Electronic Notes in Theoretical Computer Science 47 (2001)
URL: http://www.elsevier.nl/locate/entcs/volume47.html pp. 1 – 36

Assertional and Behavioural Refinement in Coalgebraic Specification

Bart Jacobs [1]

Department of Computer Science
University Nijmegen
P.O. Box 9010, 6500 GL Nijmegen, The Netherlands.

Hendrik Tews [2]

Institut für Theoretische Informatik
TU Dresden
D-01062 Dresden, Germany.

Abstract

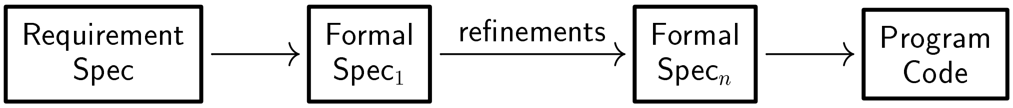
Refinements between specifications in a coalgebraic setting have been introduced in [13,14]. This paper presents a renewed, extended study of refinements, based on more practical experience. It distinguishes assertional refinement (assertions should be valid, after translation) and behavioural refinement (appropriate behaviour must be simulated, after translation). Behavioural refinement is the more general notion, it offers greater flexibility, especially in the presence of non-public methods. Assertional refinement is shown to be a special case of behavioural refinement. The (coalgebraic) notions of invariant and bisimulation will be essential in these investigations. In the end, refinement in a coalgebraic setting is illustrated via two refinements of an abstract transaction-commit mechanism.

[1] Email: bart@cs.kun.nl
[2] Email: tews@tcs.inf.tu-dresden.de

Contents

1 Introduction

Refinement is one of the most important concepts in a formal approach to software. It involves a systematic trajectory for software development, starting from a general (requirement) specification, leading to a concrete implementation in a given programming language, as suggested in:



Here we are interested in the arrow in the middle, describing (multiple) refinement steps between formal specifications. The precise nature of the first and the last arrow is left unexplained. Certainly, the idea is that the first formal specification should be so close to the (informal) requirement specification that the transition is "obvious". Similarly, the last formal specification should be so concrete that actually producing a corresponding program is "obvious". However, in practice these first and last steps are not so clear.

We shall be working within the context of so-called coalgebraic specification [15,26]. This formalism is particularly suited for the abstract specification of classes in object-oriented languages, in functional style. A key feature is that the specification describes a state space, commonly called Self, which is considered as a black box. This set of states Self carries a number of coalgebraic operations, which are usually called methods in this context. The methods are of the form $\mathsf{Self} \longrightarrow \boxed{\cdots \mathsf{Self} \cdots}$ with structured output types—as indicated by the box. These methods can be used for observation and modification, and thus give a certain amount of information about states. The behaviour of the methods can be constrained in specifications via assertions. Additionally, the specification can require an initial state, usually written as new, which is constrained by its own assertions.

Within the theory of coalgebras there are general notions of invariant, bisimulation, henceforth $\Box$ and eventually $\Diamond$, which are defined in a generic way. That is, they are defined for each (suitable) form of output $\boxed{\cdots \mathsf{Self} \cdots}$. In particular, for each class specification there are these tailor-made notions. This makes the theory very flexible and powerful, for example, because the temporal operators $\Box$ and $\Diamond$ may be used in the assertions in a specification—with the appropriate meaning in that particular specification, see the definition of $\Box$ and $\Diamond$ in Section 6.

Early on in the development of coalgebraic specifications, refinements have been defined, see [13,14], making use of invariants and bisimulations. This notion of refinement has been used since, mainly in the context of the experimental formal specification language CCSL [26] (for Coalgebraic Class Specification Language, see Section 6), with its translation to the theorem provers PVS [24] and ISABELLE [25]. The practical experience showed the need for generalisations of the original notion, which will be illustrated in Subsections 4.1 – 4.4 below. Also, it became clear that it makes sense to distinguish "assertional" and "behavioural" refinement.

