Behaviour-Refinement of Coalgebraic Specifications with Coinductive Correctness Proofs

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

Abstract. *A notion of refinement is defined in the context of coalgebraic specification of classes in object-oriented languages. It tells us when objects in a "concrete" class behave exactly like (or: simulate) objects in an "abstract" class. The definition of refinement involves certain selection functions between procedure-inputs and attribute-outputs, which gives this notion considerable flexibility. The coalgebraic approach allows us to use coinductive proof methods in establishing refinements (via (bi)simulations). This is illustrated in several examples.*

1 Introduction

Refinement is an important notion in the stepwise construction of reliable software. It is used to express that an abstract description is realised by a concrete one, typically by filling-in some implementation details. This paper concentrates on refinement in an object-oriented setting. What is typical there is re-use of classes[1]: one tries to refine towards existing classes (available in some library). There are two important ways to construct new classes from old: inheritance and aggregation. Inheritance involves specialisation and puts classes in the "is-a" relationship, whereas aggregation involves using one class as a component of another, in the "has-a" relationship (see [19, 14.5] for a discussion of when to use "is-a" or "has-a"). Below we shall see examples of both inheritance and aggregation (for class specifications).

But first we shall define a notion of refinement between class specifications, using the "coalgebraic" specification format developed in [14, 13] (following [22]). Such a coalgebraic specification consists of a (black box) state space (typically written as X) with a number of attributes, capturing its data (like in instance variables), and a number of procedures which may change the state (and hence the values of these attributes). These attributes and procedures have to satisfy certain assertions (or constraints), which determine the appropriate behaviour. What is typically coalgebraic in this approach is that we say nothing about what is inside the state space X of a class (or about how to "algebraically" construct its elements), but only something about what can be observed (via the attributes) about an arbitrary state (i.e. inhabitant of X). Objects may be identified with such inhabitants. This coalgebraic state space X corresponds to the (product of the) hidden sorts in hidden-sorted algebra, see [6, 5, 18, 7, 1, 8].

In this setting we define what it means for a "concrete" class to refine an "abstract" class. The idea is that every object of the concrete class (when considered with appropriately selected attributes and procedures) behaves exactly like an object of the abstract

[1]In this paper we shall be concerned mostly with class specifications, in contrast to implementations. It can be argued that re-use of specifications is as important as re-use of implementations—especially in the long run, since implementations are more susceptible to change of technology.

class. The selection of attributes and procedures is essential because the concrete class may have many more attributes and procedures than needed for realising the desired behaviour of the abstract class. This selection is accomplished via two "selection" functions (the f, g in Definition 3.1) and yields a form of hiding. This emphasis on simulation of behaviour puts our notion of refinement firmly in the automata-theoretic tradition (where refinement (also called implementation) is defined as inclusion between sets of traces, see e.g. [17]), and not, in contrast, in the model-theoretic tradition (with emphasis on (behavioural) validity), see Section 5 for a brief comparison. In fact, the semantics of our coalgebraic class specifications may be described in terms of certain deterministic automata, see [14, 22]. But coalgebraic specification is different from automata-theoretic specification in that it does not describe states explicitly (e.g. in transition diagrams), but only implicitly via their observable behaviour[2]. This work is inspired by the earlier work on refinement in automata theory and in hidden sorted algebra (notably [6, 7]).

What makes the (coalgebraic) notion of refinement particularly useful is that it comes with a certain "coinductive" proof-technique. It allows us to answer the question of whether we have indistinguishable behaviour (for objects of the concrete and abstract classes) by giving an appropriate (bi)simulation relation on the state spaces. Showing that a given relation is a (bi)simulation involves proof-obligations for each of the attributes and procedures individually, which substantially reduces the proof burden. Also the use of such (bi)simulations is well-established in automata-theoretic approaches. Bisimilarity corresponds to behavioural satisfaction in hidden-sorted algebra, see e.g. [7, 2]. Therefore, coinduction can also be used as a proof-technique in hidden sorted algebra, see [7].

The contribution of the present paper lies in the following: it adapts these automata-theoretic approaches to refinement to an object-oriented setting (interpreted coalgebraically), involving non-trivial class specifications with inheritance and aggregation, and non-trivial correctness proofs. In doing so it clearly shows how to deal with different attributes and procedures in different classes via the earlier mentioned selection functions. The resulting approach uses arguments in elementary predicate logic, which should be unproblematic, both for humans and for computers. In fact, all the refinements in this paper have been fully formalised and proved in PVS [21][3]. Details about such formalisations may appear elsewhere.

The organisation of this paper is simple: we start in Section 2 by recalling the essentials of coalgebraic specification of classes. Subsequently, we describe the associated notion of refinement in Section 3, together with a coinductive proof-technique. Essentially, the remainder of this paper is devoted to several (standard) examples, involving counters, buffers, stacks and queues. Only in the final Section 5 we briefly compare our automata-theoretic notion of behaviour-refinement to an alternative, model-based notion of refinement.

This paper is the fourth (after [10, 14, 13]) in a series of papers by the author on using coalgebraic (in contrast to algebraic) notions in an object-oriented setting. The earlier papers are more foundational in nature. The theoretical content of the present paper is of very limited depth, and is hardly original, but it leads to applications of the earlier insights to an important aspect of object-oriented software construction, namely to establishing the correctness of various refinements. The eventual usefulness of this approach can only be established in its actual use. What seems encouraging is that the coalgebraic style of specifying classes is rather low level, and close to actual implementation. Therefore it is easy to understand. Moreover, it has a well-defined mathematical semantics (see notably [14]). Our (coalgebraic) notion of refinement scales up to a "hybrid" setting [12], combining discrete and continuous behaviour. And in future work we plan to generalise the present proof-techniques to include invariants (in a coalgebraic setting). This will allow us

[2] In particular, there is no way of restricting one's attention to *finite* state spaces in coalgebra.
[3] Using the proof tool actually revealed a few minor bugs in the original hand-written proofs.

to deal with "underspecified" classes (in which only part of the behaviour is prescribed). Such underspecification may be understood as a form of non-determinism, see [7]. Also in this future work, a semantical justification (using terminal models) will be given of the coalgebraic notion of refinement. But here we concentrate on actual use of coinductive proof techniques for refinements.

2 Coalgebraic specification

A coalgebraic class specification, as used in this paper, consists of three parts: methods, assertions and creation conditions, see the figures below. The methods are either *attributes* $\mathsf{at}\colon X \longrightarrow A$ or *procedures* $\mathsf{proc}\colon X \times B \longrightarrow X$. The X is an (unknown) state space, on which these methods act. The A and B are (known) constant sets: the set A gives the observable attribute values of a state space, and the set B serves as set of inputs (or parameters) of the procedure proc. Procedures may change states, and the effect of such changes may be visible via the attributes. The assertions in a coalgebraic specification describe the behaviour of the methods. They are as in algebraic specification, except that (1) an assertion only involves one single state, typically written as s ($\in X$), and (2) we use the post-fix "dot" notation, instead of the functional notation: $\mathsf{s.proc}(b)\mathsf{.at}$ means $\mathsf{at}(\mathsf{proc}(\mathsf{s}, b))$. The creation conditions describe the attribute values for a newly created object new of the class.

Suppose we have a class specification with attributes $\mathsf{at}_1\colon X \longrightarrow A_1, \ldots, \mathsf{at}_n\colon X \longrightarrow A_n$ and procedures $\mathsf{proc}_1\colon X \times B_1 \longrightarrow X, \ldots, \mathsf{proc}_m\colon X \times B_m \longrightarrow X$. The elements of the attribute output sets A_i will be considered as observable values to which clients have direct access. Hence we shall use actual equality $a = a'$ between elements $a, a' \in A_i$. In contrast, the state space X is seen as a black box to which clients only have limited access via the available operations. In particular, we cannot speak about equality $\mathsf{s} = \mathsf{t}$ of states $\mathsf{s}, \mathsf{t} \in X$, but only about bisimilarity $\mathsf{s} \leftrightarroweq \mathsf{t}$. Bisimilarity means: indistinguishability (via the coalgebraic operations). It need not be the same as equality, since two states may be different (internally), but display the same (external) behaviour. Then they are not equal $\neq$, but bisimilar $\leftrightarroweq$.

