \bigcap_{i \in i} \mathrm{Pred}(T)(P_i)$.
2. Let $c: X \to T(X)$ be a coalgebra of a polynomial functor T. Prove that:
   (a) $P \subseteq X$ is an invariant if and only if $\{(x, y) \mid x = y \wedge P(x)\}$ is a bisimulation.
   (b) If R is a bisimulation, then $\pi R = \{x \mid \exists y.\, R(x, y)\}$ is an invariant.
3. Elaborate what it means for a relation to be a congruence for Abelian groups and vector spaces, using the above formulation (and the interface functors from Section 3).

7 Temporal logic for coalgebras

Temporal logic is a formalism for reasoning about assertions varying with time. Time may be understood here in a loose sense, including situations where one deals with later or earlier stages, or with successor and predecessor states. Typical operators of temporal logic are "henceforth" $\Box$ and "eventually" $\Diamond$. It was Pnueli [31, 26] who first argued that temporal logic could be useful for reasoning about computer systems with potentially infinite behaviour, such as controllers which are meant to be running forever. Hence it comes as no surprise that the operators of temporal logic arise quite naturally for coalgebras, because they abstractly capture dynamical systems, typically with infinite behaviour. In this section we shall define $\Box$ and $\Diamond$ in terms of invariants, and show how to use them in the running example of binary trees.

Let $c\colon X \to T(X)$ be an arbitrary coalgebra of a polynomial functor T. For a predicate $P \subseteq X$ we shall define two new predicates $\Box(c)(P) \subseteq X$ and $\Diamond(c)(P) \subseteq X$. If the coalgebra c is clear from the context, we often omit it, and simply write $\Box(P)$ and $\Diamond(P)$. For an element $x \in X$, we define the predicate "henceforth P" (with respect to c) as:

$$\Box(c)(P)(x) \Leftrightarrow \exists I \subseteq X.\, I \text{ is an invariant for } c \text{ and } I \subseteq P \text{ and } I(x). \qquad (4)$$

This says that $\Box(c)(P)$ is the greatest invariant contained in P[8]. The associated "eventually" operator $\Diamond$ is defined as $\neg\Box\neg$, or, more precisely as:

$$\Diamond(c)(P)(x) \Leftrightarrow \neg\Box(c)(\{y \in X \mid \neg P(y)\})(x). \qquad (5)$$

Here we use $\neg$ for negation on truth values. These temporal operators satisfy the familiar properties from modal logic. Basically they say that $\Box$ is an interior operation: $\Box(P) \subseteq P$, $\Box(P) \subseteq \Box(\Box(P))$, and $P \subseteq Q \Rightarrow \Box(P) \subseteq \Box(Q)$. Invariants are its opens (*i.e.* fixed points).

In the remainder of this section we shall use $\Box$ and $\Diamond$ for binary tree spaces, as described in Figure 4 in the previous section (*i.e.* with mirror operation). There we have already seen what it means for a predicate to be an invariant for binary trees. With the above definitions we thus have $\Box$ and $\Diamond$ for these trees. We will first redescribe these operators concretely. Then we shall show how they can be used in specification and verification (without relying on the concrete description).

7.1 A concrete description of $\Box$ and $\Diamond$ for binary trees

In order to convince the reader that $\Box(P)$ holds for x if P holds for all future states of x, and that $\Diamond(P)$ holds for x if P holds for some future state of x, we first have to make explicit what 'future state' means for binary trees. In principle we can proceed to a successor state via left and right children, and via mirroring. A path to a future state of x can thus be described by a finite lists of elements from $\{0, 1, 2\}$, where 0 stands for left child, 1 for right child (if any), and 2 for mirror. But from the way that the children and mirror operations interact, as given by assertion 'children_mirror' in Figure 4, we can simplify such paths into certain normal forms, with at most one mirror operation, right at the beginning, and with paths consisting only of 0's and 1's. Therefore we first introduce the following auxiliary function $\mathsf{offspring}\colon X \times \{0,1\}^\star \to 1 + X$ producing the successor state $\mathsf{offspring}(x, \alpha)$ of a tree $x \in X$ along a path α, if

[8] More formally, $\Box(c)(P) = \mathsf{gfp}(\lambda Q \in \mathcal{P}X.\, P \wedge \mathrm{NextTime}(c)(Q))$, using NextTime from Definition 2 and the greatest fixed point operator gfp that will be introduced in the beginning of Section 8.

any.

$$
\begin{array}{rl}
\mathsf{offspring}(x, \langle\rangle) = & \kappa' x \\
\mathsf{offspring}(x, a \cdot \alpha) = & \mathsf{CASES}\ \mathsf{offspring}(x, \alpha)\ \mathsf{OF} \\
& \quad \kappa y \mapsto \kappa * \\
& \quad \kappa' z \mapsto \mathsf{CASES}\ \mathsf{children}(z)\ \mathsf{OF} \\
& \qquad\qquad \kappa u \mapsto \kappa * \\
& \qquad\qquad \kappa'(v_1, v_2) \mapsto \mathsf{IF}\ a = 0 \\
& \qquad\qquad\qquad\qquad\qquad \mathsf{THEN}\ \kappa' v_1 \\
& \qquad\qquad\qquad\qquad\qquad \mathsf{ELSE}\ \kappa' v_2 \\
& \qquad\quad \mathsf{ENDCASES} \\
& \mathsf{ENDCASES}
\end{array}
$$

First we establish some auxiliary results about this offspring function. The last point in the lemma below shows how offspring commutes with mirroring.

Lemma 1. *Let x, y be arbitrary tree spaces. Then*

1. *For all lists $\alpha, \beta \in \{0,1\}^\star$,*

$$
\begin{array}{rl}
\mathit{offspring}(x, \alpha \cdot \beta) = & \mathit{CASES}\ \mathit{offspring}(x, \beta)\ \mathit{OF} \\
& \quad \kappa z \mapsto \kappa * \\
& \quad \kappa' w \mapsto \mathit{offspring}(w, \alpha) \\
& \mathit{ENDCASES}
\end{array}
$$

   where $\cdot$ is concatenation.

2. *Successor trees along the same path of bisimilar trees are bisimilar again:*

$$
\begin{array}{l}
x \mathrel{\underline{\leftrightarrow}} y \Longrightarrow \forall \alpha \in \{0,1\}^\star.\ \mathit{CASES}\ \mathit{offspring}(x, \alpha)\ \mathit{OF} \\
\qquad\qquad \kappa z \mapsto \mathit{CASES}\ \mathit{offspring}(y, \alpha)\ \mathit{OF} \\
\qquad\qquad\qquad\quad \kappa u \mapsto \mathit{true} \\
\qquad\qquad\qquad\quad \kappa' v \mapsto \mathit{false} \\
\qquad\qquad\qquad \mathit{ENDCASES} \\
\qquad\qquad \kappa' w \mapsto \mathit{CASES}\ \mathit{offspring}(y, \alpha)\ \mathit{OF} \\
\qquad\qquad\qquad\quad \kappa u \mapsto \mathit{false} \\
\qquad\qquad\qquad\quad \kappa' v \mapsto w \mathrel{\underline{\leftrightarrow}} v \\
\qquad\qquad\qquad \mathit{ENDCASES} \\
\qquad\quad \mathit{ENDCASES}
\end{array}
$$

3. *Let $\overline{\alpha} \in \{0,1\}^\star$ be obtained from $\alpha \in \{0,1\}^\star$ by changing 0's in 1's and vice-versa. Then:*

$$
\begin{array}{l}
\forall \alpha \in \{0,1\}^\star.\ \mathit{CASES}\ \mathit{offspring}(x, \alpha)\ \mathit{OF} \\
\qquad \kappa z \mapsto \mathit{CASES}\ \mathit{offspring}(\mathit{mirror}(x), \overline{\alpha})\ \mathit{OF} \\
\qquad\qquad\quad \kappa u \mapsto \mathit{true} \\
\qquad\qquad\quad \kappa' v \mapsto \mathit{false} \\
\qquad\qquad \mathit{ENDCASES} \\
\qquad \kappa' w \mapsto \mathit{CASES}\ \mathit{offspring}(\mathit{mirror}(x), \overline{\alpha})\ \mathit{OF} \\
\qquad\qquad\quad \kappa u \mapsto \mathit{false} \\
\qquad\qquad\quad \kappa' v \mapsto \mathit{mirror}(w) \mathrel{\underline{\leftrightarrow}} v \\
\qquad\qquad \mathit{ENDCASES} \\
\qquad \mathit{ENDCASES}
\end{array}
$$

The CASES notation gives a precise way of stating, for example in the last point, that if $\mathsf{offspring}(x,\alpha) = \kappa' w$, then $\mathsf{offspring}(\mathsf{mirror}(x), \overline{\alpha})$ is of the form $\kappa' v$, and this v is bisimilar to $\mathsf{mirror}(w)$.

Proof. By induction on α (in all three cases). □

Proposition 2. *Let $P \subseteq X$ be a predicate on a binary tree space X, which respects bisimilarity, i.e. which satisfies $x \leftrightarrow y \wedge P(x) \Rightarrow P(y)$, for all $x, y \in X$. Then:*

1. $\Box(P)(x) \iff$
$$\forall \alpha \in \{0,1\}^{\star}.\ \begin{pmatrix} \textsf{CASES offspring}(x,\alpha)\ \textsf{OF} \\ \quad \kappa z \mapsto \textsf{true} \\ \quad \kappa' w \mapsto P(w) \\ \textsf{ENDCASES} \end{pmatrix} \wedge \begin{pmatrix} \textsf{CASES offspring}(\textsf{mirror}(x),\alpha)\ \textsf{OF} \\ \quad \kappa z \mapsto \textsf{true} \\ \quad \kappa' w \mapsto P(w) \\ \textsf{ENDCASES} \end{pmatrix}$$
2. $\Diamond(P)(x) \iff$
$$\exists \alpha \in \{0,1\}^{\star}.\ \begin{pmatrix} \textsf{CASES offspring}(x,\alpha)\ \textsf{OF} \\ \quad \kappa z \mapsto \textsf{false} \\ \quad \kappa' w \mapsto P(w) \\ \textsf{ENDCASES} \end{pmatrix} \vee \begin{pmatrix} \textsf{CASES offspring}(\textsf{mirror}(x),\alpha)\ \textsf{OF} \\ \quad \kappa z \mapsto \textsf{false} \\ \quad \kappa' w \mapsto P(w) \\ \textsf{ENDCASES} \end{pmatrix}$$

Proof. The second point follows directly from the first one, since $\Diamond = \neg\Box\neg$, and so we concentrate on 1.