In general, a refinement involves two specifications, an "abstract" one and its refinement, a "concrete" one. The idea is that the concrete specification adds implementation details which are left open in the abstract specification. For example, the concrete specification can lay down how computations should be done (where the abstract specification may only say what should be computed). Alternatively it can reduce the level of underspecification / non-determinism. In relating the concrete and abstract specification, one must be able to express the (public) methods of the abstract class in terms of the methods of the concrete class. That is, there must be a translation of "signatures" from concrete to abstract. Via such a translation of signatures one can translate models of the concrete specification into "possible models" of the abstract specification. We say "possible models" because the translated structures provide interpretations for all the abstract (public) methods, but it need not be the case that the abstract assertions hold. At this stage the difference between assertional and behavioural refinement appears.

In an assertional refinement one requires that the assertions of the abstract specification should hold for each translated model of the concrete specification. Essentially this means that refinement is implication, or inclusion of models. This view is used in data / algebra refinement (explored by Hoare, Goguen, Tarlecki, Sannella, Wirsing ..., see *e.g.* [12,2,29]), in program refinement (by Back [3]), and also in the Temporal Logic of Actions (TLA) of Lamport [17].

One problem with this assertional refinement is that the abstract specification may involve certain private methods, and that its assertions are formulated in terms of these private methods. Clearly, the translation of signatures from concrete to abstract need not to cover the private methods, therefore it might be impossible to determine if the assertions hold after the translation.

In behavioural refinement one therefore requires that each translated model of the concrete specification should be *behaviourally equal* (or, synonymously, *observationally indistinguishable*) to some abstract model. This indistinguishability is expressed in terms of bisimilarity. This approach of defining refinement via possible observations may be found in refinement of actions (investigated by Gorrieri, Degano, Hennessey, van Glabbeek ..., see [7]) or of automata / transition systems (by Lynch & Vaandrager [19]).

This paper will introduce both notions of refinement in a coalgebraic setting, and will establish their relationship: assertional refinement is a special case of behavioural refinement. In addition, actual refinement in CCSL is illustrated in a non-trivial example involving a transaction-commit mechanism, used to make sequences of actions atomic, for example in the context of data bases, or smart cards.

The paper is structured as follows: The next section defines those coalgebraic notions, that are relevant for this paper. Section 3 gives a short introduction into coalgebraic specification. Section 4 discusses several generalisations of assertional refinement. Behavioural refinement is in Section 5. The experimental specification language CCSL (Coalgebraic Class Specification Language) supports our approach to coalgebraic specification and refinement. Section 6 presents CCSL and discusses the modal operators $\Box$ and $\Diamond$ in more detail. Section 7 shows a non-trivial application of CCSL: The refinement of a general transaction-commit mechanism. Finally, in Section 8, we conclude.

2 Coalgebras for Polynomial Functors

This section reviews polynomial functors and coalgebras for polynomial functors with the associated definitions of invariant and bisimulation. For a more complete account on coalgebras see [27]. The notion of a functor comes from category theory [20]. In this paper it is used to describe coalgebraic signatures. The class of **polynomial functors** on the category **Set** of sets and total functions are those endofunctors that are finitely generated from the following grammar:

$$\mathcal{T}(X) \quad = \quad A \mid X \mid \mathcal{T}_1(X) \times \mathcal{T}_2(X) \mid \mathcal{T}_1(X) + \mathcal{T}_2(X) \mid A \Rightarrow \mathcal{T}(X)$$

Here A is an arbitrary constant set, $\times$ denotes the cartesian product, $+$ the disjoint union, and $A \Rightarrow \mathcal{T}(X)$ stands for the function space (with constant domain). We restrict ourselves to polynomial functors in this paper, because they are expressive enough for many problems. All interesting aspects of refinement can be explained with polynomial functors.