We shall use the bisimilarity sign $\leftrightarroweq$ in assertions in specifications (between terms inhabiting the state space). The proof rules for $\leftrightarroweq$ are the equivalence relation rules (reflexivity, symmetry and transitivity), plus the following two rules (for each $i \leq n$, $j \leq m$ and $b \in B_j$).

$$\frac{\mathsf{s} \leftrightarroweq \mathsf{t}}{\mathsf{s.at}_i = \mathsf{t.at}_i} \qquad\qquad \frac{\mathsf{s} \leftrightarroweq \mathsf{t}}{\mathsf{s.proc}_j(b) \leftrightarroweq \mathsf{t.proc}_j(b)}$$

(In fact, $\leftrightarroweq$ is the greatest relation satisfying these rules, so that $\mathsf{s} \leftrightarroweq \mathsf{t}$ can be identified with $\mathsf{s.proc}_{j_1}(b_1).\cdots.\mathsf{proc}_{j_n}(b_n)\mathsf{.at} = \mathsf{t.proc}_{j_1}(b_1).\cdots.\mathsf{proc}_{j_n}(b_n)\mathsf{.at}$ for all sequences $b_1 \in B_{j_1}, \ldots, b_n \in B_{j_n}$ of procedure-inputs.)

Figure 1 gives two typical examples of coalgebraic specifications involving stacks (last-in-first-out) and queues (first-in-first-out) for some data set A. Notice that the methods (attributes plus procedures) are the same in both specifications, but that the assertions are essentially different. The output type $1 + A$ of the top attributes is the set A augmented with an extra element $* \in 1 = \{*\}$ for undefined[4]. It allows us describe top as a partial function $X \rightharpoonup A$.

In coalgebraic specification—like in algebraic specification—it is often convenient to import an already existing specification into a new specification. This facilitates the incremental construction of specifications. Coalgebraically, this import-mechanism corresponds to what is called *inheritance*, but algebraically it corresponds to *parametrisation*, see [13]

[4]One can read $*$ as null.

```
class spec: Stack($A$)
    methods:
        push: $X \times A \longrightarrow X$
        pop: $X \longrightarrow X$
        top: $X \longrightarrow 1 + A$
    assertions:
        s.push($a$).top $= a$
        s.push($a$).pop $\underline{\leftrightarrow}$ s
        s.top $= *$ $\vdash$ s.pop $\underline{\leftrightarrow}$ s
    creation:
        new.top $= *$
end class spec
```

```
class spec: Queue($A$)
    methods:
        push: $X \times A \longrightarrow X$
        pop: $X \longrightarrow X$
        top: $X \longrightarrow 1 + A$
    assertions:
        s.top $= *$ $\vdash$ s.push($a$).top $= a$
        s.top $= *$ $\vdash$ s.push($a$).pop $\underline{\leftrightarrow}$ s
        s.top $= *$ $\vdash$ s.pop $\underline{\leftrightarrow}$ s
        s.top $\neq *$ $\vdash$ s.push($a$).top $=$ s.top
        s.top $\neq *$ $\vdash$
            s.push($a$).pop $\underline{\leftrightarrow}$ s.pop.push($a$)
    creation:
        new.top $= *$
end class spec
```

Figure 1: Stack and queue specifications

for details about the underlying semantical dualities[5]. We shall use inheritance in some of the examples later on, via the keyword "**inherits from:** $\mathcal{P}$" in a specification $\mathcal{C}$—with $\mathcal{P}$ for 'parent' (sometimes called ancestor) and $\mathcal{C}$ for 'child' (also called descendant, or subclass). The specialised class specification $\mathcal{C}$ then automatically contains all the methods, assertions and creation conditions of the general class specification $\mathcal{P}$. But $\mathcal{C}$ may add its own (additional) methods, assertions, and creation conditions.

3 Behaviour-refinements

In this section we define what it means for one "concrete" class specification $\mathcal{C}$ to refine an "abstract" class specification $\mathcal{A}$. Typically in such a situation, $\mathcal{C}$ contains more implementation details, or is more easily available than $\mathcal{A}$. In an object-oriented setting with a library of classes at hand, one tries to refine towards existing classes, for example because (reliable) implementations of these are available. What we will define is behaviour-refinement in contrast to what may be called model-refinement. Behaviour-refinement is about imitation of behaviour and model-refinement is about validity of assertions, see Section 5.

Assume our abstract class specification $\mathcal{A}$ has n attributes and m procedures with the following types.

$$X \xrightarrow{\mathsf{at}_1} A_1, \ldots, X \xrightarrow{\mathsf{at}_n} A_n \qquad \text{and} \qquad X \times B_1 \xrightarrow{\mathsf{proc}_1} X, \ldots, X \times B_m \xrightarrow{\mathsf{proc}_m} X$$

For convenience we shall form one set containing all these procedure-input types B_i via disjoint union $+$:

$$B_1 + \cdots + B_m = \{\langle i, b\rangle \mid i \leq m \text{ and } b \in B_i\}$$

and for $\beta = \langle i, b\rangle \in B_1 + \cdots + B_m$ we shall write $\mathsf{s.proc}(\beta)$ for $\mathsf{s.proc}_i(b)$. In this way we think that the m procedures $\mathsf{proc}_1 : X \times B_1 \longrightarrow X, \ldots, \mathsf{proc}_m : X \times B_m \longrightarrow X$ in $\mathcal{A}$ are combined into one single procedure $\mathsf{proc} : X \times (B_1 + \cdots + B_m) \longrightarrow X$[6].

[5]Restriction versus extension via right versus left adjoints to forgetful functors.

[6]In a similar way one can combine the n attributes $\mathsf{at}_1 : X \longrightarrow A_1, \ldots, \mathsf{at}_n : X \longrightarrow A_n$ into a single attribute $\mathsf{at} : X \longrightarrow (A_1 \times \cdots \times A_n)$ using Cartesian products $\times$. This will be used implicitly.

Similarly we assume we have a "concrete" class specification $\mathcal{C}$, say with methods

$$X \xrightarrow{\mathsf{at}_1} C_1, \ldots, X \xrightarrow{\mathsf{at}_k} C_k \qquad \text{and} \qquad X \times D_1 \xrightarrow{\mathsf{proc}_1} X, \ldots, X \times D_\ell \xrightarrow{\mathsf{proc}_\ell} X$$

These procedures can be combined into a single procedure $\mathsf{proc}\colon X \times (D_1 + \cdots + D_\ell) \longrightarrow X$.

3.1. Definition. For an "abstract" and a "concrete" class specifications $\mathcal{A}$ and $\mathcal{C}$ as above, we say that $\mathcal{C}$ is a **behaviour-refinement** (or simply a **refinement**) of $\mathcal{A}$ if there are both

1. a reachable state r in $\mathcal{C}$ (i.e. a state-term r which can be obtained from the initial state new in $\mathcal{C}$ via a number of procedure applications);
2. two "selection" functions g, f between (combined) procedure-input and attribute-output sets

$$\begin{tikzcd}
\underbrace{\begin{matrix} D_1 + \cdots + D_\ell \\ C_1 \times \cdots \times C_k \end{matrix}}_{\mathcal{C}} & \underbrace{\begin{matrix} B_1 + \cdots + B_m \\ A_1 \times \cdots \times A_n \end{matrix}}_{\mathcal{A}}
\end{tikzcd}$$

$$D_1 + \cdots + D_\ell \xleftarrow{\quad g \quad} B_1 + \cdots + B_m \qquad C_1 \times \cdots \times C_k \xrightarrow[\quad f \quad]{} A_1 \times \cdots \times A_n$$

such that the n-tuple of attribute values

$$(\mathsf{new.proc}(\beta_1).\cdots.\mathsf{proc}(\beta_p).\mathsf{at}_1,\ \ldots,\ \mathsf{new.proc}(\beta_1).\cdots.\mathsf{proc}(\beta_p).\mathsf{at}_n)$$

in $A_1 \times \cdots \times A_n$ is the same as the outcome of the selection

$$f(\mathsf{r.proc}(g(\beta_1)).\cdots.\mathsf{proc}(g(\beta_p)).\mathsf{at}_1,\ \ldots,\ \mathsf{r.proc}(g(\beta_1)).\cdots.\mathsf{proc}(g(\beta_p)).\mathsf{at}_k),$$

for all sequences $\beta_1, \ldots, \beta_p \in B_1 + \cdots + B_m$ of inputs (in class $\mathcal{A}$).