($\Rightarrow$) Assume $\Box(P)(x)$. Then there is an invariant $Q \subseteq X$ with $Q \subseteq P$ and $Q(x)$. The result follows from the statement:

$$\forall \alpha \in \{0,1\}^{\star}.\ \begin{pmatrix} \textsf{CASES offspring}(x,\alpha)\ \textsf{OF} \\ \quad \kappa z \mapsto \textsf{true} \\ \quad \kappa' w \mapsto Q(w) \\ \textsf{ENDCASES} \end{pmatrix} \wedge \begin{pmatrix} \textsf{CASES offspring}(\textsf{mirror}(x),\alpha)\ \textsf{OF} \\ \quad \kappa z \mapsto \textsf{true} \\ \quad \kappa' w \mapsto \exists y \in X.\, y \leftrightarrow w \wedge Q(y) \\ \textsf{ENDCASES} \end{pmatrix}$$

which can be proved by induction on α.

($\Leftarrow$) We have to produce an invariant $Q \subseteq X$ with $Q \subseteq P$ and $Q(x)$. We can take $Q(y)$ as:

$$\forall \alpha \in \{0,1\}^{\star}.\ \begin{pmatrix} \textsf{CASES offspring}(y,\alpha)\ \textsf{OF} \\ \quad \kappa z \mapsto \textsf{true} \\ \quad \kappa' w \mapsto P(w) \\ \textsf{ENDCASES} \end{pmatrix} \wedge \begin{pmatrix} \textsf{CASES offspring}(\textsf{mirror}(y),\alpha)\ \textsf{OF} \\ \quad \kappa z \mapsto \textsf{true} \\ \quad \kappa' w \mapsto P(w) \\ \textsf{ENDCASES} \end{pmatrix}$$

Then $Q(x)$ holds by assumption; $Q \subseteq P$ follows easily by taking $\alpha = \langle\rangle$, but invariance is more difficult. It requires Lemma 1. □

7.2 Using $\Box$ and $\Diamond$ for specification and verification of binary trees

Now that we have seen how $\Box$ and $\Diamond$ (for binary trees) indeed express "for all future states" and "for some future state", we start using these temporal operators in specifications. Subsequently we show how to reason with them (without relying on the concrete descriptions of Proposition 2).

BEGIN BinaryTreeSpace[A: TYPE]
  METHODS
    $\mathsf{label}\colon X \longrightarrow A$
    $\mathsf{children}\colon X \longrightarrow 1 + (X \times X)$
    $\mathsf{size}\colon X \longrightarrow 1 + \mathbb{N}$
    $\mathsf{mirror}\colon X \longrightarrow X$
    $\mathsf{elem}\colon X \times A \longrightarrow \{\mathsf{false}, \mathsf{true}\}$
  ASSERTIONS
    size_def: $\Big[$ see Figure 3 $\Big]$
    label_mirror: $\Big[$ see Figure 4 $\Big]$
    children_mirror: $\Big[$ see Figure 4 $\Big]$
    elem_def: $\Big[\ \forall x \in X.\, \forall a \in A.\, \mathsf{elem}(x, a) = \Diamond(\{y \in X \mid \mathsf{label}(y) = a\})(x)\ \Big]$
  CONSTRUCTORS
    $\mathsf{leaf}\colon A \longrightarrow X$
    $\mathsf{fill}\colon A \longrightarrow X$
  CREATION
    leaf_def: $\Big[\ \forall a \in A.\, \mathsf{label}(\mathsf{leaf}(a)) = a \wedge \mathsf{children}(\mathsf{leaf}(a)) = \kappa\, *\ \Big]$
    fill_def: $\Big[\ \forall a \in A.\, \Box(\{y \in X \mid \mathsf{label}(y) = a \wedge \mathsf{children}(y) \neq \kappa *\})(\mathsf{fill}(a))\ \Big]$
END BinaryTreeSpace

Fig. 5. Specification of binary trees, version 3.

Recall from Subsection 4.1 that the naive assertion for a membership method $\mathsf{elem}\colon X \times A \longrightarrow \{\mathsf{false}, \mathsf{true}\}$ does not work. We are now in a position to express $\mathsf{elem}(x, a)$ as "in some future state y of x, $\mathsf{label}(y) = a$", namely as $\Diamond(\{y \in X \mid \mathsf{label}(y) = a\})(x)$. This is incorporated in the next version of the binary tree space specification in Figure 5.

Of course one can also define $\mathsf{elem}(x, a)$ via the auxiliary function offspring from the previous subsection. But such a definition is too concrete, too verbose, and not at the right level of abstraction. For instance, if we were using ternary trees, with a children operation $X \to 1 + (X \times X \times X)$, instead of binary trees, then the eventually assertion for elem would still work, because $\Diamond$ abstracts from the particular tree structure.

New in this specification are the constructors. These are functions of the form $I \to X$, giving initial states, parametrised by the set I. These initial states are required to satisfy the assertions listed under 'CREATION'. In this case we have two[9] constructors, namely $\mathsf{leaf}\colon A \to X$ and $\mathsf{fill}\colon A \to X$. The intention is that $\mathsf{leaf}(a)$ is the tree with a as label and with no children. This is easy to express, see the assertion 'leaf_def' in Figure 5. The other constructor yields a tree $\mathsf{fill}(a)$ that has a not only as direct label but also as label for all of its successors, which are all required to exist. This can be expressed conveniently via a $\Box$ formula.

We give an indication of what can be proved about this latest specification in two lemmas. The next section elaborates further on (in)finiteness of behaviour of binary trees.

Lemma 2. *Consider the binary tree specification from Figure 5. The mirror of a tree $x \in X$ has the same elements as x, i.e. for all $a \in A$,*

$$\mathit{elem}(x, a) \Longleftrightarrow \mathit{elem}(\mathit{mirror}(x), a).$$

With this result one can show that mirroring is its own inverse, for binary tree spaces as in Figure 5 (like in Figure 4, see Proposition 1 (2)).

Proof. What we have to prove is, for an arbitrary $x \in X$ and $a \in A$:

$$\begin{aligned}
&\exists I \subseteq X.\,(I \text{ is an invariant}) \wedge (I \subseteq \{y \in X \mid \mathsf{label}(y) \neq a\}) \wedge (I(\mathsf{mirror}(x))\\
&\Longleftrightarrow\\
&\exists I \subseteq X.\,(I \text{ is an invariant}) \wedge (I \subseteq \{y \in X \mid \mathsf{label}(y) \neq a\}) \wedge I(x).
\end{aligned}$$

For the direction ($\Leftarrow$) one can use the same predicate I, but for ($\Rightarrow$) one has to do some work: assume I as in the assumption, and define $I' \subseteq X$ as:

$$I'(x) \Leftrightarrow \exists y \in X.\, y \,\underline{\leftrightarrow}\, \mathsf{mirror}(x) \wedge I(y)$$

Then I' is the required predicate. $\Box$

A next exercise that we set ourselves is to prove that the size of an initial state $\mathsf{fill}(a)$, for $a \in A$, is infinite (*i.e.* is $\kappa *$). We do so by first proving a more general statement.

Lemma 3. *Still in the context of the specification in Figure 5,*

1. $\Box(\{y \in X \mid \mathit{children}(y) \neq \kappa *\})(x) \Rightarrow \mathit{size}(x) = \kappa *$, *for all* $x \in X$.
2. $\mathit{size}(\mathit{fill}(a)) = \kappa *$, *for all* $a \in A$.

Proof. The second point follows immediately from the first one using the definition of $\mathsf{fill}(a)$. For 1 we first prove an auxiliary statement. Assume $\Box(\{y \in X \mid \mathsf{children}(y) \neq \kappa *\})(x)$. Then, for all $n \in \mathbb{N}$,

$$(\mathsf{size}(x) = \kappa' n) \Longrightarrow \forall m \in \mathbb{N}.\, \Diamond(\{y \in X \mid \begin{array}[t]{l}\mathsf{CASES}\ \mathsf{size}(y)\ \mathsf{OF}\\ \quad \kappa u \mapsto \mathsf{false}\\ \quad \kappa' v \mapsto v + m \leq n\\ \mathsf{ENDCASES}\,\})(x)\end{array}$$

In words: if the size of x is $\kappa' n$, then for all $m \in \mathbb{N}$ there is a future state y of x whose size is $\kappa' v$ and satisfies $v + m \leq n$. The conclusion of this statement follows by induction on m. This shows that $\mathsf{size}(x)$ cannot be of the form $\kappa' n$—take $m = n + 1$ in that case—and thus that $\mathsf{size}(x)$ must be $\kappa *$. $\Box$

[9] These two constructors can be combined into a single one with type $A + A \to X$.

Exercises

1. Prove that $\Box(\bigcap_{i\in I} P_i) = \bigcap_{i\in I} \Box(P_i)$.
2. Prove that the predicate I' in the proof of Lemma 2 is an invariant.
3. Prove the following "induction rule of temporal logic":

$$P \cap \Box(P \supset \mathsf{NextTime}(P)) \subseteq \Box(P),$$

where $P \supset Q = \{x \mid P(x) \text{ implies } Q(x)\}$.
4. Consider the specification in Figure 5, and prove, for all $a, b \in A$,

$$\mathsf{elem}(\mathsf{fill}(a), b) \Longrightarrow a = b.$$

8 Towards a μ-calculus for coalgebras

We continue the study of binary trees as in Figure 5. In the first point in Lemma 3 in the previous section we have seen a certain property, namely $\Box(\{y \in X \mid \mathsf{children}(y) \neq \kappa *\})(x)$, guaranteeing that $\mathsf{size}(x)$ is infinite. At this stage we wonder: are there formulas expressing finiteness of behaviour, and also infiniteness behaviour, in such a way that a tree has finite behaviour if and only if its size is finite (*i.e.* of the form $\kappa' n$, for some $n \in \mathbb{N}$), and has infinite behaviour if and only if its size is infinite (*i.e.* $\kappa *$)?