Let $\mathcal{T}$ be an arbitrary polynomial functor in what follows. A $\mathcal{T}$–**coalgebra** is a function $X \longrightarrow \mathcal{T}(X)$. In this case X is the **state space** of the coalgebra. Let $c : X \longrightarrow \mathcal{T}(X)$ and $d : Y \longrightarrow \mathcal{T}(Y)$ be two $\mathcal{T}$–coalgebras. A $\mathcal{T}$–**coalgebra morphism** $c \longrightarrow d$ (*i.e.*, a structure preserving map from c to d) is a function $f : X \longrightarrow Y$ that makes the following diagram commute:

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

The $\mathcal{T}$–coalgebras and $\mathcal{T}$–coalgebra morphisms from a category.

(For symmetry we note the following: A $\mathcal{T}$–algebra is a function $a : \mathcal{T}(X) \longrightarrow X$. If $b : \mathcal{T}(Y) \longrightarrow Y$ is another $\mathcal{T}$–algebra then a function $f : X \longrightarrow Y$ is a $\mathcal{T}$–algebra morphism precisely if $f \circ a = b \circ \mathcal{T}(f)$. An example is the data type of finite lists over a set A, which is an algebra (actually the initial one) for the functor $\mathcal{T}(X) = A \times X \ + \ \mathbf{1}$.)

An important notion for coalgebras is that of *behavioural equality*. Formally behavioural equality is captured by the notion of *bisimulation*. This is a generalisation of the familiar notion of (strong) bisimulation for transition systems [22]. Another important notion is that of an *invariant*. Invariants are predicates on the state space of a coalgebra. Intuitively invariants correspond to safety properties: Once an invariant holds for a state x, it continues to hold for all successor states of x. Formally invariants and bisimulations are defined via predicate and relation lifting, respectively.

Definition 2.1 Let $\mathcal{T}$ be a polynomial functor. The **predicate lifting** of $\mathcal{T}$, denoted with $\mathrm{Pred}(\mathcal{T})$, is an operation on predicates. The **relation lifting** of $\mathcal{T}$, denoted with $\mathrm{Rel}(\mathcal{T})$, is an operation on relations. More precisely, for a predicate $P \subseteq X$, and a relation $R \subseteq X \times Y$ we have

$$\mathrm{Pred}(\mathcal{T}) : \ (P \subseteq X) \ \longmapsto \ (\mathrm{Pred}(\mathcal{T})(P) \ \subseteq \ \mathcal{T}(X))$$

$$\mathrm{Rel}(\mathcal{T}) : \ (R \subseteq X \times Y) \ \longmapsto \ (\mathrm{Rel}(\mathcal{T})(R) \ \subseteq \ \mathcal{T}(X) \times \mathcal{T}(Y))$$

Predicate and relation lifting are defined by induction on the structure of $\mathcal{T}$:

- If $\mathcal{T}(X) = X$, then

$$\begin{array}{rcl} \mathrm{Pred}(\mathcal{T})(P) & = & P \\ \mathrm{Rel}(\mathcal{T})(R) & = & R \end{array}$$

- If $\mathcal{T}(X) = A$ (for A a constant set), then

$$\begin{array}{rcl}\mathrm{Pred}(\mathcal{T})(P) & = & (A \subseteq A) \\ \mathrm{Rel}(\mathcal{T})(R) & = & \big(\{(a,a) \mid a \in A\} \subseteq A \times A\big)\end{array}$$

- If $\mathcal{T}(X) = \mathcal{T}_1(X) \times \mathcal{T}_2(X)$, then

$$\begin{array}{rcl}\mathrm{Pred}(\mathcal{T})(P) & = & \{(x,y) \mid \mathrm{Pred}(\mathcal{T}_1)(P)(x) \text{ and } \mathrm{Pred}(\mathcal{T}_2)(P)(y)\} \\ \mathrm{Rel}(\mathcal{T})(R) & = & \{((x_1,x_2),(y_1,y_2)) \mid \mathrm{Rel}(\mathcal{T}_1)(R)(x_1,y_1) \\ & & \qquad \text{and } \mathrm{Rel}(\mathcal{T}_2)(R)(x_2,y_2)\}\end{array}$$