The function g translates procedure-inputs in $\mathcal{A}$ into procedure-inputs in $\mathcal{C}$, and f translates attribute-outputs in $\mathcal{C}$ back into attribute-outputs in $\mathcal{A}$. The required equation says that the f-selection of the observable attribute-outputs of a g-selected procedure-input sequence applied to r is the same as the observations resulting in $\mathcal{A}$ from this same procedure-input sequence. This shows that we can simulate (via f, g) in the concrete class $\mathcal{C}$ the observable behaviour in the abstract class $\mathcal{A}$. The opposite direction of these selection functions—contravariantly between inputs and covariantly between outputs—plays an important role in a so-called behaviour-realisation adjunction (see [11]), giving a canonical relation between automata displaying certain behaviour, and behaviours which can be realised.

In many situations the above reachable state r in the concrete class $\mathcal{C}$ will simply be the initial state new. And mostly, $\mathcal{C}$ will have more attributes and procedures than the abstract class $\mathcal{A}$. The attribute-selection function f can then consist of a number of projection functions selecting appropriate attributes. And the procedure-selection function g can consist of several coprojection functions (or insertions), selecting appropriate procedures. In this way we hide the additional methods. In practice, the concrete class $\mathcal{C}$ will often simply contain all the attributes and procedures of the abstract class $\mathcal{A}$. This then determines the selection functions f and g in an obvious way. We shall see examples below.

We should mention that the above definition only really makes sense for class specifications in which the behaviour of the initial state new is completely determined. All abstract and concrete example specifications below will be of this kind. Refinement between "underspecified" classes (in which there may be several states satisfying the behavioural constraints of new) will be studied in future work.

We conclude this section with a crucial *coinductive* proof technique for refinements. It allows us to consider refinements step-by-step, instead of at once for all sequences $\beta_1, \ldots, \beta_p$ as in the previous definition. Such a coinduction result may be found in various forms, see e.g. [25, Theorem 3.2], [17, Proposition 12], and may be traced back to [15, 20]. See [16] for an overview (concerning non-deterministic automata).

3.2. Lemma. *Consider abstract and concrete classes $\mathcal{A}$ and $\mathcal{C}$ as in the previous definition, together with a reachable state* r *as in 1. and selection functions g, f as in 2. Then $\mathcal{C}$ refines $\mathcal{A}$ (via r, g, f) if there is a bisimulation relation $R \subseteq \mathcal{C} \times \mathcal{A}$ satisfying*

$$(\mathsf{r}, \mathsf{new}) \in R \qquad \textit{and} \qquad (\mathsf{s}, \mathsf{t}) \in R \Rightarrow \begin{cases} f(\mathsf{s.at}_1, \ldots, \mathsf{s.at}_k) = (\mathsf{t.at}_1, \ldots, \mathsf{t.at}_n) & \textit{and} \\ (\mathsf{s.proc}(g(\beta)), \mathsf{t.proc}(\beta)) \in R, & \textit{for all } \beta. \end{cases}$$

Proof. The result follows directly from the fact that for all sequences $\beta_1, \ldots, \beta_p$

$$(\mathsf{r.proc}(g(\beta_1)). \cdots .\mathsf{proc}(g(\beta_p)),\ \mathsf{new.proc}(\beta_1). \cdots .\mathsf{proc}(\beta_p)) \in R,$$

which is shown by induction on the length p of the sequence. □

The essence of this result is that bisimilar elements in a (state space of a) coalgebra become equal when mapped to the terminal coalgebra, see e.g. [24, 14]. Hence we speak of a "coinductive" proof[7].

4 Examples of refinements

We illustrate the coalgebraic approach to (behaviour-) refinement in a number of (standard) examples. First we show how counting to n^2 can be simulated via to counters counting to n (just like counting to 100 can be done via two counters to 10 with a 'carry'). Then we present a refinement of a reliable buffer via an unreliable buffer with a repeater, and finally we consider various refinements of the stack and queue specifications in Figure 1 via arrays.

4.1 Counters

```
class spec: Count($n$: $\mathbb{N}_{>0}$)
    methods:
        val: $X \longrightarrow \{0, 1, 2, \ldots, n-1\}$
        next: $X \longrightarrow X$
        clear: $X \longrightarrow X$
    assertions:
        $\mathsf{s.val} \neq n-1 \vdash \mathsf{s.next.val} = \mathsf{s.val} + 1$
        $\mathsf{s.val} = n-1 \vdash \mathsf{s.next.val} = 0$
        $\mathsf{s.clear.val} = 0$
    creation:
        $\mathsf{new.val} = 0$
end class spec
```

Figure 2: A specification of counters modulo n

Our starting point is the specification in Figure 2 of a simple counter counting modulo $n : \mathbb{N}_{>0} = \{m \in \mathbb{N} \mid m > 0\}$, via a next procedure, producing a state with the next value.

[7]The dual notion of "inductive" proof is based on initiality (of algebras).

This n is a parameter in the specification. Our aim is to refine counting up to n^2 via two counters up to n, serving as first and second digit, see the double counter specification DCount(n) in Figure 3. The auxiliary counters to n appear as attribute components Count(n) in the specification. This use of classes as components in another class is called aggregation[8]. There is a new "global" attribute dval defined in terms of the "local" val attributes of the first and second digit[9]. Further there are methods dnext and dclear, which—as we will show—behave as in Count(n^2). We have added an additional rounding procedure round which sets the first digit to 0, and which possibly increments the second digit depending on whether the first digit is closer to 0 or closer to $n-1$. The $\underline{\leftrightarrow}$ signs in this specification refers to bisimilarity on Count(n). And similarly, the new's on the right hand side of the $\underline{\leftrightarrow}$ sign in the creation clause refer to the initial state of the Count(n) specification.

```
class spec: DCount($n$: $\mathbb{N}_{>0}$)
  methods:
    first: $X \longrightarrow$ Count($n$)
    second: $X \longrightarrow$ Count($n$)
    dnext: $X \longrightarrow X$
    dclear: $X \longrightarrow X$
    round: $X \longrightarrow X$
    dval: $X \longrightarrow \{0, 1, 2, \ldots, n^2 - 1\}$
  assertions:
    s.dnext.first $\underline{\leftrightarrow}$ s.first.next
    s.dclear.first $\underline{\leftrightarrow}$ s.first.clear
    s.dclear.second $\underline{\leftrightarrow}$ s.second.clear
  assertions:
    s.dval $= n \cdot$ (s.second.val) $+$ s.first.val
    s.first.val $\neq n - 1 \vdash$
      s.dnext.second $\underline{\leftrightarrow}$ s.second
    s.first.val $= n - 1 \vdash$
      s.dnext.second $\underline{\leftrightarrow}$ s.second.next
    s.round.first $\underline{\leftrightarrow}$ s.first.clear
    s.first.val $< \frac{n}{2} \vdash$
      s.round.second $\underline{\leftrightarrow}$ s.second
    s.first.val $\geq \frac{n}{2} \vdash$
      s.round.second $\underline{\leftrightarrow}$ s.second.next
  creation:
    new.first $\underline{\leftrightarrow}$ new
    new.second $\underline{\leftrightarrow}$ new
end class spec
```

Figure 3: A specification of two coupled counters (both modulo n)

Intuitively, it may be clear that DCount(n) refines Count(n^2). But we seek a formal proof. Therefore we first define appropriate selection functions f, g between the (combined) attribute-outputs and procedure-inputs. In the Count(n^2) specification the output type is simply $\{0, 1, \ldots, n^2 - 1\}$. And the combined input type is $1 + 1$, where 1 is the singleton set $\{*\}$, which serves as trivial input set of both the next and of the clear procedure. This set $1 + 1$ may be identified with the two-element set $\{0, 1\}$, where 0 stands for the trivial input of next and 1 for the input of clear. In this way we can combine the three separate methods in Count(n^2) into a single (coalgebraic) method $X \longrightarrow \{0, 1, \ldots, n^2 - 1\} \times X^{\{0,1\}}$.