In this section we give a positive answer to this question. In order to express such formulas, say $\mathsf{finite}(x)$ and $\mathsf{infinite}(x)$, for a binary tree x, we shall make use of least and greatest fixed points, via operators μ and ν as are standard in what is called the μ-calculus (see [36]). In the μ-calculus one allows μ and ν in the construction of logical formulas, with associated rules for reasoning. Here we shall only illustrate the usefulness of μ and ν in our running example. We proceed at a semantical level, without developing a logic. So first we have to make clear what we mean by these fixed points.

Let Y be an arbitrary set. The set $\mathcal{P}(Y)$ of subsets of Y is partially ordered by inclusion $\subseteq$. A function $F\colon \mathcal{P}(Y) \to \mathcal{P}(Y)$ is called monotone if it respects inclusions: $P \subseteq Q \Rightarrow F(P) \subseteq F(Q)$. For such an F we define

$$\begin{aligned} \mathsf{lfp}(F) &= \bigcap\{U \subseteq Y \mid F(U) \subseteq U\} \\ \mathsf{gfp}(F) &= \bigcup\{U \subseteq Y \mid U \subseteq F(U)\} \end{aligned}$$

Then it is not hard to prove (using that F is monotone):

$$\begin{array}{ll} F(P) \subseteq P \Rightarrow \mathsf{lfp}(F) \subseteq P & \qquad F(\mathsf{lfp}(F)) = \mathsf{lfp}(F) \\ P \subseteq F(P) \Rightarrow P \subseteq \mathsf{gfp}(F) & \qquad F(\mathsf{gfp}(F)) = \mathsf{gfp}(F). \end{array}$$

This says that $\mathsf{lfp}(F)$ is the least fixed point, and $\mathsf{gfp}(F)$ the greatest fixed point of F. The μ-calculus offers a special syntax for the functions lfp and gfp, namely the binding operators μ and ν. Their meaning is:

$$\mu P.\, F(P) = \mathsf{lfp}(F) \qquad \text{and} \qquad \nu P.\, F(P) = \mathsf{gfp}(F).$$

We return to binary tree spaces, and define for a subset $P \subseteq X$, two predicates:

$$
\begin{aligned}
\mathsf{Fin}(P) &= \{y \in X \mid \mathsf{Pred}(1 + (X \times X))(P)(\mathsf{children}(y))\} \\
&= \{y \in X \mid \mathsf{CASES}\ \mathsf{children}(y)\ \mathsf{OF} \\
&\qquad\qquad \kappa u \mapsto \mathsf{true} \\
&\qquad\qquad \kappa'(v_1, v_2) \mapsto P(v_1) \wedge P(v_2) \\
&\qquad\quad \mathsf{ENDCASES}\,\} \\
\mathsf{Inf}(P) &= \{y \in X \mid \neg\mathsf{Pred}(1 + (X \times X))(\neg P)(\mathsf{children}(y))\} \\
&= \{y \in X \mid \mathsf{CASES}\ \mathsf{children}(y)\ \mathsf{OF} \\
&\qquad\qquad \kappa u \mapsto \mathsf{false} \\
&\qquad\qquad \kappa'(v_1, v_2) \mapsto P(v_1) \vee P(v_2) \\
&\qquad\quad \mathsf{ENDCASES}\,\}
\end{aligned}
$$

It is not hard to see that both Fin and Inf are monotone functions[10]. Therefore we can define the following two predicates on X.

$$\mathsf{finite} = \mathsf{lfp}(\mathsf{Fin}) \qquad \text{and} \qquad \mathsf{infinite} = \mathsf{gfp}(\mathsf{Inf})$$

In the μ-calculus one would directly define:

$$
\begin{aligned}
\mathsf{finite} &= \mu P.\, \{y \in X \mid \mathsf{CASES}\ \mathsf{children}(y)\ \mathsf{OF} \\
&\qquad\qquad \kappa u \mapsto \mathsf{true} \\
&\qquad\qquad \kappa'(v_1, v_2) \mapsto P(v_1) \wedge P(v_2) \\
&\qquad\quad \mathsf{ENDCASES}\,\} \\
\mathsf{infinite} &= \nu P.\, \{y \in X \mid \mathsf{CASES}\ \mathsf{children}(y)\ \mathsf{OF} \\
&\qquad\qquad \kappa u \mapsto \mathsf{false} \\
&\qquad\qquad \kappa'(v_1, v_2) \mapsto P(v_1) \vee P(v_2) \\
&\qquad\quad \mathsf{ENDCASES}\,\}
\end{aligned}
$$

Such definitions in μ-calculus style are usually very compact (and elegant), but not so easy to read and understand. In this case we shall first relate the definitions of the predicates finite and infinite to the concrete description in terms of the offspring function using paths (introduced in Subsection 7.1). Subsequently we shall relate these predicates to the size function.

In order to familiarise the reader with reasoning using least and greatest fixed points, we start with the following elementary observations.

Lemma 4. *For binary trees $x, y \in X$ satisfying the specification from Figure 5 we have:*

1. *If $x \,\underline{\leftrightarrow}\, y$, then*

$$\mathit{finite}(x) \Longleftrightarrow \mathit{finite}(y) \quad \text{and} \quad \mathit{infinite}(x) \Longleftrightarrow \mathit{infinite}(y).$$

[10] The occurrence of predicate lifting in this definitions is not accidental. For an arbitrary coalgebra $c\colon X \to T(X)$ of a polynomial functor, one can define the subset of states with finite behaviour as $\mu P.\, \mathsf{NextTime}(c)(P) = \mathsf{lfp}(\mathsf{NextTime}(c))$, and the subset of states with infinite behaviour as $\nu P.\, \neg\mathsf{NextTime}(c)(\neg P)$.

2. Also:

$$\mathit{finite}(\mathit{mirror}(x)) \Longleftrightarrow \mathit{finite}(x) \quad \mathit{and} \quad \mathit{infinite}(\mathit{mirror}(x)) \Longleftrightarrow \mathit{infinite}(x).$$

Proof. 1. We shall prove the first part about finiteness. So assume $x \underline{\leftrightarrow} y$ and $\mathsf{finite}(x)$, *i.e.* $\mathsf{lfp}(\mathsf{Fin})(x)$. Because $\mathsf{lfp}(\mathsf{Fin})$ is the least fixed point of Fin, it suffices to find a predicate P with $\mathsf{Fin}(P) \subseteq P$ and $P(x) \Rightarrow \mathsf{finite}(y)$. These requirements suggest $P = \{z \in X \mid \forall w \in X.\, z \underline{\leftrightarrow} w \Rightarrow \mathsf{finite}(w)\}$.

2. We do the infinite case.
   ($\Rightarrow$) Assume $\mathsf{infinite}(\mathsf{mirror}(x))$, *i.e.* $\mathsf{gfp}(\mathsf{Inf})(\mathsf{mirror}(x))$. In order to prove $\mathsf{infinite}(x)$ we simply take $P = \{y \in X \mid \mathsf{infinite}(\mathsf{mirror}(y))\}$. The inclusion $P \subseteq \mathsf{Inf}(P)$ follows from 1, and $P(x)$ is obvious. Therefore $\mathsf{infinite}(x)$.
   ($\Leftarrow$) Assume $\mathsf{infinite}(x)$. In order to prove $\mathsf{gfp}(\mathsf{Inf})(\mathsf{mirror}(x))$ it suffices to produce a predicate P with $P \subseteq \mathsf{Inf}(P)$ and $P(\mathsf{mirror}(x))$. We take $P = \{y \in X \mid \exists z \in X.\, \mathsf{mirror}(z) \underline{\leftrightarrow} y \wedge \mathsf{infinite}(z)\}$. □

Proposition 3. *Finiteness and infiniteness of behaviour for binary trees can be expressed in terms of the* offspring *function from Subsection 7.1 as follows.*

1. $\mathit{finite}(x) \Longleftrightarrow \exists n \in \mathbb{N}.\, \forall \alpha \in \{0,1\}^\star.\, |\alpha| \geq n \Rightarrow \mathit{offspring}(x,\alpha) = \kappa *.$
2. $\mathit{infinite}(x) \Longleftrightarrow \forall n \in \mathbb{N}.\, \exists \alpha \in \{0,1\}^\star.\, |\alpha| = n \wedge \mathit{offspring}(x,\alpha) \neq \kappa *.$

Proof. 1. For the direction ($\Rightarrow$) one uses that finiteness is defined as a least fixed point. The predicate P with $\mathsf{Fin}(P) \subseteq P$ that one can use is given by $P = \{y \in X \mid \exists n \in \mathbb{N}.\, \forall \alpha \in \{0,1\}^\star.\, |\alpha| \geq n \Rightarrow \mathsf{offspring}(y,\alpha) = \kappa *\}$.
   For the reverse direction ($\Leftarrow$) assume, for some $n \in \mathbb{N}$ that $\mathsf{offspring}(x,\alpha) = \kappa *$ for all $\alpha \in \{0,1\}^\star$ with $|\alpha| \geq n$. Then one can prove, by induction on m, that:

$$\begin{array}{l} \forall m \leq n.\, \forall \alpha \in \{0,1\}^\star.\, \mathsf{CASES}\ \mathsf{offspring}(x,\alpha)\ \mathsf{OF} \\ \qquad\qquad\qquad \kappa u \mapsto \mathsf{true} \\ \qquad\qquad\qquad \kappa' v \mapsto |\alpha| \geq n - m \Rightarrow \mathsf{finite}(v) \\ \qquad\qquad\quad \mathsf{ENDCASES} \end{array}$$

   But then we are done by taking $m = n$ and $\alpha = \langle\rangle$.

2. The implication ($\Leftarrow$) can be proved using that infinity is introduced via a greatest fixed point. An appropriate predicate P with $P \subseteq \mathsf{Inf}(P)$ is: $P = \{y \in X \mid \forall n \in \mathbb{N}.\, \exists \alpha \in \{0,1\}^\star.\, |\alpha| = n \wedge \mathsf{offspring}(y,\alpha) \neq \kappa *\}$.
   For the other implication ($\Rightarrow$), assume $\mathsf{infinite}(x)$. By induction on n we can prove a slightly stronger statement:

$$\begin{array}{l} \forall n \in \mathbb{N}.\, \exists \alpha \in \{0,1\}^\star.\, \mathsf{CASES}\ \mathsf{offspring}(x,\alpha)\ \mathsf{OF} \\ \qquad\qquad\qquad \kappa u \mapsto \mathsf{false} \\ \qquad\qquad\qquad \kappa' v \mapsto |\alpha| = n \wedge \mathsf{infinite}(v) \\ \qquad\qquad\quad \mathsf{ENDCASES} \end{array}$$

□

In a next step we show that both finiteness and infinity can be expressed in terms of size.