- If $\mathcal{T}(X) = \mathcal{T}_1(X) + \mathcal{T}_2(X)$, then

$$\begin{array}{rcl}\mathrm{Pred}(\mathcal{T})(P) & = & \{\kappa_1\, x \mid \mathrm{Pred}(\mathcal{T}_1)(P)(x)\} \cup \{\kappa_2\, y \mid \mathrm{Pred}(\mathcal{T}_2)(P)(y)\} \\ \mathrm{Rel}(\mathcal{T})(R) & = & \{(\kappa_1\, x_1, \kappa_1\, y_1) \mid \mathrm{Rel}(\mathcal{T}_1)(R)(x_1,y_1)\} \\ & & \quad \cup\ \{(\kappa_2\, x_2, \kappa_2\, y_2) \mid \mathrm{Rel}(\mathcal{T}_2)(R)(x_2,y_2)\}\end{array}$$

  where κ_1, κ_2 are the "coprojections" $A \xrightarrow{\kappa_1} A + B \xleftarrow{\kappa_2} B$ used for disjoint unions / coproducts $+$.
- If $\mathcal{T}(X) = A \Rightarrow \mathcal{T}_1(X)$, then

$$\begin{array}{rcl}\mathrm{Pred}(\mathcal{T})(P) & = & \big\{f \mid f : A \longrightarrow \mathcal{T}_1(X) \text{ with } \forall a \in A \,.\, \mathrm{Pred}(\mathcal{T}_1)(P)(f(a))\big\} \\ \mathrm{Rel}(\mathcal{T})(R) & = & \big\{(f,g) \mid f : A \longrightarrow \mathcal{T}_1(X),\ g : A \longrightarrow \mathcal{T}_1(Y) \\ & & \qquad \text{such that } \forall a \in A \,.\, \mathrm{Rel}(\mathcal{T}_1)(R)(f(a), g(a))\ \big\}\end{array}$$

Predicate and relation lifting work by substituting the cartesian closed structure in $\mathcal{T}$ with appropriate operations on predicates and relations. The effect is as follows: If one thinks of an element $u \in \mathcal{T}(X)$ as a structured box containing constants and elements of X, then $\mathrm{Pred}(\mathcal{T})(P)(u)$ holds if P holds for all the elements of X in u. Relation lifting relates two boxes $u \in \mathcal{T}(X)$ and $v \in \mathcal{T}(Y)$. The pair (u,v) is in the relation $\mathrm{Rel}(\mathcal{T})(R)$ if both u and v have the same structure and, additionally, constants in u and v at corresponding places are equal and elements of X and Y in u and v at corresponding places are related by R.

Definition 2.2 Consider two coalgebras $c : X \longrightarrow \mathcal{T}(X)$ and $d : Y \longrightarrow \mathcal{T}(Y)$ for the same polynomial functor $\mathcal{T}$.

- A predicate $P \subseteq X$ is an **invariant** (for c) if for all $x \in X$

$$P(x) \quad \text{implies} \quad \mathrm{Pred}(\mathcal{T})(P)(c\,x).$$

- A relation $R \subseteq X \times Y$ is called a **bisimulation** (for c and d) if for all $x \in X$ and $y \in Y$,

$$R(x,y) \quad \text{implies} \quad \mathrm{Rel}(\mathcal{T})(R)(c\,x,\, d\,y).$$

The preceding definition follows [11]. Many authors (*e.g.* [27]) follow Aczel and Mendler in [1] and define a bisimulation as a relation R carrying a coalgebra structure such that the projections $X \xleftarrow{\pi_1} R \xrightarrow{\pi_2} Y$ are coalgebra morphisms. For polynomial functors both approaches yield equivalent notions of bisimulation.