The combined output type of the DCount(n) class specification is $\{0, 1, \ldots, n^2 - 1\} \times$ Count$(n) \times$ Count(n). And the combined input type is $1 + 1 + 1 = \{0, 1, 2\}$ where 0 stands for input of dnext, 1 for input of dclear, and 2 for input of round. We have to produce selection functions

[8] So far we have used actual sets A as attribute-outputs, whereas in the class specification DCount(n) we use other classes Count(n) as attribute-outputs. Semantically, one can read for Count(n) any carrier set of a coalgebraic model of the Count(n) specification, see [14]. A canonical choice is to take the terminal model, which in this case has carrier set (or state space) $\{0, 1, \ldots, n - 1\}$.

[9] The first digit in the specification corresponds to the first digit from the right as in decimal notation.

$$\begin{tikzcd} 1+1+1=\{0,1,2\} & \{0,1\}=1+1 \arrow[l, "g"'] \\ \underbrace{\{0,1,\ldots,n^2-1\}\times \mathrm{Count}(n)\times\mathrm{Count}(n)}_{\mathrm{DCount}(n)} \arrow[r, "f"'] & \underbrace{\{0,1,\ldots,n^2-1\}}_{\mathrm{Count}(n^2)} \end{tikzcd}$$

It is clear what these functions should be: f is the first projection, and g is the identity-insertion $\{0,1\} \hookrightarrow \{0,1,2\}$. These functions select the appropriate attributes and procedures in DCount(n) which will be used in simulating the behaviour of Count(n^2). And they hide the other attributes first, second and the remaining procedure round. As reachable state r in the concrete class DCount(n) we simply take the initial state new. A coinduction proof that DCount(n) is a behaviour-refinement of Count(n^2) requires by Lemma 3.2 a bisimulation relation $R \subseteq \mathrm{DCount}(n) \times \mathrm{Count}(n^2)$ satisfying

$$(\mathsf{new},\mathsf{new}) \in R \qquad \text{and} \qquad (\mathsf{s},\mathsf{t}) \in R \Rightarrow \begin{cases} \mathsf{s.dval} = \mathsf{t.val} & \text{and} \\ (\mathsf{s.dnext},\mathsf{t.next}) \in R & \text{and} \\ (\mathsf{s.dclear},\mathsf{t.clear}) \in R. \end{cases}$$

A relation $R \subseteq \mathrm{DCount}(n) \times \mathrm{Count}(n^2)$ that does the job is:

$$R = \{(\mathsf{s},\mathsf{t}) \mid \mathsf{s.dval} = \mathsf{t.val}\}. \tag{1}$$

We show in detail that R is indeed a bisimulation.

1. In DCount(n) we have $\mathsf{new.dval} = n \cdot (\mathsf{new.second.val}) + \mathsf{new.first.val} = n \cdot (\mathsf{new.val}) + \mathsf{new.val} = n \cdot 0 + 0 = 0$. And the initial state new in Count(n^2) satisfies $\mathsf{new.val} = 0$ by definition. Hence the pair of initial states $(\mathsf{new},\mathsf{new})$ is in R.
2. If $(\mathsf{s},\mathsf{t}) \in R$, then $\mathsf{s.dval} = \mathsf{t.val}$ by definition of R.
3. If $(\mathsf{s},\mathsf{t}) \in R$, then $(\mathsf{s.dnext},\mathsf{t.next}) \in R$ holds: we distinguish the two cases (1) $\mathsf{s.first.val} = n-1$ and (2) $\mathsf{s.first.val} \neq n-1$. In the first case we calculate:

$$\begin{aligned}
&\mathsf{s.dnext.val}\\
&= n \cdot (\mathsf{s.dnext.second.val}) + \mathsf{s.dnext.first.val}\\
&= n \cdot (\mathsf{s.second.next.val}) + \mathsf{s.first.next.val}\\
&= \begin{cases} n\cdot 0 + 0 & \text{if } \mathsf{s.second.val} = n-1 \\ n \cdot (\mathsf{s.second.val}+1) + 0 & \text{otherwise} \end{cases}\\
&= \begin{cases} 0 & \text{if } n \cdot (\mathsf{s.second.val}) + (n-1) = n^2-1 \\ n \cdot (\mathsf{s.second.val}) + (n-1) + 1 & \text{otherwise} \end{cases}\\
&\overset{(*)}{=} \begin{cases} 0 & \text{if } \mathsf{t.val} = n^2-1 \\ \mathsf{t.val}+1 & \text{otherwise} \end{cases}\\
&= \mathsf{t.next.val}.
\end{aligned}$$

where the equation (*) holds since $(\mathsf{s},\mathsf{t}) \in R$. Similarly, in the second case $\mathsf{s.first.val} \neq n-1$ we get $\mathsf{t.val} \neq n^2-1$, by assumption. Hence

$$\begin{aligned}
\mathsf{s.dnext.val} &= n \cdot (\mathsf{s.dnext.second.val}) + \mathsf{s.dnext.first.val}\\
&= n \cdot (\mathsf{s.second.val}) + \mathsf{s.first.next.val}\\
&= n \cdot (\mathsf{s.second.val}) + \mathsf{s.first.val} + 1\\
&\overset{(*)}{=} \mathsf{t.val} + 1\\
&= \mathsf{t.next.val}.
\end{aligned}$$

4. The final implication $(\mathsf{s},\mathsf{t}) \in R \Rightarrow (\mathsf{s.dclear},\mathsf{t.clear}) \in R$ holds, since one easily checks that $\mathsf{s.dclear.dval} = 0 = \mathsf{t.clear.val}$.

Thus we have proved the following result.

4.1. Proposition. *The* Count(n^2) *specification in Figure 2 is refined by the* DCount(n) *specification in Figure 3, via the relation (1).* □

In the DCount specification (in Figure 3) we have chosen to use the special names dval, dnext and dclear (with 'd') for the methods corresponding to val, next and clear in the Count specification (in Figure 2). We did so in order to emphasise the difference. But, in retrospect, we see that there is no compelling reason for using different names in DCount. Even stronger, using the same names directly suggests how to define the selection functions f, g. We shall follow this approach in our other examples below.

4.2 Buffers

Our next example is adapted from [3]. It involves buffers which may be empty or contain a single element from a data set A. Figure 4 contains two class specifications describing two such buffers. The first, Buffer(A), behaves as expected. The second buffer $\text{Buffer}_{\text{UF}}(A)$ is unreliable, in the sense that putting an element in the buffer may fail. But it may not fail infinitely many times: it will succeed at some stage after a finite (but unspecified) number of trials (via the existential quantifier below). This makes it an unreliable, but fair buffer. $\text{Buffer}_{\text{UF}}(A)$ is an example of an underspecified class, involving a certain degree of non-determinism. The success or failure of putting an element is indicated by an acknowledgement attribute $\mathsf{ack}\colon X \longrightarrow \{\mathrm{n}, \mathrm{y}\}$, with outcome n for failure and y for success. We use the notation $\mathsf{s.put}(a)^{(n)}$ as abbreviation: $\mathsf{s.put}(a)^{(0)}$ is s, and $\mathsf{s.put}(a)^{(n+1)}$ is $\mathsf{s.put}(a)^{(n)}.\mathsf{put}(a)$.

```
class spec: Buffer(A)
  methods:
      push: $X \times A \longrightarrow X$
      empty: $X \longrightarrow X$
      display: $X \longrightarrow 1 + A$
  assertions:
      s.empty.display $= *$
      s.display $= * \vdash$
          s.push($a$).display $= a$
      s.display $= b \vdash$
          s.push($a$) $\underline{\leftrightarrow}$ s
  creation:
      new.display $= *$
end class spec
```

```
class spec: Buffer_UF(A)
  methods:
      put: $X \times A \longrightarrow X$
      empty: $X \longrightarrow X$
      display: $X \longrightarrow 1 + A$
      ack: $X \longrightarrow \{\mathrm{n}, \mathrm{y}\}$
  assertions:
      s.display $= *$, s.put($b$).ack $= \mathrm{y} \vdash$
          s.put($b$).display $= b$
      s.display $= *$, s.put($b$).ack $= \mathrm{n} \vdash$
          s.put($b$) $\underline{\leftrightarrow}$ s
      s.display $= b \vdash$ s.put($a$) $\underline{\leftrightarrow}$ s
      s.display $= * \vdash \exists n > 0$ s.put($a$)$^{(n)}$.ack $= \mathrm{y}$
      s.empty.display $= *$
      s.empty.ack $= \mathrm{y}$
  creation:
      new.display $= *$
      new.ack $= \mathrm{n}$
end class spec
```