Proposition 4. *For a binary tree space x as in Figure 5 we have*

1. $\mathsf{finite}(x) \Longleftrightarrow \mathsf{size}(x) \neq \kappa *$.
2. $\mathsf{infinite}(x) \Longleftrightarrow \mathsf{size}(x) = \kappa *$.

Proof. 1. The direction ($\Rightarrow$) follows from finite being a least fixed point. For ($\Leftarrow$) we prove by well-founded induction (on the natural numbers) that:

$$\forall n \in \mathbb{N}.\, \mathsf{size}(x) = \kappa' n \Rightarrow \mathsf{finite}(x).$$

2. The greatest fixed point property of $\mathsf{infinite}$ takes care of the implication ($\Leftarrow$). For ($\Rightarrow$), assume $\mathsf{infinite}(x)$ and $\mathsf{size}(x) = \kappa' n$. The statement

$$\begin{array}{rl}\forall m \in \mathbb{N}.\, \Diamond(\{y \in X \mid \mathsf{infinite}(y) \wedge & \mathsf{CASES}\ \mathsf{size}(y)\ \mathsf{OF} \\ & \quad \kappa u \mapsto \mathsf{false} \\ & \quad \kappa' v \mapsto n \geq v + m \\ & \mathsf{ENDCASES}\})(x)\end{array}$$

can be proved by induction on m. But then we get a contradiction by taking $m = n + 1$. Hence $\mathsf{size}(x) = \kappa *$. $\square$

This concludes our brief tour of a μ-calculus in a coalgebraic setting. It also concludes our discussion of binary trees. We only mentioned a few operations that can be used for binary trees, leaving out many others. Typically, one also has operations for inserting and deleting labels, and for extracting (possibly infinite) sequences of labels, using for example pre-order, in-order, or post-order tree traversal. Insertion and deletion are usually defined for so-called *binary search trees* where all labels appear in an ordered fashion (assuming the set A of labels is totally ordered): labels in a left child are below the current label, which is below labels in the right child. Such properties typically appear as invariants. Here we did not include these additional operations in our discussion because our sole aim is to illustrate coalgebraic specification and verification techniques (especially using temporal and fixed point operators), and not to be in any sense complete[11].

Exercises

1. Prove that $\Box(P) = \mathsf{gfp}(\lambda Q.\, P \cap \mathsf{NextTime}(Q))$.

9 A case study: Peterson's mutual exclusion algorithm

In this section we present a non-entirely-trivial application in coalgebraic specification and verification: Peterson's mutual exclusion algorithm. It provides a well-known mechanism for regulated access to what are called critical regions (or

[11] Also, the way we have set up our binary trees is probably not optimal for storage, because, as is not hard to see, a finite binary tree will always have an odd size. Hence adding single elements is problematic.

critical sections): often in computer sciences there situations with two (or more) processes (running programs) which share certain resources, such as computing devices or data in memory (or in files or on disks). In such cases it is important that access is well-regulated, so that no corruption can take place. This issue called the mutual exclusion problem. What one needs is some mechanism which grants access to a critical region only to one process at a time. This should be done in such a way that all requests for access will be granted at some stage, in a reasonably fair manner. This issue is often discussed at length in books on operating systems, like [37].

9.1 Peterson's solution for mutual exclusion

A particularly nice and easy way to achieve mutual exclusion was developed by Peterson [29], see also [37, 2.2.3], [5, 7.2], [1, 6.5] or [24, 10.5.1]. The latter three references also contain correctness proofs: in [5] the algorithm is presented as a parallel composition of two automata, and its correctness is proved in temporal logic; in [1] the algorithm is described as a parallel composition of two programs, whose (safety only) correctness is established using (the Owicki-Gries extension of) Hoare logic; and in [25] state machines are used with a precondition-effect notation. Here we shall describe the algorithm in coalgebraic specification, making use of structuring techniques in object-oriented style. We shall return to this example in the next section on refinement.

A first, intuitive impression of Peterson's algorithm can be obtained from the automata theoretic description used in [5]. It consists of two essentially identical automata, put in parallel, see Figure 6. They both have a critical section, indicated by the locations ℓ_3 and m_3 with double circles. They share a variable t, which indicates whose turn it is (to proceed to the critical section), with true for automaton 1 on the left, false for automaton 2 on the right. Both automata can read and change the value of t. Reading is indicated by the question mark '?', and writing by the assignment symbol ':='. There are two additional variables y_i, for $i \in \{1,2\}$, indicating whether automaton i is interested in becoming critical. Automaton i can read both y_1 and y_2, but change only y_i. Through a subtle interaction of these variables t, y_1, y_2 it is prevented that the automata are both in their critical sections (*i.e.* at ℓ_3 and m_3 at the same time).

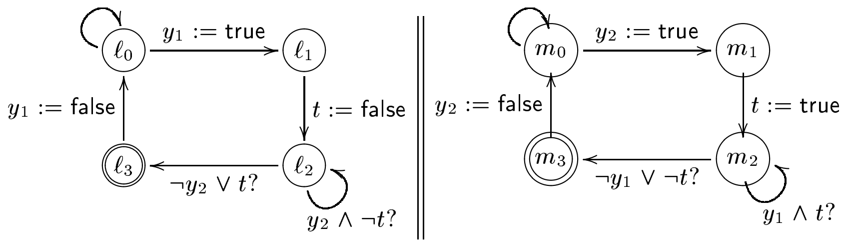


Fig. 6. Peterson's algorithm, described as composition of two parallel automata (from [5])

Of course, this statement needs a rigorous proof. First of all, such a proof requires a precise formalisation of the algorithm. As such, the automata theoretic description from Figure 6 is not ideal. For example, it is somehow implicit that transitions $\ell_1 \to \ell_2$ and $m_1 \to m_2$ cannot happen at the same time, because they involve an incompatible assignment to t. Also, in the transitions $\ell_2 \to \ell_3$, $m_2 \to m_3$ the turn variable t does not change. Such facts are crucial in verification[12]. Therefore we seek an alternative, purely assertional formalisation, in which all these points will be written explicitly. Necessarily, such a description will be much more verbose. It will be given as a coalgebraic specification below (in Figure 13). Before presenting it, we have to deal with some new aspects that are used in this specification.

9.2 Dealing with time in coalgebraic specification

We shall briefly discuss a way of handling time in a coalgebraic setting, following [16]. We concentrate on discrete time, as this will be used in the specification of Peterson's algorithm. But continuous time will also be mentioned.

BEGIN DiscreteTime
  METHODS
    $\mathsf{tick}\colon X \longrightarrow X$
END DiscreteTime

Fig. 7. Discrete time

Discrete time can be modelled via a single operation $\mathsf{tick}\colon X \to X$, see Figure 7. The idea is that every unit of time, this function is called. How this happens is not relevant, but one can think of it as resulting from the clock of a computer system. So for a state $x \in X$, the state after two units of time is $\mathsf{tick}(\mathsf{tick}(x))$. It is convenient to have suitable notation, like tick^n, for iteration, where:

$$\mathsf{tick}^0(x) = x \qquad \text{and} \qquad \mathsf{tick}^{n+1}(x) = \mathsf{tick}(\mathsf{tick}^n(x)).$$

Figure 8 presents a timer which can be set by the user, and then goes off "automatically" after N units of time (given as parameter). We have used **INHERIT FROM** to indicate that this specification also has a method $\mathsf{tick}\colon X \to X$. The polynomial functor $\mathcal{T}$ underlying the Timer specification then contains the polynomial functor $\mathcal{DT}$ of the DiscreteTime specification as $\times$-component[13]:

$$\mathcal{T}(X) = \mathcal{DT}(X) \times X \times \{\mathsf{on}, \mathsf{off}\} = X \times X \times \{\mathsf{on}, \mathsf{off}\}.$$

[12] All these aspects can be made precise in an automaton theoretic framework, by defining appropriate notions of behaviour and composition. But doing so is not our aim. The description in Figure 6 only serves as a first introduction.

[13] And not as +-component, like in algebraic specification, see the functor for the vector space example (2).

$$
\begin{array}{|l|}
\hline
\textbf{BEGIN}\ \mathrm{Timer}[N\colon \mathbb{N}] \\
\quad \textbf{INHERIT FROM}\ \mathrm{DiscreteTime} \quad //\ \text{yields}\ \mathsf{tick} \\
\quad \textbf{METHODS} \\
\quad\quad \mathsf{set}\colon X \longrightarrow X \\
\quad\quad \mathsf{status}\colon X \longrightarrow \{\mathsf{on}, \mathsf{off}\} \\
\quad \textbf{ASSERTIONS} \\
\quad\quad \text{off\_remains:} \left[\ \forall x \in X.\, \forall n \in \mathbb{N}.\ \mathsf{status}(x) = \mathsf{off} \Rightarrow \mathsf{status}(\mathsf{tick}^n(x)) = \mathsf{off}\ \right] \\
\quad\quad \text{off\_happens:} \left[\ \forall x \in X.\, \forall n \in \mathbb{N}.\ n \geq N \Rightarrow \mathsf{status}(\mathsf{tick}^n(x)) = \mathsf{off}\ \right] \\
\quad\quad \text{status\_set:} \left[\ \forall x \in X.\, \forall n \in \mathbb{N}.\ n < N \Rightarrow \mathsf{status}(\mathsf{tick}^n(\mathsf{set}(x))) = \mathsf{on}\ \right] \\
\textbf{END}\ \mathrm{Timer} \\
\hline
\end{array}
$$

Fig. 8. A simple parametrised timer

The assertion 'off_remains' tells that once the timer is off, it does not "spontaneously" become on by passage of time. This could be expressed via temporal operators—see below—but here we choose to write it explicitly via iteration. The next assertion 'off_happens' expresses that no matter in which state the timer is, it will be off after at least N units of time. Finally, the first N ticks after an invocation of set the timer will be on, as expressed by the third assertion 'status_set'.