Example 2.3 The canonical example for coalgebras is the type of (possibly infinite) sequences over a set (or alphabet) A (the example has been fully worked out in [10]). The signature of sequences is captured by the functor $\mathcal{T}_{\mathsf{Seq}}(X) = (A \times X) + \mathbf{1}$. If $\mathsf{next} : X \longrightarrow \mathcal{T}_{\mathsf{Seq}}(X)$ is a sequence coalgebra and $s \in X$ is a sequence then either $\mathsf{next}(s) = \kappa_2 *$, where $*$ is the only element of $\mathbf{1}$ or $\mathsf{next}\, s = \kappa_1(a, s')$. In the former case the sequence s is empty, in the latter it can be decomposed into its head element a and the sequence s' of the remaining elements.

Assume another sequence coalgebra $\mathsf{next}' : Y \longrightarrow \mathcal{T}_{\mathsf{Seq}}(Y)$. Let $P \subseteq X$ be a predicate on the state space of next and let $R \subseteq X \times Y$ be a relation on the state spaces of next and next'. The predicate and relation lifting for P and R is as follows:

$$\begin{array}{rcl} \mathrm{Pred}(T)(P) & = & \{\kappa_1(a,s) \mid a \in A \text{ and } s \in P\} \cup \{\kappa_2 *\} \\ \mathrm{Rel}(T)(R) & = & \{(\kappa_1(a,s), \kappa_1(b,r)) \mid a, b \in A \wedge a = b \wedge (s,r) \in R\} \\ & & \cup\, \{(\kappa_2 *, \kappa_2 *)\} \end{array}$$

Thus $\mathrm{Pred}(T)(P)$ holds on $\mathsf{next}(s)$ if either s is the empty sequence or the remainder of s is in P. And $(\mathsf{next}(s),\ \mathsf{next}'(r))$ is in $\mathrm{Rel}(T)(R)$ if either *both* s and r are empty or *both* have the same head a and the remainders are again in R.

A relation R is a sequence bisimulation for next and next' in case

$$s\, R\, r \quad \text{implies} \quad \begin{cases} \mathsf{next}(s) = \mathsf{next}'(r) = \kappa_2\ * & \text{or} \\ \mathsf{next}(s) = \kappa_1(a, s') \wedge \mathsf{next}'(r) = \kappa_1(b, s') \wedge a = b \wedge s'\, R\, r' \end{cases}$$

for all possible s and r. Similarly, a predicate P is an invariant if

$$s \in P \qquad \text{implies} \qquad s' \in P \ \text{ if } \ \mathsf{next}(s) = \kappa_1(a, s')$$

An example of a sequence invariant is the predicate P_∞ that holds precisely for the infinite sequences in X.

In this paper we need the following properties of bisimulations and invariants.

Lemma 2.4 *Let $\mathcal{T}$ be a polynomial functor and let $c : X \longrightarrow \mathcal{T}(X)$, $d : Y \longrightarrow \mathcal{T}(Y)$, and $e : Z \longrightarrow \mathcal{T}(Z)$ be $\mathcal{T}$–coalgebras.*

(i) *All bisimulations for c and d, ordered by inclusion, form a complete lattice, with unions as join. In particular there exists a greatest bisimulation, which is called bisimilarity and denoted with ${}_c\!\underline{\leftrightarrow}_d$.*

(ii) *Bisimulations are closed under relational composition: If R is a bisimulation for c and d and S is a bisimulations for d and e, then $R \circ S$ is a bisimulation for c and e.*

(iii) *Bisimilarity on a single coalgebra c, denoted with ${}_c\underline{\leftrightarrow}_c$, is an equivalence relation (on the state space of c).*

(iv) *The invariants for c, ordered by inclusion, form a complete lattice, with unions as join. In particular for any predicate $P \subseteq X$ there exists a greatest invariant contained in P and a least invariant containing P.*