Figure 4: Buffer specifications

Our aim is to hide the unreliability of $\text{Buffer}_{\text{UF}}(A)$ by adding an extra level. We do this by first writing a specification of a class R-Buffer(A) "on top of" $\text{Buffer}_{\text{UF}}(A)$ which hides the possible failure of the put by repeating this put until it does succeed. And secondly, by showing that this new class specification refines the "unproblematic" specification Buffer(A). We shall use inheritance to make R-Buffer(A) a subclass specification of $\text{Buffer}_{\text{UF}}(A)$. This means that R-Buffer(A) has all the methods, assertions and creation conditions of $\text{Buffer}_{\text{UF}}(A)$, plus something extra, which is required explicitly.

```
class spec: R-Buffer($A$)
    inherits from: $\text{Buffer}_{\text{UF}}(A)$
    methods:
        push: $X \times A \longrightarrow X$
    assertions:
        $\text{s.display} = b \vdash \text{s.push}(a) \underline{\leftrightarrow} \text{s}$
        $\text{s.display} = *, \text{s.put}(a).\text{ack} = \text{y} \vdash \text{s.push}(a) \underline{\leftrightarrow} \text{s.put}(a)$
        $\text{s.display} = *, \text{s.put}(a).\text{ack} = \text{n} \vdash \text{s.push}(a) \underline{\leftrightarrow} \text{s.put}(a).\text{push}(a)$
end class spec
```

Figure 5: A buffer repeating the unreliable put

4.2. Proposition. *The* R-Buffer(A) *class specification in Figure 5 with repeating unreliable* put *refines the reliable* Buffer(A) *class specification from Figure 4.*

Notice that the selection functions f, g from Definition 3.1 are trivial in this case by our choice of method names: what we need is a relation $R \subseteq$ R-Buffer(A) $\times$ Buffer(A) which holds for the initial states: $R(\text{new}, \text{new})$ and also satisfies: $R(\text{s}, \text{t})$ implies both $R(\text{s.empty}, \text{t.empty})$ and $R(\text{s.push}(a), \text{t.push}(a))$.

Proof. Take $R = \{(\text{s}, \text{t}) \mid \text{s.display} = \text{t.display}\}$. Then it is easy to see that $(\text{new}, \text{new}) \in R$ and $(\text{s}, \text{t}) \in R \Rightarrow (\text{s.empty}, \text{t.empty}) \in R$. The implication $(\text{s}, \text{t}) \in R \Rightarrow (\text{s.push}(a), \text{t.push}(a)) \in R$ holds directly in case $\text{t.display} = \text{s.display} = b$. And if $\text{t.display} = \text{s.display} = *$, then clearly $\text{t.push}(a).\text{display} = a$. But also $\text{s.push}(a).\text{display} = a$ by the following argument. Let n be least with $\text{s.put}(a)^{(n)}.\text{ack} = \text{y}$. Then for $i < n$ we have $\text{s.put}(a)^{(i)}.\text{ack} = \text{n}$ and $\text{s.put}(a)^{(i)}.\text{display} = *$. Hence $\text{s.push}(a).\text{display} = \text{s.put}(a)^{(n-1)}.\text{put}(a).\text{display} = a$. □

This idea of putting a new layer on top of an unreliably functioning existing layer in order to improve the quality of service is well-established and often used (e.g. in data-storage or in communication). We have shown in a very simple example how our notion of refinement can be used to formally show the correctness of such layered systems. The same is done in terms of appropriate notions of refinement between automata (see e.g. [17, 23]).

4.3 Stacks

The standard way to refine stacks uses arrays, see e.g. [4, 7]: a stack is represented as an initial segment of an array, with pushing and popping at the end of the segment. We shall illustrate this in our coalgebraic setting, and therefore we first introduce a coalgebraic specification Array(A) of (unbounded) arrays[10], of some data set A, see Figure 6.

Using this specification of arrays, we can write a refinement $\text{Stack}_1(A)$ of Stack(A) as in Figure 7, with Array(A) as a component. There is another component $\mathbb{N}$ in this $\text{Stack}_1(A)$ specification, given by the end attribute, referring to the end of the segment in the array. Inserting an element will be done in the next position end + 1. The top, push and pop

[10] This Array(A) specification contains one attribute tell, whose type we have written as $X \times \mathbb{N} \longrightarrow 1 + A$. Formally, it should have been an attribute $X \longrightarrow (1 + A)^{\mathbb{N}}$, but that is less readable.

```
class spec: Array($A$)
    methods:
        tell: $X \times \mathbb{N} \longrightarrow 1 + A$
        put: $X \times A \times \mathbb{N} \longrightarrow X$
        clear: $X \times \mathbb{N} \longrightarrow X$
    assertions:
        $n = m \vdash \mathsf{s.put}(a, n).\mathsf{tell}(m) = a$
        $n \neq m \vdash \mathsf{s.put}(a, n).\mathsf{tell}(m) = \mathsf{s.tell}(m)$
        $n = m \vdash \mathsf{s.clear}(n).\mathsf{tell}(m) = *$
        $n \neq m \vdash \mathsf{s.clear}(n).\mathsf{tell}(m) = \mathsf{s.tell}(m)$
    creation:
        $\mathsf{new.tell}(n) = *$
end class spec
```

Figure 6: Array specification

methods are defined in terms of the other methods. The specification uses the monus (or truncated subtraction) function $\dot{-}$ given by $x \dot{-} y = \max\{x - y, 0\}$.

```
class spec: Stack$_1$($A$)
  methods:
    end: $X \longrightarrow \mathbb{N}$
    ar: $X \longrightarrow \mathrm{Array}(A)$
    top: $X \longrightarrow 1 + A$
    push: $X \times A \longrightarrow X$
    pop: $X \longrightarrow X$
  assertions:
    $\mathsf{s.push}(a).\mathsf{end} = \mathsf{s.end} + 1$
    $\mathsf{s.push}(a).\mathsf{ar} \underline{\leftrightarrow} \mathsf{s.ar.put}(a, \mathsf{s.end} + 1)$
  assertions:
    $\mathsf{s.pop.end} = \mathsf{s.end} \dot{-} 1$
    $\mathsf{s.pop.ar} \underline{\leftrightarrow} \mathsf{s.ar}$
    $\mathsf{s.top} = \mathsf{s.ar.tell(s.end)}$
  creation:
    $\mathsf{new.end} = 0$
    $\mathsf{new.ar} \underline{\leftrightarrow} \mathsf{new}$
end class spec
```

Figure 7: The refinement of stacks via arrays

We shall coinductively prove that the specification Stack$_1(A)$ refines the earlier specification Stack(A). The proof requires a relation $R \subseteq \mathrm{Stack}_1(A) \times \mathrm{Stack}(A)$ satisfying:

$$(\mathsf{new}, \mathsf{new}) \in R \qquad \text{and} \qquad (\mathsf{s}, \mathsf{t}) \in R \Rightarrow \begin{cases} \mathsf{s.top} = \mathsf{t.top} & \text{and} \\ (\mathsf{s.pop}, \mathsf{t.pop}) \in R & \text{and} \\ (\mathsf{s.push}(a), \mathsf{t.push}(a)) \in R. & \end{cases}$$

The relation $R \subseteq \mathrm{Stack}_1(A) \times \mathrm{Stack}(A)$ that we shall use is

$$R = \{(\mathsf{s}, \mathsf{t}) \mid \forall n \in \mathbb{N}\ \mathsf{s.pop}^{(n)}.\mathsf{top} = \mathsf{t.pop}^{(n)}.\mathsf{top}\}. \tag{2}$$

We check that R satisfies the four requirements.