This specification has many models. For example, one can take for X the set $[0, N] \subseteq \mathbb{N}$ with functions $\mathsf{tick}, \mathsf{set}\colon [0, N] \to [0, N]$ and $\mathsf{status}\colon [0, N] \to \{\mathsf{on}, \mathsf{off}\}$ given by on $x \in [0, N]$ as:

$$
\mathsf{tick}(x) = \begin{cases} N & \text{if } x \geq N-1 \\ x+1 & \text{else} \end{cases} \qquad \mathsf{set}(x) = 0 \quad \mathsf{status}(x) = \begin{cases} \mathsf{on} \text{ if } x < N \\ \mathsf{off} \text{ if } x = N \end{cases}
$$

A state $x \in [0, N]$ thus represents the number of time units until the timer's status will be off.

The tick function in the specification of discrete time describes the passage to a next state through the passage of time. It can thus be used to described primed versions of attributes: for a method (or attribute) $a\colon X \to A$, one sometimes sees in the computer science literature the notation $a'\colon X \to A$ for "the attribute a evaluated in the next state". Thus we can understand a' as $a \circ \mathsf{tick}$. In such a way one can translate specifications in the Temporal Logic of Actions (TLA), see [22, 23], into coalgebraic specifications. Actions in TLA are predicates, which become methods $X \to \mathsf{bool}$. They describe the preconditions and postconditions in a single conjunction, such as: $(a(x) > 0) \wedge (a'(x) = a(x) - 1)$. The temporal logic of TLA is linear temporal logic, and its $\Box$ and $\Diamond$ operators are the ones that are associated with the DiscreteTime specification from Figure 7 (following Section 7):

$$
\begin{aligned}
\Box(P)(x) &\iff \forall n \in \mathbb{N}.\, P(\mathsf{tick}^n(x)) \\
\Diamond(P)(x) &\iff \exists n \in \mathbb{N}.\, P(\mathsf{tick}^n(x)).
\end{aligned} \tag{6}
$$

Because $\{y \in X \mid \forall n \in \mathbb{N}. P(\mathsf{tick}^n(y))\}$ is the greatest DiscreteTime-invariant contained in P.

BEGIN ContinuousTime
METHODS
$\mathsf{flow}\colon X \times \mathbb{R}_{\geq 0} \longrightarrow X \quad // \, \mathbb{R}_{\geq 0} = \{s \in \mathbb{R} \mid s \geq 0\}$
ASSERTIONS
flow_zero: $\left[\ \forall x \in X.\, \mathsf{flow}(x, 0) = x\ \right]$
flow_plus: $\left[\ \forall x \in X.\, \forall s, t \in \mathbb{R}_{\geq 0}.\, \mathsf{flow}(x, s + t) = \mathsf{flow}(\mathsf{flow}(x, t), s)\ \right]$
END ContinuousTime

Fig. 9. Continuous time

Notice that iteration yields a function $\mathsf{ticks}\colon X \times \mathbb{N} \to X$, namely $\mathsf{ticks}(x, n) = \mathsf{tick}^n(x)$. This forms an *action* for the natural number monoid $(\mathbb{N}, +, 0)$, since

$$\mathsf{ticks}(x, 0) = x \qquad \text{and} \qquad \mathsf{ticks}(x, n + m) = \mathsf{ticks}(\mathsf{ticks}(x, m), n).$$

This action aspect is taken as fundamental in handling continuous time, see Figure 9. There we have an action $\mathsf{flow}\colon X \times \mathbb{R}_{\geq 0} \to X$ with respect to the monoid $(\mathbb{R}_{\geq 0}, +, 0)$ of positive real numbers. Such flows are fundamental in system theory (see *e.g.* [21]) and are also known as motions, trajectories or solutions. Indeed, unique solutions to differential equations give such flows, see *e.g.* [9, 8.7]. The temporal operators associated with the ContinuousTime specification of Figure 9 are:

$$\begin{aligned} \Box(P)(x) &\iff \forall s \in \mathbb{R}_{\geq 0}.\, P(\mathsf{flow}(x, s)) \\ \Diamond(P)(x) &\iff \exists s \in \mathbb{R}_{\geq 0}.\, P(\mathsf{flow}(x, s)). \end{aligned}$$

Since the predicate $\{y \in X \mid \forall s \in \mathbb{R}_{\geq 0}.\, P(\mathsf{flow}(y, s))\}$ is the greatest Continuous-Time-invariant contained in P. We shall not use these flows in this paper. For more information, see [16].

9.3 Class-valued methods

So far we have seen methods that return a value, like $\mathsf{size} \to 1+\mathbb{N}$ or $\mathsf{label}\colon X \to A$ in Figure 3. Such methods simply describe functions. The question arises whether one can also have "class-valued" methods such as $\mathsf{tree}\colon X \to \mathsf{BinaryTreeSpace}[\mathbb{N}]$ in coalgebraic specifications, yielding values in classes (also called objects). For each state $x \in X$, $\mathsf{tree}(x)$ should then be a binary tree with natural numbers as labels. Such methods are convenient because they allow us to incrementally build structured specifications from smaller components (*i.e.* classes).

The question is: what kind of functions are such class-valued methods like tree? They should return elements of a model[14] of the tree specification. But

[14] Even this condition can be relaxed: one can just require elements of the state space of a coalgebra providing the binary tree space methods, without requiring that the

which model? We have seen several models in Section 4, but there seems to be no canonical choice. Actually there is a canonical choice here, namely the so-called *terminal* (or *final*) model. It is the defined as the model F such that for an arbitrary model M there is a unique homomorphism $M \to F$, see [18] for more information. Terminality is useful when casting is required, *i.e.* when elements of classes which inherit from a class C must be regarded as elements of C. This can be done via the unique homomorphism just mentioned, see [11]. But this is beyond scope.

For many purposes, it really does not matter which model is used for class-valued attributes. In such a situation one can use an arbitrary model—assuming there is one[15]. Since the chosen model is arbitrary, there is nothing specific that can be used about it. This is called *loose* semantics. It is what we shall use below.

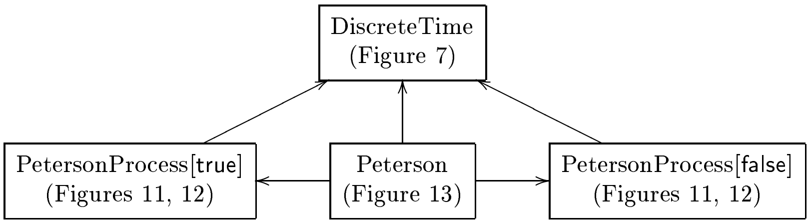


Fig. 10. The structure between specifications in Peterson's algorithm

9.4 Peterson's algorithm in coalgebraic specification

We can finally describe Peterson's mutual exclusion algorithm in coalgebraic specification. The specification as a whole will be built up from smaller specifications, see Figure 10. The central specification is called Peterson. The arrows going upwards point to superclasses (using inheritance), the horizontal ones to component classes (via class-valued methods). The PetersonProcess is parametrised by the set $\mathsf{bool} = \{\mathsf{false}, \mathsf{true}\}$, so that the two versions (like in Figure 6) can be described together.

We shall explain the main specifications PetersonProcess (Figures 11, 12) and Peterson (Figure 13) in some more detail. Basically, it follows the automata-theoretic description from Figure 6. In the PetersonProcess specification the real work is done. There are Boolean-valued attributes critical, for describing whether this process is in its critical section, own_interest, telling whether this process is interested in becoming critical (corresponding to y_1 or y_2 in Figure 6), other_interest, telling whether the other process is interested (corresponding to y_2 or y_1), turn, describing whose turn it is (like t does in Figure 6), and pre_critical,

coalgebra also satisfies the assertions. In such a way one can model "casting with late binding". But that is beyond the scope of the present paper.

[15] It is good practice to construct a model after introducing a new specification.

$$
\begin{array}{l}
\textbf{BEGIN}\ \text{PetersonProcess}[\,\text{ident}: \mathsf{bool}\,] \quad //\ \text{where}\ \mathsf{bool} = \{\mathsf{false}, \mathsf{true}\} \\
\quad \textbf{INHERIT FROM}\ \text{DiscreteTime} \quad //\ \text{see Figure 7} \\
\quad \textbf{METHODS} \\
\qquad \mathsf{critical}\colon X \longrightarrow \mathsf{bool} \\
\qquad \mathsf{own\_interest}\colon X \longrightarrow \mathsf{bool} \\
\qquad \mathsf{other\_interest}\colon X \longrightarrow \mathsf{bool} \quad //\ \text{describes interest of other process} \\
\qquad \mathsf{turn}\colon X \longrightarrow \mathsf{bool} \quad //\ \text{will be shared} \\
\qquad \mathsf{pre\_critical}\colon X \longrightarrow \mathsf{bool} \quad //\ \text{describes waiting state} \\
\qquad \mathsf{interest}\colon X \longrightarrow X \quad //\ \text{enables a user to indicate interest} \\
\qquad \qquad \qquad \qquad \quad //\ \text{to proceed to critical section} \\
\quad \textbf{ASSERTIONS} \\
\qquad \text{interest:}\ \big[\ \mathsf{own\_interest}(\mathsf{interest}(x)) \wedge \mathsf{turn}(\mathsf{interest}(x)) = \mathsf{turn}(x) \wedge \\
\qquad \qquad \mathsf{other\_interest}(\mathsf{interest}(x)) = \mathsf{other\_interest}(x) \wedge \\
\qquad \qquad \neg\mathsf{pre\_critical}(\mathsf{interest}(x)) \wedge \neg\mathsf{critical}(\mathsf{interest}(x))\ \big] \\
\qquad \text{remain\_not\_interested:}\ \big[\ \neg\mathsf{own\_interest}(x) \Rightarrow \big(\neg\mathsf{own\_interest}(\mathsf{tick}(x)) \wedge \\
\qquad \qquad \neg\mathsf{critical}(\mathsf{tick}(x)) \wedge \neg\mathsf{pre\_critical}(\mathsf{tick}(x))\big)\ \big] \\
\qquad \text{become\_pre\_critical:}\ \big[\ \big(\mathsf{own\_interest}(x) \wedge \neg\mathsf{pre\_critical}(x) \wedge \neg\mathsf{critical}(x)\big) \\
\qquad \qquad \Rightarrow \big(\mathsf{own\_interest}(\mathsf{tick}(x)) \wedge \mathsf{pre\_critical}(\mathsf{tick}(x)) \wedge \\
\qquad \qquad \neg\mathsf{critical}(\mathsf{tick}(x)) \wedge \\
\qquad \qquad //\ \mathsf{turn\ cannot\ remain\ ident}\text{:} \\
\qquad \qquad (\mathsf{turn}(x) = \mathsf{ident} \Rightarrow \mathsf{turn}(\mathsf{tick}(x)) \neq \mathsf{ident})\big)\ \big] \\
\qquad \text{remain\_pre\_critical:}\ \big[\ \big(\mathsf{own\_interest}(x) \wedge \mathsf{pre\_critical}(x) \wedge \neg\mathsf{critical}(x) \wedge \\
\qquad \qquad \mathsf{other\_interest}(x) \wedge \mathsf{turn}(x) \neq \mathsf{ident}\big) \\
\qquad \qquad \Rightarrow \big(\mathsf{own\_interest}(\mathsf{tick}(x)) \wedge \mathsf{pre\_critical}(\mathsf{tick}(x)) \wedge \\
\qquad \qquad \neg\mathsf{critical}(\mathsf{tick}(x))\big)\ \big] \\
\qquad \qquad \cdots \quad //\ \text{see Figure 12} \\
\textbf{END}\ \text{PetersonProcess}
\end{array}
$$