(v) *If $P \subseteq X$ is an invariant for c, then the restriction of c to P, denoted with $c|_P$ is a coalgebra $P \longrightarrow \mathcal{T}(P)$.*

(vi) *A function $f : X \longrightarrow Y$ is a morphism of coalgebras if and only if its graph $\{(x, y) \mid f(x) = y\}$ is a bisimulation.*

(vii) *Let $R \subseteq X \times Y$ be a bisimulation for c and d. Then the predicate $\{x \mid \exists y \in Y \,.\, R(x, y)\} \subseteq X$ is an invariant for c.*

Proof. [Sketch] Prove that predicate and relation lifting commutes with conjunction and disjunction of predicates and relations, respectively. This proves Item (i) and Item (iv). For Item (ii) and (iii) show that equality is a bisimulation and that relation lifting commutes with relational composition and with taking the opposite relation. The remaining items are obtained by similar basic properties of predicate and relation lifting. □

3 Coalgebraic Specifications

This section gives a brief introduction to coalgebraic specification, mainly via examples. A more thorough account may be found in [15] and [26]. At this stage we use *ad hoc* notation. Later, in Section 6 a formal language CCSL for such specifications will be described. CCSL is used in the examples in Section 7.

Example 3.1 In this illustration of a coalgebraic specification we use the set $2 = \{0, 1\}$ of bits, or Boolean values, with negation operation $\neg\colon 2 \to 2$ given by $\neg b = 1 - b$. We wish to specify a simple negator system with an unit delay: the result of a negation request only becomes visible after a second request. The specification of the system is given in Figure 1 in CCSL–like syntax. It has two "public" methods for accessing the value and for the negation operation. For the intermediate value the specification declares a "private" auxiliary operation. For the time being we shall ignore the difference between private and ordinary methods; it only becomes relevant in Section 5. The formulae in the specification restrict the possible behaviour of the methods and the initial state new. Note, that the specification does not allow to derive the result of $\mathsf{val}(\mathsf{neg}(x, b))$ for arbitrary x. Only if we know that $x = \mathsf{neg}(y, b')$, then we can derive $\mathsf{val}(\mathsf{neg}(x, b)) = \neg b'$.

The main idea underlying coalgebraic specifications is to have coalgebras as

Begin DelayAuxNeg : **ClassSpec**
  Method
    $\mathsf{val}\colon \mathsf{Self} \longrightarrow 2$
    $\mathsf{neg}\colon \mathsf{Self} \times 2 \longrightarrow \mathsf{Self}$
  Private Method
    $\mathsf{aux}\colon \mathsf{Self} \longrightarrow 2$
  Assertion
    $\forall b \in 2.\ \mathsf{aux}(\mathsf{neg}(x, b)) = \neg b$
    $\forall b \in 2.\ \mathsf{val}(\mathsf{neg}(x, b)) = \mathsf{aux}(x)$
  Creation
    $\mathsf{aux}(\mathsf{new}) = 0$
    $\mathsf{val}(\mathsf{new}) = 0$
End DelayAuxNeg

Fig. 1. Coalgebraic specification of the negator system, see Example 3.1

methods. These methods themselves can be combined into a single coalgebra, of the form $c\colon \mathsf{Self} \to \mathcal{T}(\mathsf{Self})$, where, in this example, $\mathcal{T}(\mathsf{Self}) = 2 \times \mathsf{Self}^2 \times 2$ and $c = \langle \mathsf{val}, \mathsf{neg}, \mathsf{aux} \rangle$. The functor or type $\mathcal{T}$ describes the coalgebraic signature (or class interface) of the DelayAuxNeg specification. The types $\mathcal{T}$ that we allow are called "polynomial", because they are built up from simple constituents, see Section 2.