1. The pair of initial states (new, new) is in R since in Stack$_1(A)$ we get $\mathsf{new.pop}^{(n)}.\mathsf{top} = \mathsf{new.pop}^{(n)}.\mathsf{ar.tell}(\mathsf{new.pop}^{(n)}.\mathsf{end}) = \mathsf{new.ar.tell}(0) = \mathsf{new.tell}(0) = *$. And similarly, in Stack(A) we have $\mathsf{new.pop}^{(n)}.\mathsf{top} = *$, by an easy induction on $n \in \mathbb{N}$.
2. The second requirement $(\mathsf{s}, \mathsf{t}) \in R \Rightarrow \mathsf{s.top} = \mathsf{t.top}$ holds by taking $n = 0$ in R.

3. The third requirement $(\mathsf{s},\mathsf{t}) \in R \Rightarrow (\mathsf{s.pop}, \mathsf{t.pop}) \in R$ holds by definition of R.
4. The fourth requirement $(\mathsf{s},\mathsf{t}) \in R \Rightarrow (\mathsf{s.push}(a), \mathsf{t.push}(a)) \in R$ is most complicated. We shall prove $\mathsf{s.push}(a).\mathsf{pop}^{(n)}.\mathsf{top} = \mathsf{t.push}(a).\mathsf{pop}^{(n)}.\mathsf{top}$ by induction on $n \in \mathbb{N}$. The base case $n = 0$ holds, since

$$\begin{aligned}
\mathsf{s.push}(a).\mathsf{pop}^{(0)}.\mathsf{top} &= \mathsf{s.push}(a).\mathsf{ar.tell}(\mathsf{s.push}(a).\mathsf{end}) \\
&= \mathsf{s.ar.put}(a, \mathsf{s.end}+1).\mathsf{tell}(\mathsf{s.end}+1) \\
&= a \\
&= \mathsf{t.push}(a).\mathsf{pop}^{(0)}.\mathsf{top}.
\end{aligned}$$

For the induction step we compute:

$$\begin{aligned}
\mathsf{s.push}(a).\mathsf{pop}^{(n+1)}.\mathsf{top} &= \mathsf{s.push}(a).\mathsf{pop}^{(n+1)}.\mathsf{ar.tell}(\mathsf{s.push}(a).\mathsf{pop}^{(n+1)}.\mathsf{end}) \\
&= \mathsf{s.push}(a).\mathsf{ar.tell}((\mathsf{s.end}+1) \mathbin{\dot{-}} (n+1)) \\
&= \mathsf{s.ar.put}(a, \mathsf{s.end}+1).\mathsf{tell}(\mathsf{s.end} \mathbin{\dot{-}} n) \\
&= \mathsf{s.ar.tell}(\mathsf{s.end} \mathbin{\dot{-}} n) \\
&= \mathsf{s.pop}^{(n)}.\mathsf{ar.tell}(\mathsf{s.pop}^{(n)}.\mathsf{end}) \\
&= \mathsf{s.pop}^{(n)}.\mathsf{top} \\
&\overset{\text{(IH)}}{=} \mathsf{t.pop}^{(n)}.\mathsf{top} \\
&= \mathsf{t.push}(a).\mathsf{pop}^{(n+1)}.\mathsf{top}.
\end{aligned}$$

Thus we have proved the following result.

4.3. Proposition. *The* Stack(A) *specification in Figure 1 is refined by the* Stack$_1(A)$ *specification in Figure 7, via the relation (2).* □

```
class spec: Queue₁(A)
  methods:
    begin: $X \longrightarrow \mathbb{N}$
    end: $X \longrightarrow \mathbb{N}$
    ar: $X \longrightarrow \mathrm{Array}(A)$
    top: $X \longrightarrow 1 + A$
    push: $X \times A \longrightarrow X$
    pop: $X \longrightarrow X$
  assertions:
    s.begin $\leq$ s.end
    s.push($a$).begin = s.begin
    s.push($a$).end = s.end + 1
  assertions:
    s.push($a$).ar $\underline{\leftrightarrow}$
      s.ar.put($a$, s.end).clear(s.end + 1)
    s.begin $<$ s.end $\vdash$ s.pop.begin = s.begin + 1
    s.begin = s.end $\vdash$ s.pop.begin = s.begin
    s.pop.end = s.end
    s.pop.ar $\underline{\leftrightarrow}$ s.ar
    s.top = s.ar.tell(s.begin)
  creation:
    new.begin = 0
    new.end = 0
    new.ar $\underline{\leftrightarrow}$ new
end class spec
```

Figure 8: The first refinement of queues, using segments in an array with beginning and end

4.4 Queues

We turn to refinement of the queue class specification in Figure 1. We shall do this in two different ways, each time using arrays. In the first refinement we shall describe a queue as a segment in an array, given by two coordinates for beginning and for end. In adding an element to the end of the segment, we increment this end coordinate, and in popping off an element at the front, we increment the begin coordinate. The segment representing the queue thus moves upwards through the array. This will be different in our second refinement, where we keep this segment at the beginning of the array. But we start with the first refinement in Figure 8, which we shall call $\text{Queue}_1(A)$.

4.4. Lemma. *Consider the specification* $\text{Queue}_1(A)$, *and write* $|\mathsf{s}| = \mathsf{s.end} - \mathsf{s.begin}$, *for an arbitrary state* s. *Then*

$$\begin{aligned} \mathsf{s.pop}^{(n)}.\mathsf{begin} &= \min\{\mathsf{s.begin}+n, \mathsf{s.end}\} \\ \mathsf{s.push}(a).\mathsf{pop}^{(n)}.\mathsf{top} &= \begin{cases} \mathsf{s.pop}^{(n)}.\mathsf{top} & \textit{if } n < |\mathsf{s}| \\ a & \textit{if } n = |\mathsf{s}| \\ * & \textit{if } n > |\mathsf{s}|. \end{cases} \end{aligned} \qquad \square$$

4.5. Proposition. *The* $\text{Queue}(A)$ *specification in Figure 1 is refined by the* $\text{Queue}_1(A)$ *specification in Figure 8, via the relation* $R \subseteq \text{Queue}_1(A) \times \text{Queue}(A)$ *given by*

$$\begin{aligned} R = \{(\mathsf{s},\mathsf{t}) \mid (\forall n \in \mathbb{N}\ \mathsf{s.pop}^{(n)}.\mathsf{top} = \mathsf{t.pop}^{(n)}.\mathsf{top}) \wedge |\mathsf{s}| > 0 \\ \wedge\ (\forall n < |\mathsf{s}|\ \mathsf{s.pop}^{(n)}.\mathsf{top} \neq *) \wedge (\forall n \geq |\mathsf{s}|\ \mathsf{s.pop}^{(n)}.\mathsf{top} = *)\}. \end{aligned}$$

Proof. Clearly, if $(\mathsf{s},\mathsf{t}) \in R$, then $\mathsf{s.top} = \mathsf{t.top}$. What remains to show is that R is appropriately closed under the operations. Essentially, this follows from the previous lemma. We shall do part of the push-case. For $(\mathsf{s},\mathsf{t}) \in R$ we need to show that $(\mathsf{s.push}(a), \mathsf{t.push}(a)) \in R$; we concentrate on $\mathsf{s.push}(a).\mathsf{pop}^{(n)}.\mathsf{top} = \mathsf{t.push}(a).\mathsf{pop}^{(n)}.\mathsf{top}$, for all $n \in \mathbb{N}$. The formula for the left hand side occurs in the previous lemma, so we compute the right hand side accordingly (in $\text{Queue}(A)$):

- If $n < |\mathsf{s}|$, then for each $i \leq n$ we have $\mathsf{t.pop}^{(i)}.\mathsf{top} = \mathsf{s.pop}^{(i)}.\mathsf{top} \neq *$. Hence $\mathsf{t.push}(a).\mathsf{pop}^{(n)}.\mathsf{top} = \mathsf{t.pop}^{(n)}.\mathsf{push}(a).\mathsf{top} = \mathsf{t.pop}^{(n)}.\mathsf{top} = \mathsf{s.push}(a).\mathsf{pop}^{(n)}.\mathsf{top}$.