Fig. 11. Specification of a (parametrised) Peterson process, part I

indicating whether this process is in the waiting state before becoming critical (corresponding to locations ℓ_2 and m_2 in Figure 6). Finally, there is a method interest which can be used to indicate that a process is interested in becoming critical, as expressed by the assertion 'interest' in Figure 11. The other assertions tell how the system changes under the influence of time: they describe the effect of the tick method under various circumstances. These assertions precisely describe the behaviour, by listing the values of various attributes before and after a tick. One can notice that nothing is stated about the other_interest attribute after a tick, since this attribute is not under the control of this process and can be changed at any time by the other process. The turn attribute is under control of both processes. Hence changes to turn must be expressed in a careful manner, to avoid clashes (leading to inconsistencies) with possible changes

$$
\begin{array}{l}
\textbf{BEGIN}\ \text{PetersonProcess}[\,\text{ident}\colon \mathsf{bool}\,] \\
\qquad\qquad \cdots \quad //\ \text{see Figure 11} \\
\quad \textbf{ASSERTIONS} \\
\qquad\qquad \cdots \quad //\ \text{see Figure 11} \\
\quad \text{become\_critical:}\ \Big[\ \Big(\mathsf{own\_interest}(x) \wedge \mathsf{pre\_critical}(x) \wedge \neg\mathsf{critical}(x) \wedge \\
\qquad\qquad\qquad\qquad (\neg\mathsf{other\_interest}(x) \vee \mathsf{turn}(x) = \mathsf{ident}\Big) \\
\qquad\qquad\qquad \Rightarrow \Big(\mathsf{own\_interest}(\mathsf{tick}(x)) \wedge \neg\mathsf{pre\_critical}(\mathsf{tick}(x)) \wedge \\
\qquad\qquad\qquad\qquad \mathsf{critical}(\mathsf{tick}(x)) \wedge \\
\qquad\qquad\qquad\qquad //\ \text{turn cannot change to}\ \neg\mathsf{ident}\text{:} \\
\qquad\qquad\qquad\qquad (\mathsf{turn}(x) = \mathsf{ident} \Rightarrow \mathsf{turn}(\mathsf{tick}(x)) = \mathsf{ident})\Big)\ \Big] \\
\quad \text{critical\_remains\_or\_stops:}\ \Big[\ \Big(\mathsf{own\_interest}(x) \wedge \neg\mathsf{pre\_critical}(x) \wedge \mathsf{critical}(x)\Big) \\
\qquad\qquad\qquad\qquad \Rightarrow \Big(\neg\mathsf{pre\_critical}(\mathsf{tick}(x)) \wedge \\
\qquad\qquad\qquad\qquad\quad (//\ \text{either remain critical} \\
\qquad\qquad\qquad\qquad\quad (\mathsf{own\_interest}(\mathsf{tick}(x)) \wedge \mathsf{critical}(\mathsf{tick}(x)) \wedge \\
\qquad\qquad\qquad\qquad\qquad (\mathsf{turn}(x) = \mathsf{ident} \Rightarrow \mathsf{turn}(\mathsf{tick}(x)) = \mathsf{ident})) \\
\qquad\qquad\qquad\qquad\quad \vee \quad //\ \text{or stop being critical} \\
\qquad\qquad\qquad\qquad\quad (\neg\mathsf{own\_interest}(\mathsf{tick}(x)) \wedge \neg\mathsf{critical}(\mathsf{tick}(x))))\Big)\ \Big] \\
\quad \text{critical\_stops:}\ \Big[\ \mathsf{critical}(x) \Rightarrow \\
\qquad\qquad \Big(\Diamond(\mathsf{super})(\{y \in X \mid \neg\mathsf{critical}(y) \wedge \neg\mathsf{pre\_critical}(y) \wedge \\
\qquad\qquad\qquad\qquad\qquad \neg\mathsf{own\_interest}(y)\})(x)\Big)\ \Big] \\
\textbf{END}\ \text{PetersonProcess}
\end{array}
$$

Fig. 12. Specification of a (parametrised) Peterson process, part II

by the other process, see assertions ‘become_pre_critical’, ‘become_critical’ and ‘critical_remains_or_stops’ where restrictions occur which do not block changes by the other process. Notice how the assertion ‘become_pre_critical’ can be used for both processes at the same time—corresponding to transitions $\ell_1 \to \ell_2$, $m_1 \to m_2$ in Figure 6—but that only one “assignment” to turn can take place. The final assertion ‘critical_stops’ uses the eventually operator $\Diamond(\mathsf{super})$. It refers to $\Diamond$ for the superclass DiscreteTime, which can be expressed in terms of iteration, see the equivalences (6). It thus says that the critical section will be left, after an unspecified amount of time. This is needed for fairness: if processes do not leave their critical sections, their competitors will never get access.

We turn to the central specification Peterson in Figure 13. It has two attributes for the two processes, and two methods for indicating interest of these processes[16]. The assertions establish appropriate connections between the two processes (‘share_turn’ and ‘exchange_interest’), and also between the methods

[16] Actually, using some more expressive language of types, one could combine methods ppF, ppT into $\mathsf{pp}\colon X \to \prod_{b\in\mathsf{bool}} \text{PetersonProcess}[b]$ and $\mathsf{interestT}$, $\mathsf{interestF}$ into $\mathsf{interest}\colon X \to X^{\mathsf{bool}}$.

tick, interestT, interestF of the Peterson specification and corresponding methods of the processes. Notice that we do not write a requirement ppF(interestT(x)) $\underline{\leftrightarrow}$ ppF(x), because it is too strong: interestT sets own_interest of ppT to true, and thereby other_interest of ppF to false. Hence it does have an effect on ppF.

BEGIN Peterson
INHERIT FROM DiscreteTime // see Figure 7
METHODS
ppT: $X \longrightarrow$ PetersonProcess[true]
ppF: $X \longrightarrow$ PetersonProcess[false]
interestT: $X \longrightarrow X$
interestF: $X \longrightarrow X$
ASSERTIONS
share_turn: $\left[\ \mathsf{turn}(\mathsf{ppT}(x)) = \mathsf{turn}(\mathsf{ppF}(x))\ \right]$
exchange_interest: $\left[\ \begin{array}{l} \mathsf{own\_interest}(\mathsf{ppT}(x)) = \mathsf{other\_interest}(\mathsf{ppF}(x)) \wedge \\ \mathsf{other\_interest}(\mathsf{ppT}(x)) = \mathsf{own\_interest}(\mathsf{ppF}(x)) \end{array}\ \right]$
interestT: $\left[\ \begin{array}{l} \mathsf{ppT}(\mathsf{interestT}(x)) \underline{\leftrightarrow} \mathsf{interest}(\mathsf{ppT}(x)) \wedge \\ \mathsf{own\_interest}(\mathsf{ppF}(\mathsf{interestT}(x))) = \mathsf{own\_interest}(\mathsf{ppF}(x)) \wedge \\ \mathsf{pre\_critical}(\mathsf{ppF}(\mathsf{interestT}(x))) = \mathsf{pre\_critical}(\mathsf{ppF}(x)) \wedge \\ \mathsf{critical}(\mathsf{ppF}(\mathsf{interestT}(x))) = \mathsf{critical}(\mathsf{ppF}(x)) \end{array}\ \right]$
interestF: $\left[\ \begin{array}{l} \mathsf{ppF}(\mathsf{interestF}(x)) \underline{\leftrightarrow} \mathsf{interest}(\mathsf{ppF}(x)) \wedge \\ \mathsf{own\_interest}(\mathsf{ppT}(\mathsf{interestF}(x))) = \mathsf{own\_interest}(\mathsf{ppT}(x)) \wedge \\ \mathsf{pre\_critical}(\mathsf{ppT}(\mathsf{interestF}(x))) = \mathsf{pre\_critical}(\mathsf{ppT}(x)) \wedge \\ \mathsf{critical}(\mathsf{ppT}(\mathsf{interestF}(x))) = \mathsf{critical}(\mathsf{ppT}(x)) \end{array}\ \right]$
synchronise: $\left[\ \begin{array}{l} \mathsf{ppT}(\mathsf{tick}(x)) \underline{\leftrightarrow} \mathsf{tick}(\mathsf{ppT}(x)) \wedge \\ \mathsf{ppF}(\mathsf{tick}(x)) \underline{\leftrightarrow} \mathsf{tick}(\mathsf{ppF}(x)) \end{array}\ \right]$
CONSTRUCTORS
new: X
CREATION
init: $\left[\ \begin{array}{l} \neg\mathsf{own\_interest}(\mathsf{ppT}(\mathsf{new}) \wedge \neg\mathsf{pre\_critical}(\mathsf{ppT}(\mathsf{new}) \wedge \neg\mathsf{critical}(\mathsf{ppT}(\mathsf{new}) \wedge \\ \neg\mathsf{own\_interest}(\mathsf{ppF}(\mathsf{new}) \wedge \neg\mathsf{pre\_critical}(\mathsf{ppF}(\mathsf{new}) \wedge \neg\mathsf{critical}(\mathsf{ppF}(\mathsf{new}) \end{array}\ \right]$
END Peterson