The two assertions that follow the keyword **Assertion** can be combined via conjunction into a single assertion $E = E(c)$, involving a coalgebra symbol c. This E also involves a special variable $x\colon \mathsf{Self}$, and thus forms a predicate on Self. Formally, in the example:

$$\begin{aligned} E(c)(x) = {} & \big(\forall b \in 2.\ (\pi_3 \circ c)((\pi_2 \circ c)(x)(b)) = \neg b\big) \quad \wedge \\ & \big(\forall b \in 2.\ (\pi_1 \circ c)((\pi_2 \circ c)(x)(b)) = (\pi_3 \circ c)(x)\big) \end{aligned}$$

where the projections $\pi_i \circ c$ extract the appropriate methods from the coalgebra c. Similarly, the two creation conditions following **Creation** describe an assertion $I = I(c)$ involving a special symbol new for an initial state. Hence it also forms a predicate on Self.

For the topic of the present paper—refinement—the actual logic that is used in the assertions and creation conditions is not so important. Therefore we only sketch the logic and spare the reader the technical details (they can be found in [26]). Our logic is a variant of first-order equational logic. For a polynomial type $\mathcal{T}$ (representing a coalgebraic signature) the **terms** over $\mathcal{T}$ are built as usual from constants, lambda abstraction, application, tuples, (co)projections, and case distinction. To allow methods in the terms we assume a special $\mathcal{T}$-coalgebra symbol $c\colon \mathsf{Self} \to \mathcal{T}(\mathsf{Self})$ as a primitive term over $\mathcal{T}$. Atomic **formulae** are equalities $t_1 = t_2$ and for terms of type Self also bisimilarities $t_1 \underline{\leftrightarrow} t_2$. Formulae are closed under the usual connectives and quantifiers. For the semantics of the formulae one interprets the symbol c with the coalgebra of the model and $\underline{\leftrightarrow}$ with the bisimilarity relation ${}_c\underline{\leftrightarrow}_c$.

A formula is called an **assertion**, if it contains precisely one free variable $x : \mathsf{Self}$. Thus the semantics of an assertion is a predicate on the state space of the model (holding for those states that fulfil the assertion). A formula is called a **creation condition**, if it contains one free variable $\mathsf{new} : \mathsf{Self}$. In Section 5 we assume that assertions and creation conditions respect bisimilarity, in the sense that if two states are bisimilar, then they fulfil the same set of formulae. This can be enforced via suitable syntactic restrictions.

Definition 3.2 A **coalgebraic specification** is a triple $\mathfrak{C} = (\mathcal{T}, E, I)$ where $\mathcal{T} = \mathcal{T}(\mathsf{Self})$ is a polynomial type / functor involving one special type symbol Self, and where E is an assertion and I is a creation condition over $\mathcal{T}$.

For a concrete coalgebra $d\colon X \to \mathcal{T}(X)$ and element $x \in X$, we write:

- $d, x \models E$ if x is in the interpretation $[\![E]\!](d) \subseteq X$ of E in which d is used as interpretation of the coalgebra symbol c in $E = E(c)$;
- $d \models E$ if $d, x \models E$ for all $x \in X$. This will also be written as $d\colon X \stackrel{E}{\longrightarrow} \mathcal{T}(X)$.

Similarly for I instead of E.

A **model** of a coalgebraic specification $(\mathcal{T}, E, I)$ consists of a coalgebra $d\colon X \longrightarrow \mathcal{T}(X)$ with an initial state $x_0 \in X$ such that $d \models E$ and $d, x_0 \models I$. Thus a model is an element of the disjoint union:

$$\coprod_{d\colon X \stackrel{E}{\longrightarrow} \mathcal{T}(X)} \{x \in X \mid d, x \models I\} \quad \text{or, more briefly:} \quad \coprod_{d\colon X \stackrel{E}{\longrightarrow} \mathcal{T}(X)} I(d).$$