- In case $n = |\mathsf{s}|$ we get $\mathsf{t.pop}^{(i)}.\mathsf{top} \neq *$ for $i < |\mathsf{s}| = n$ and $\mathsf{t.pop}^{(n)}.\mathsf{top} = *$. This yields $\mathsf{t.push}(a).\mathsf{pop}^{(n)}.\mathsf{top} = \mathsf{t.pop}^{(n)}.\mathsf{push}(a).\mathsf{top} = a = \mathsf{s.push}(a).\mathsf{pop}^{(n)}.\mathsf{top}$.

- Finally, if $n > |\mathsf{s}|$, then we get $\mathsf{t.push}(a).\mathsf{pop}^{(n)}.\mathsf{top} = \mathsf{t.pop}^{(|\mathsf{s}|)}.\mathsf{push}(a).\mathsf{pop}^{(n-|\mathsf{s}|)}.\mathsf{top}$ $= \mathsf{t.pop}^{(|\mathsf{s}|)}.\mathsf{pop}^{(n-|\mathsf{s}|-1)}.\mathsf{top} = \mathsf{t.pop}^{(n-1)}.\mathsf{top} = \mathsf{s.pop}^{(n-1)}.\mathsf{top} = *$. And in $\text{Queue}_1(A)$ we also have $\mathsf{s.push}(a).\mathsf{pop}^{(n)}.\mathsf{top} = *$, by the above lemma. $\square$

The disadvantage of this first refinement is that it requires segments with both a beginning and an end. It would be easier to use initial segments with 0 as beginning, so that only an end attribute is needed. Such segments have a fixed place (at the beginning) and do not wander off into infinity (possibly using much memory space).

Using such initial segments with 0 as beginning forces us to shift the whole segment one place forward if we wish to pop off an element. This requires an extra operation on arrays, which we introduce via inheritance, giving us a class specification $\text{ShiftArray}(A)$, see Figure 9. It contains as main operations a shift, which takes an array and a parameter $n \in \mathbb{N}$, and produces a new array in which the first n elements are moved one position forward. Doing so requires an auxiliary procedure aux_shift describing a loop. Lemma 4.6 sums up the main property of the shift method.

4.6. Lemma. *In* $\text{ShiftArray}(A)$ *one has*

$$\begin{aligned} j < n &\vdash \mathsf{s.shift}(n).\mathsf{tell}(j) = \mathsf{s.tell}(j+1) \\ j \geq n &\vdash \mathsf{s.shift}(n).\mathsf{tell}(j) = \mathsf{s.tell}(j). \end{aligned} \qquad \square$$

```
class spec: ShiftArray($A$)
    inherits from: Array($A$)
    methods:
        shift: $X \times \mathbb{N} \longrightarrow X$
        aux_shift: $X \times \mathbb{N} \times \mathbb{N} \longrightarrow X$
    assertions:
        $\mathsf{s.shift}(n) \underline{\leftrightarrow} \mathsf{s.aux\_shift}(0, n)$
        $i < n, \mathsf{s.tell}(i+1) \neq * \vdash$
            $\mathsf{s.aux\_shift}(i, n) = \mathsf{s.put}(\mathsf{s.tell}(i+1), i).\mathsf{aux\_shift}(i+1, n)$
        $i < n, \mathsf{s.tell}(i+1) = * \vdash$
            $\mathsf{s.aux\_shift}(i, n) = \mathsf{s.clear}(i).\mathsf{aux\_shift}(i+1, n)$
        $i \geq n \vdash \mathsf{s.aux\_shift}(i, n) \underline{\leftrightarrow} s$
end class spec
```

Figure 9: Arrays with an additional shift operation

```
class spec: Queue$_2$($A$)
  methods:
    end: $X \longrightarrow \mathbb{N}$
    ar: $X \longrightarrow$ ShiftArray($A$)
    top: $X \longrightarrow 1 + A$
    push: $X \times A \longrightarrow X$
    pop: $X \longrightarrow X$
  assertions:
    $\mathsf{s.push}(a).\mathsf{end} = \mathsf{s.end} + 1$
    $\mathsf{s.push}(a).\mathsf{ar} \underline{\leftrightarrow}$
      $\mathsf{s.ar.put}(a, \mathsf{s.end}).\mathsf{clear}(\mathsf{s.end} + 1)$
    $\mathsf{s.pop.end} = \mathsf{s.end} \dot{-} 1$
    $\mathsf{s.pop.ar} \underline{\leftrightarrow} \mathsf{s.ar.shift}(\mathsf{s.end})$
    $\mathsf{s.top} = \mathsf{s.ar.tell}(0)$
  creation:
    $\mathsf{new.end} = 0$
    $\mathsf{new.ar} \underline{\leftrightarrow} \mathsf{new}$
end class spec
```

Figure 10: The second refinement of queues, using initial segments in an array

Now we turn to the second refinement in Figure 10. It leads to the following result.

4.7. Lemma. (i) *Consider the* Queue$_2(A)$ *specification (in Figure 10), and let* s *be a state satisfying* $\mathsf{s.ar.tell}(m) = *$ *for* $m \geq \mathsf{s.end}$. *Then*

$$\mathsf{s.pop}^{(n)}.\mathsf{ar.tell}(m) = \begin{cases} \mathsf{s.ar.tell}(n+m) & \textit{if } n+m < \mathsf{s.end} \\ * & \textit{otherwise.} \end{cases}$$

(ii) *Let* s *satisfy the same assumption as in (i). Then*

$$\mathsf{s.push}(a).\mathsf{pop}^{(n)}.\mathsf{top} = \begin{cases} \mathsf{s.pop}^{(n)}.\mathsf{top} & \textit{if } n < \mathsf{s.end} \\ a & \textit{if } n = \mathsf{s.end} \\ * & \textit{otherwise.} \end{cases}$$ □

4.8. Proposition. *The* Queue$_2(A)$ *specification in Figure 10 (also) refines the* Queue(A) *specification in Figure 1, via the relation* $R \subseteq \text{Queue}_2(A) \times \text{Queue}(A)$ *given by*

$$R = \{(\mathsf{s}, \mathsf{t}) \mid (\forall n \in \mathbb{N}\ \mathsf{s.pop}^{(n)}.\mathsf{top} = \mathsf{t.pop}^{(n)}.\mathsf{top}) \wedge (\forall n \geq \mathsf{s.end}\ \mathsf{s.ar.tell}(n) = *) \\ \wedge (\forall n \geq \mathsf{s.end}\ \mathsf{s.ar.tell}(n) \neq *)\}.$$

Proof. Obviously, if $(\mathsf{s},\mathsf{t}) \in R$, then $\mathsf{s.top} = \mathsf{t.top}$. The pair $(\mathsf{new},\mathsf{new})$ of initial states is in R because the initial state new in $\text{Queue}_2(A)$ satisfies $\mathsf{new.ar.tell}(m) = *$, for all $m \geq 0 = \mathsf{new.end}$. Hence $\mathsf{new.pop}^{(n)}.\mathsf{top} = *$, by Lemma 4.7 (i). Closure of R under pop and push is easy, using Lemma 4.7 (and the formulation of $\mathsf{t.push}(a).\mathsf{pop}^{(n)}.\mathsf{top}$ in the proof of Proposition 4.5). □

The two requirements $\forall n \geq \mathsf{s.end}\ \mathsf{s.ar.tell}(n) = *$ and $\forall n \geq \mathsf{s.end}\ \mathsf{s.ar.tell}(n) \neq *$ in the definition of R may be understood as an *invariant* for the specification $\text{Queue}_2(A)$. Similar invariants are part of the definition of the refinement relation R in Proposition 4.5.

5 Behaviour-refinement versus model-refinement

Our notion of refinement (in Definition 3.1) is based on simulation of behaviour, as is usual for automata. There is an important alternative approach which is based on models (especially on hidden-sorted algebras), see e.g. [9, 4, 8, 2, 6, 7, 18]. It defines a concrete specification $\mathcal{C}$ to be a refinement of an abstract specification $\mathcal{A}$ if all models of $\mathcal{A}$, after appropriate restriction, are also models of $\mathcal{C}$. We add two comments. This "appropriate restriction" corresponds in our approach to the effect of the selection functions in Definition 3.1. And a model of a specification may be taken in a behavioural sense, which means that the equations are required to hold only with respect to contexts of observable sort. This leads to "context induction" as a proof-technique, see [8], but also to coinduction, see [7]. We shall refer to this notion as "model-refinement" in contrast to "behaviour-refinement" as used in this paper.