Fig. 13. Specification of Peterson's algorithm

Next we come to correctness, expressed in Theorem 1 below. In our reasoning about the $\mathsf{Peterson}$ specification we use the next five predicates: for $x \in X$,

$$
\begin{aligned}
&\mathsf{critical\_exclusionT}(x)\\
&= \big(\,\mathsf{critical}(\mathsf{ppT}(x)) \Rightarrow \mathsf{own\_interest}(\mathsf{ppT}(x)) \wedge \neg\mathsf{pre\_critical}(\mathsf{ppT}(x))\,\big)\\
&\mathsf{critical\_exclusionF}(x)\\
&= \big(\,\mathsf{critical}(\mathsf{ppF}(x)) \Rightarrow \mathsf{own\_interest}(\mathsf{ppF}(x)) \wedge \neg\mathsf{pre\_critical}(\mathsf{ppF}(x))\,\big)\\
&\mathsf{critical\_turnT}(x)\\
&= \big(\,\mathsf{critical}(\mathsf{ppT}(x)) \wedge \mathsf{pre\_critical}(\mathsf{ppF}(x)) \Rightarrow \mathsf{turn}(\mathsf{ppT}(x))\,\big)\\
&\mathsf{critical\_turnF}(x)\\
&= \big(\,\mathsf{critical}(\mathsf{ppF}(x)) \wedge \mathsf{pre\_critical}(\mathsf{ppT}(x)) \Rightarrow \neg\mathsf{turn}(\mathsf{ppF}(x))\,\big)\\
&\mathsf{critical\_exclusion}(x)\\
&= \big(\;\mathsf{critical\_exclusionT}(x) \wedge \mathsf{critical\_exclusionF}(x) \wedge\\
&\qquad \mathsf{critical\_turnT}(x) \wedge \mathsf{critical\_turnF}(x) \wedge\\
&\qquad \neg\big(\mathsf{critical}(\mathsf{ppT}(x)) \wedge \mathsf{critical}(\mathsf{ppF}(x))\big)\,\big).
\end{aligned}
$$

Lemma 5. *The following five predicates are invariants for the* $\mathsf{Peterson}$ *class specification in Figure 13.*

$$
\begin{gathered}
\mathit{critical\_exclusionT},\;\; \mathit{critical\_exclusionF},\\
\mathit{critical\_exclusionT} \wedge \mathit{critical\_exclusionF} \wedge \mathit{critical\_turnT},\\
\mathit{critical\_exclusionT} \wedge \mathit{critical\_exclusionF} \wedge \mathit{critical\_turnF},\\
\mathit{critical\_exclusion}.
\end{gathered}
$$

Below we shall only use the last invariant $\mathsf{critical\_exclusion}$, but the others are convenient as intermediate steps.

Proof. According to Definition 2 we have to prove for each of the above predicates, say denoted by $Q \subseteq X$, that for all $x \in X$,

$$
Q(x) \Longrightarrow \begin{cases} Q(\mathsf{tick}(x)) \\ Q(\mathsf{interestT}(x)) \\ Q(\mathsf{interestF}(x)) \end{cases}
$$

This is not hard, but a lot of work, because of the many case distinctions that have to be made. Hence a proof tool is useful. $\square$

Theorem 1. *The specification of Peterson's algorithm in Figure 13 satisfies the following two safety and progress properties.*

1. *Mutual exclusion holds in all reachable states:*

$$\Box(\{y \in X \mid \neg(\mathit{critical}(\mathit{ppT}(y)) \wedge \mathit{critical}(\mathit{ppF}(y)))\})(\mathit{new})$$

2. *For both processes, requests to proceed to their critical section will eventually be granted: for all $x \in X$,*

$$
\begin{gathered}
\Diamond(\{y \in X \mid \mathit{critical}(\mathit{ppT}(y))\})(\mathit{interestT}(x))\\
\Diamond(\{y \in X \mid \mathit{critical}(\mathit{ppF}(y))\})(\mathit{interestF}(x)).
\end{gathered}
$$

Proof. 1. According to the definition of $\Box$, we have to produce an invariant Q with $Q(y) \Rightarrow \neg(\mathsf{critical}(\mathsf{ppT}(y)) \wedge \mathsf{critical}(\mathsf{ppF}(y)))$ and $Q(\mathsf{new})$. We take $Q = \mathsf{critical\_exclusion}$ from Lemma 5.

2. The two statements are proved in the same way, so we concentrate on the first one. The following intermediate statement is convenient: for all $x \in X$,

$$\begin{array}{l}\big(\,\mathsf{own\_interest}(\mathsf{ppT}(x)) \wedge \mathsf{pre\_critical}(\mathsf{ppT}(x)) \wedge \neg\mathsf{critical}(\mathsf{ppT}(x)) \wedge \\ \quad \exists n \in \mathbb{N}.\, \neg\mathsf{own\_interest}(\mathsf{tick}^n(\mathsf{ppF}(x))) \vee \mathsf{turn}(\mathsf{tick}^n(\mathsf{ppF}(x)))\,\big) \\ \qquad \Longrightarrow \\ \exists n \in \mathbb{N}.\, \mathsf{critical}\,(\mathsf{ppT}(\mathsf{tick}^n(x)))\end{array}$$

The proof of this statement proceeds by considering the least n such that $\neg\mathsf{own\_interest}(\mathsf{tick}^n(\mathsf{ppF}(x))) \vee \mathsf{turn}(\mathsf{tick}^n(\mathsf{ppF}(x)))$. For all $m \leq n$, one can show that after m ticks $\mathsf{own\_interest}$, $\mathsf{pre\_critial}$ and $\neg\mathsf{critical}$ hold for ppT. Hence the assertion 'become_critical' does the job after n ticks. $\Box$

10 Refinements between coalgebraic specifications

A refinement is a general construction to turn a model of one specification (usually called the concrete one) into a model of another specification (called the abstract specification in this context), using (essentially[17]) the same set of states. Typically, the concrete specification has more structure, and describes a particular way to realise the structure of the abstract specification in terms of its own concrete structure. In computer science a refinement usually adds certain implementation details, reducing the level of underspecification (sometimes called non-determinism), and possibly increasing the use of concurrency.

Constructions to turn a model of one specification into a model of another specification are well-known in mathematics. Typically, one can turn a pointed topological space into its "fundamental" or "Poincaré" group by using as elements homotopy classes of paths with the base point as beginning and end. Also, one can construct the integers from the natural numbers (via a quotient of $\mathbb{N} \times \mathbb{N}$), but in both these cases the state space changes in an essential way. An actual (but trivial) refinement is the construction of a topological space out of a metric space.

In this section we define refinements between coalgebraic specifications, basically as in [14]. Further, we present an abstract Peterson specification, and show how the specification from the previous section forms a refinement.

We now assume two coalgebraic specifications: $\mathcal{A}$ for abstract, and $\mathcal{C}$ for concrete. Let $\mathcal{IA}$ and $\mathcal{IC}$ be the associated polynomial functors capturing the interface of methods. For didactic reasons we first define a "simple refinement" (of $\mathcal{A}$ by $\mathcal{C}$), and postpone the general definition. We assume that both specifications have precisely one initial state[18], written as new. For a coalgebra $c\colon X \to \mathcal{IA}(X)$,

[17] We shall see later what 'essentially' means: a subset of the set of states forming an invariant is also allowed.

[18] And not a parametrised collection of initial states. The definition of refinement can easily be extended to include them as well.

we have predicates

$$\mathcal{A}\text{-Assert}(c) \subseteq X \qquad \text{and} \qquad \mathcal{A}\text{-Create}(c) \subseteq X$$

combining all assertions and creation conditions. Similarly for $\mathcal{C}$. How these predicates are obtained can best be seen in Figures 11 and 12 where we have not written quantifiers $\forall x \in X. \cdots$ in assertions. The induced predicate Peterson-Process-Assert(x) is then obtained as conjunction of all seven assertions. The predicate PetersonProcess-Create is obtained by viewing new as a parameter in the creation condition init.

A *simple refinement* of $\mathcal{A}$ by $\mathcal{C}$ consists of a construction which turns an arbitrary model of $\mathcal{C}$, consisting of a coalgebra $c\colon X \to \mathcal{IC}(X)$ and initial state $\mathsf{new} \in X$ with $\forall x \in X. \mathcal{C}\text{-Assert}(c)(x)$ and $\mathcal{C}\text{-Create}(c)(\mathsf{new})$, into a model of $\mathcal{A}$, consisting of a coalgebra $c'\colon X \to \mathcal{IA}(X)$ on the same state space, constructed out of c, and an initial state $\mathsf{new}' \in X$ constructed out of c and new, in such a way that $\forall x \in X. \mathcal{A}\text{-Assert}(c')(x)$ and $\mathcal{A}\text{-Create}(c')(\mathsf{new}')$.

A simple refinement between coalgebraic specifications thus consists of two things (as in TLA [22, 23]): a "refinement mapping"

$$(c\colon X \to \mathcal{IC}(X), \mathsf{new} \in X) \longmapsto (c'\colon X \to \mathcal{IA}(X), \mathsf{new}' \in X) \tag{7}$$

together with an implication

$$\begin{array}{l} \mathcal{C}\text{-Create}(c)(\mathsf{new}) \wedge \forall x \in X. \mathcal{C}\text{-Assert}(c)(x) \\ \quad\Longrightarrow \\ \mathcal{A}\text{-Create}(c')(\mathsf{new}') \wedge \forall x \in X. \mathcal{A}\text{-Assert}(c')(x). \end{array} \tag{8}$$

The definition of refinement says that a concrete model can be turned into an abstract model, in a straightforward way. In practice it usually does not work like this, because the requirement $\forall x \in X. \mathcal{A}\text{-Assert}(c')(x)$ cannot be proved for all $x \in X$. Then we often need to restrict the state space X to some subset $P \subseteq X$ for which we can prove $\forall x \in X. P(x) \Rightarrow \mathcal{A}\text{-Assert}(c')(x)$. This is good enough if we can additionally prove that P is an invariant (with respect to the constructed coalgebra c'), and that P holds for the constructed initial state new'. Then we know that all states we can reach from new' using c' remain within P. Thus, for a (non-simple) *refinement* the above implication changes into:

$$\begin{array}{l} \mathcal{C}\text{-Create}(c)(\mathsf{new}) \wedge \forall x \in X. \mathcal{C}\text{-Assert}(c)(x) \\ \quad\Longrightarrow \\ \mathcal{A}\text{-Create}(c')(\mathsf{new}') \wedge \Box(c')(\mathcal{A}\text{-Assert}(c'))(\mathsf{new}'). \end{array} \tag{9}$$

See the definition of $\Box$ in (4).

We turn to an example of a refinement with such an implication.

Proposition 5. *Consider the MutualExclusion specification in Figure 14 describing the essentials of mutual exclusion. The Peterson specification from Figure 13 is a refinement of MutualExclusion.*

BEGIN MutualExclusion

  INHERIT FROM DiscreteTime // see Figure 7

  METHODS

$$\begin{array}{l}
\mathsf{criticalT}\colon X \longrightarrow \mathsf{bool} \\
\mathsf{criticalF}\colon X \longrightarrow \mathsf{bool} \\
\mathsf{interestT}\colon X \longrightarrow X \\
\mathsf{interestF}\colon X \longrightarrow X
\end{array}$$

  ASSERTIONS

$$\text{exclusion: } \Big[\ \neg(\mathsf{criticalT}(x) \wedge \mathsf{criticalF}(x))\ \Big]$$

$$\text{progressT: } \Big[\ \Diamond(\mathsf{super})(\{y \in X \mid \mathsf{criticalT}(y)\})(\mathsf{interestT}(x))\ \Big]$$

$$\text{progressF: } \Big[\ \Diamond(\mathsf{super})(\{y \in X \mid \mathsf{criticalF}(y)\})(\mathsf{interestF}(x))\ \Big]$$

$$\text{critical_interest: } \left[\ \begin{array}{l}
\neg\mathsf{criticalT}(\mathsf{interestT}(x)) \wedge \\
\mathsf{criticalF}(\mathsf{interestT}(x)) = \mathsf{criticalF}(x) \wedge \\
\mathsf{criticalT}(\mathsf{interestF}(x)) = \mathsf{criticalT}(x) \wedge \\
\neg\mathsf{criticalF}(\mathsf{interestF}(x))
\end{array}\ \right]$$

  CONSTRUCTORS

$$\mathsf{new}\colon X$$

  CREATION

$$\text{init: } \Big[\ \neg\mathsf{criticalT}(\mathsf{new}) \wedge \neg\mathsf{criticalF}(\mathsf{new})\ \Big]$$

END MutualExclusion

Fig. 14. Specification of mutual exclusion

Proof. We will construct a refinement mapping as in (7) with an implication (9). Therefore we assume a coalgebra $c = (\mathsf{tick}, \mathsf{ppT}, \mathsf{ppF}, \mathsf{interestT}, \mathsf{interestF})$ and initial state new for the Peterson specification. We get a new coalgebra $c' = (\mathsf{tick}, \mathsf{criticalT}, \mathsf{criticalF}, \mathsf{interestT}, \mathsf{interestF})$ for the MutualExclusion specification by defining $\mathsf{criticalT}(x) = \mathsf{critical}(\mathsf{ppT}(x))$ and $\mathsf{criticalF}(x) = \mathsf{critical}(\mathsf{ppF}(x))$. As new initial state new' we simply take new from Peterson. Then clearly, the (abstract) creation conditions from MutualExclusion hold for new. Further, the predicate critical_exclusion from Lemma 5 is an invariant for c' (since it already is one for c) and implies the abstract assertions, see Theorem 1, and the proof of its first point. □

Refinement is a fundamental technique to establish the correctness of larger software (and also hardware) systems: the idea is to first concentrate on an abstract specification of the system—describing the essentials of the required behaviour, without going into any realisation issues—and refine this abstract specification, possibly in several steps, into a concrete specification, coming close to an actual implementation.

To conclude, we have introduced the subject of specification and verification for coalgebras via several examples, for which we have proved non-trivial results. The area is still under development, but has already reached a certain level of maturity, with its own theory and logic. For further background reading we refer to [18, 35]. An impression of current research topics in coalgebra can be obtained from the volumes [17, 19].

Acknowledgements Thanks are due to the members of the LOOP group at Nijmegen and Dresden (Joachim van den Berg, Marieke Huisman, Erik Poll, Ulrich Hensel and Hendrik Tews) for discussions and collaboration on the development of the LOOP compiler, and also for feedback on a draft version of this paper. Thanks also to Angelika Mader for discussions on temporal logic and the μ-calculus.

References

1. K.R. Apt and E.-R. Olderog. *Verification of Sequential and Concurrent Programs.* Springer, 2^{nd} rev. edition, 1997.
2. E. Astesiano, H.-J. Kreowski, and B. Krieg-Brückner, editors. *Algebraic Foundations of Systems Specification.* IFIP State-of-the-Art Reports. Springer, 1999.
3. C. Cîrstea. Coalgebra semantics for hidden algebra: parametrised objects and inheritance. In F. Parisi Presicce, editor, *Recent Trends in Data Type Specification*, number 1376 in Lect. Notes Comp. Sci., pages 174–189. Springer, Berlin, 1998.
4. B.A. Davey and H.A. Priestley. *Introduction to Lattices and Order.* Math. Textbooks. Cambridge Univ. Press, 1990.
5. E.A. Emerson. Temporal and modal logic. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B, pages 995–1072. Elsevier/MIT Press, 1990.
6. U. Hensel. *Definition and Proof Principles for Data and Processes.* PhD thesis, Techn. Univ. Dresden, Germany, 1999.
7. 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 Lect. Notes Comp. Sci., pages 105–121. Springer, Berlin, 1998.
8. C. Hermida and B. Jacobs. Structural induction and coinduction in a fibrational setting. *Inf. & Comp.*, 145:107–152, 1998.
9. M.W. Hirsch and S. Smale. *Differential Equations, Dynamical Systems, and Linear Algebra.* Academic Press, New York, 1974.
10. B. Jacobs. Mongruences and cofree coalgebras. In V.S. Alagar and M. Nivat, editors, *Algebraic Methodology and Software Technology*, number 936 in Lect. Notes Comp. Sci., pages 245–260. Springer, Berlin, 1995.
11. 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.
12. 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.
13. 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.

14. B. Jacobs. Invariants, bisimulations and the correctness of coalgebraic refinements. In M. Johnson, editor, *Algebraic Methodology and Software Technology*, number 1349 in Lect. Notes Comp. Sci., pages 276–291. Springer, Berlin, 1997.
15. B. Jacobs. The temporal logic of coalgebras via Galois algebras. Techn. Rep. CSI-R9906, Comput. Sci. Inst., Univ. of Nijmegen, 1999.
16. B. Jacobs. Object-oriented hybrid systems of coalgebras plus monoid actions. *Theor. Comp. Sci.*, 239, 1999, to appear.
17. B. Jacobs, L. Moss, H. Reichel, and J. Rutten, editors. *Coalgebraic Methods in Computer Science (CMCS'98)*, number 11 in Elect. Notes in Theor. Comp. Sci. Elsevier, Amsterdam, 1998. Available from URL: `http://www.elsevier.nl/locate/entcs/volume11.html`.
18. B. Jacobs and J. Rutten. A tutorial on (co)algebras and (co)induction. *EATCS Bulletin*, 62:222–259, 1997.
19. B. Jacobs and J. Rutten, editors. *Coalgebraic Methods in Computer Science (CMCS'99)*, number 19 in Elect. Notes in Theor. Comp. Sci. Elsevier, Amsterdam, 1999. Available from URL: `http://www.elsevier.nl/locate/entcs/volume19.html`.
20. B. Jacobs, J. van den Berg, M. Huisman, M. van Berkum, U. Hensel, and H. Tews. Reasoning about classes in Java (preliminary report). In *Object-Oriented Programming, Systems, Languages and Applications*, pages 329–340. ACM Press, 1998.
21. R.E. Kalman, P.L. Falb, and M.A. Arbib. *Topics in Mathematical System Theory.* McGraw-Hill Int. Series in Pure & Appl. Math., 1969.
22. L. Lamport. The temporal logic of actions. *ACM Trans. on Progr. Lang. and Systems*, 16(3):872–923, 1994.
23. L. Lamport. Specifying concurrent systems with TLA^+. In M. Broy and R. Steinbrüggen, editors, *Calculational System Design*, number 173 in Series F: Computer and Systems Sciences, pages 183–247. IOS Press, Amsterdam, 1999.
24. N. Lynch. Simulation techniques for proving properties of real-time systems. 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 375–424. Springer, Berlin, 1994.
25. N. Lynch. *Distributed Algorithms.* Morgan Kaufmann, 1996.
26. Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems.* Springer-Verlag, Berlin, 1992.
27. L.S. Moss. Coalgebraic logic. *Ann. Pure & Appl. Logic*, 1999. To appear.
28. S. Owre, J.M. Rushby, N. Shankar, and F. von Henke. Formal verification for fault-tolerant architectures: Prolegomena to the design of PVS. *IEEE Trans. on Softw. Eng.*, 21(2):107–125, 1995.
29. G.L. Peterson. Myths about the mutual exclusion problem. *Information Processing Letters*, 12(3):115–116, 1981.
30. G. Plotkin and M. Abadi. A logic for parametric polymorphism. In M. Bezem and J.F. Groote, editors, *Typed Lambda Calculi and Applications*, number 664 in Lect. Notes Comp. Sci., pages 361–375. Springer, Berlin, 1993.
31. A. Pnueli. The temporal semantics of concurrent programs. *Theor. Comp. Sci.*, 31:45–60, 1981.
32. E. Poll and J. Zwanenburg. A logic for abstract data types as existential types. In J.-Y. Girard, editor, *Typed Lambda Calculus and Applications*, number 1581 in Lect. Notes Comp. Sci., pages 310–324. Springer, Berlin, 1999.
33. H. Reichel. An approach to object semantics based on terminal co-algebras. *Math. Struct. in Comp. Sci.*, 5:129–152, 1995.

34. J. Rutten. Automata and coinduction (an exercise in coalgebra). In D. Sangiorigi and R. de Simone, editors, *Concur'98: Concurrency Theory*, number 1466 in Lect. Notes Comp. Sci., pages 194–218. Springer, Berlin, 1998.
35. J. Rutten. Universal coalgebra: a theory of systems. *Theor. Comp. Sci.*, 1999. To appear.
36. C. Stirling. Modal and temporal logics. In S. Abramsky, Dov M. Gabbai, and T.S.E. Maibaum, editors, *Handbook of Logic in Computer Science*, volume 2, pages 477–563. Oxford Univ. Press, 1992.
37. A.S. Tanenbaum and A.S. Woodhull. *Operating Systems: Design and Implementation*. Prentice Hall, 2^{nd} rev. edition, 1997.