Our aim in this section is to briefly illustrate the difference between model-refinement and behaviour-refinement via an example. This example involves a concrete specification which is a behaviour-refinement, but not a model-refinement, of an abstract specification. The difference arises because in behaviour-refinement one only considers reachable states. Of course, this difference disappears if one restricts oneself to reachable states (as is often done).

We define an abstract coalgebraic class specification $\mathcal{A}$ with one attribute $\mathsf{val}\colon X \longrightarrow \{0,1\}$ satisfying $\mathsf{s.val} = 1$. And a concrete class specification $\mathcal{C}$ with two attributes $\mathsf{val}\colon X \longrightarrow \{0,1\}$, $\mathsf{count}\colon X \longrightarrow \mathbb{N}$ and one procedure $\mathsf{next}\colon X \longrightarrow X$, with four conditional equations: $\mathsf{s.count} \leq 10 \vdash \mathsf{s.next.count} = \min\{\mathsf{s.count}+1, 10\}$, $\mathsf{s.count} > 10 \vdash \mathsf{s.next.count} = \mathsf{s.count} + 1$, $\mathsf{s.count} \leq 10 \vdash \mathsf{s.val} = 1$, $\mathsf{s.count} > 10 \vdash \mathsf{s.val} = 0$ with initial state $\mathsf{new.count} = 0$. Then $\mathcal{C}$ is a behaviour-refinement of $\mathcal{A}$, but not a model-refinement of $\mathcal{A}$. The first is easy to see, via the relation $R \subseteq \mathcal{C} \times \mathcal{A}$ with $R(\mathsf{s},\mathsf{t})$ given by $\mathsf{s.val} = \mathsf{t.val}$. But $\mathcal{C}$ is not a model-refinement of $\mathcal{A}$. Consider the model of $\mathcal{C}$ consisting of state space $\mathbb{N}$ with operations $\mathsf{val}\colon \mathbb{N} \to \{0,1\}$ given by $\mathsf{val}(x) = 1$ for $x \leq 10$ and $\mathsf{val}(x) = 0$ for $x > 10$, $\mathsf{count}\colon \mathbb{N} \to \mathbb{N}$ by $\mathsf{count}(x) = x$, and $\mathsf{next}\colon \mathbb{N} \to \mathbb{N}$ given by $\mathsf{next}(x) = x$ if $x = 10$ and $\mathsf{next}(x) = x + 1$ otherwise. This clearly forms a model of $\mathcal{C}$. But it does not form a model of $\mathcal{A}$, since the required equation $\mathsf{val}(x) = 1$ does not hold for all $x \in \mathbb{N}$. (But it does hold for all reachable $x \leq 10$.)

References

1. M. Bidoit and R. Hennicker. Proving the correctness of behavioural implementations. In V.S. Alagar and M. Nivat, editors, *Algebraic Methods and Software Technology*, number 936 in Lect. Notes Comp. Sci., pages 152–168. Springer, Berlin, 1995.
2. M. Bidoit, R. Hennicker, and M. Wirsing. Behavioural and abstractor specifications. *Science of Comput. Progr.*, 25:149–186, 1995.
3. M. Broy. Specification and refinement of a buffer of length one. Marktoberdorf Summerschool, 1994.

4. J.A. Goguen. An algebraic approach to refinement. In D. Bjørner, C.A.R. Hoare, and H. Langmaack, editors, *VDM '90. VDM and Z—Formal Methods in Software Development*, number 428 in Lect. Notes Comp. Sci., pages 12–28. Springer, Berlin, 1990.
5. J.A. Goguen and R. Diaconescu. Towards an algebraic semantics for the object paradigm. In H. Ehrig and F. Orejas, editors, *Recent Trends in Data Type Specification*, number 785 in Lect. Notes Comp. Sci., pages 1–29. Springer, Berlin, 1994.
6. J.A. Goguen and G. Malcom. Proof of correctness of object representations. In A.W. Roscoe, editor, *A Classical Mind. Essays in honour of C.A.R. Hoare*, pages 119–142. Prentice Hall, 1994.
7. J.A. Goguen and G. Malcom. An extended abstract of a hidden agenda. In J., A. Meystel, and R. Quintero, editors, *Proceedings of the Conference on Intelligent Systems: A Semiotic Perspective*, pages 159–167. Nat. Inst. Stand. & Techn., 1996.
8. R. Hennicker. Context induction: a proof principle for behavioural abstractions and algebraic implementations. *Formal Aspects of Comp.*, 3(4):326–345, 1991.
9. C.A.R. Hoare. Proof of correctness of data representations. *Acta Informatica*, 1:271–281, 1972.
10. B. Jacobs. Mongruences and cofree coalgebras. In V.S. Alagar and M. Nivat, editors, *Algebraic Methods and Software Technology*, number 936 in Lect. Notes Comp. Sci., pages 245–260. Springer, Berlin, 1995.
11. B. Jacobs. Automata and behaviours in categories of processes. CWI Techn. Rep. CS-R9607, 1996.
12. B. Jacobs. Coalgebraic specifications and models of deterministic hybrid systems. In M. Wirsing and M. Nivat, editors, *Algebraic Methods and Software Technology*, number 1101 in Lect. Notes Comp. Sci., pages 520–535. Springer, Berlin, 1996.
13. B. Jacobs. Inheritance and cofree constructions. In P. Cointe, editor, *European Conference on Object-Oriented Programming*. number 1098 in Lect. Notes Comp. Sci., pages 210–231. Springer, Berlin, 1996.
14. B. Jacobs. Objects and classes, co-algebraically. In B. Freitag, C.B. Jones, C. Lengauer, and H.-J. Schek, editors, *Object-Orientation with Parallelism and Persistence*, pages 83–103. Kluwer Acad. Publ., 1996.
15. P. Lucas. Two constructive realizations of the block concept and their equivalence. Technical Report 25.085, IBM Laboratory, Vienna, 1968.
16. N. Lynch and F. Vaandrager. Forward and backward simulations. I. Untimed systems. *Inf. & Comp.*, 121(2):214–233, 1995.
17. N.A. Lynch and M.R. Tuttle. An introduction to input/output automata. *CWI Quarterly*, 2(3):219–246, 1989.
18. G. Malcolm and J.A. Goguen. Proving correctness of refinement and implementation. Techn. Monogr. PRG 114, Oxford Univ., 1996.
19. B. Meyer. *Object-Oriented Software Construction*. Prentice Hall, 1988.
20. R. Milner. An algebraic definition of simulation between programs. In *Sec. Int. Joint Conf. on Artificial Intelligence*, pages 481–489. British Comp. Soc. Press, London, 1971.
21. S. Owre, S. Rajan, J.M. Rushby, N. Shankar, and M. Srivas. PVS: Combining specification, proof checking, and model checking. In R. Alur and T.A. Henzinger, editors, *Computer Aided Verification*, number 1102 in Lect. Notes Comp. Sci., pages 411–414. Springer, Berlin, 1996.
22. H. Reichel. An approach to object semantics based on terminal co-algebras. *Math. Struct. Comp. Sci.*, 5:129–152, 1995.
23. B. Rumpe and C. Klein. Automata describing object behaviour. In H. Kilov and W. Harvey, editors, *Specification of Behavioral Semantics in Object-Oriented Information modeling*, pages 265–286. Kluwer Acad. Publ., 1996.
24. J. Rutten and D. Turi. On the foundations of final semantics: non-standard sets, metric spaces and partial orders. In J.W. de Bakker, W.P. de Roever, and G. Rozenberg, editors, *Semantics: Foundations and Applications*, number 666 in Lect. Notes Comp. Sci., pages 477–530. Springer, Berlin, 1993.
25. O. Schoett. Behavioural correctness of data representations. *Science of Comput. Progr.*, 14:43–57, 1990.