Example 3.3 As a model of the delayed negator specification from Example 3.1 one can take the set $X = 2 \times 2$ as state space with the initial state $(0, 0) \in X$ and the coalgebra $\langle \mathsf{val}, \mathsf{neg}, \mathsf{aux} \rangle \colon X \to 2 \times X^2 \times 2$ given by:

$$\begin{array}{ccc} X \stackrel{\mathsf{val}}{\longrightarrow} 2 & X \times 2 \stackrel{\mathsf{neg}}{\longrightarrow} X & X \stackrel{\mathsf{aux}}{\longrightarrow} 2 \\ (b_1, b_2) \longmapsto b_1 & ((b_1, b_2), b) \longmapsto (b_2, \neg b) & (b_1, b_2) \longmapsto b_2 \end{array} \tag{1}$$

where we have implicitly used Currying for neg. It is not hard to see that the assertions are satisfied.

Many other models are possible. For example one can keep track of all the bits that have been used in a "history model" with state space $2 \times 2 \times 2^*$— where 2^* is the set of finite sequences of elements of 2. The definitions of the methods and initial state on this state space are left to the reader.

It is not hard to see that the above model on 2×2 is final: for an arbitrary coalgebra $d\colon Y \to 2 \times Y^2 \times 2$ satisfying the assertions there is an unique homomorphism $f\colon Y \to 2 \times 2$, namely $f = \langle \pi_1 \circ d, \pi_3 \circ d \rangle = \langle \mathsf{val}_d, \mathsf{aux}_d \rangle$.

Commutation with the val and aux methods is obvious; and for neg we have:

$$\begin{aligned}\mathsf{neg}(f(y), b) &= (\mathsf{aux}_d(y), \neg b) \\ &= (\mathsf{val}_d(\mathsf{neg}_d(y, b)), \mathsf{aux}_d(\mathsf{neg}_d(y, b))) \\ &= f(\mathsf{neg}_d(y, b)).\end{aligned}$$

Uniqueness of f is immediate.

4 Assertional Refinement

We start with the following basic situation describing assertional refinements. Briefly, they require validity of the abstract assertions after translation. We shall soon introduce several modifications and extensions on this definition. Later, in Section 5 we shall introduce "behavioural" refinements in which the requirement about validity of the abstract assertions is replaced by a requirement about appropriateness of the behaviour after translation.

Definition 4.1 [Base case] Consider a "concrete" coalgebraic specification $\mathfrak{C} = (\mathcal{C}, E_{\mathcal{C}}, I_{\mathcal{C}})$ and an "abstract" coalgebraic specifcation $\mathfrak{A} = (\mathcal{A}, E_{\mathcal{A}}, I_{\mathcal{A}})$.

An **assertional refinement** of $\mathfrak{A}$ by $\mathfrak{C}$ consists of a natural transformation $\tau\colon \mathcal{C} \Rightarrow \mathcal{A}$, translating the interfaces, such that for every coalgebra $c\colon X \to \mathcal{C}(X)$ and initial state $x_0 \in X$ with $c \models E_{\mathcal{C}}$ and $c, x_0 \models I_{\mathcal{C}}$, there is a $x'_0 \in X$ such that $c' \models E_{\mathcal{A}}$ and $c', x'_0 \models I_{\mathcal{A}}$, where $c' = \tau_X \circ c\colon X \to \mathcal{A}(X)$ is the translation of the coalgebra c via τ.

This definition is in a sense the obvious one, and is similar to refinement definitions used in algebraic specification, see for example [29]. What it involves is a "relative model construction": from an arbitrary model of the concrete specification one builds a model of the abstract specification. There is no requirement that this constructed model should be universal in some sense.

Example 4.2 In Example 3.1 we have specified a delayed negator via an auxiliary (private) operation. In this example we consider a second version of the negator system that works without an auxiliary operation. Its specification is in Figure 2. The aim is now to show that the specification DelayAuxNeg from Figure 1 is a refinement of the specification DelayNeg from Figure 2. (Later, in Section 5 we shall see that there is also a refinement, but a "behavioural" one, in the other direction.)

First of all, a refinement involves a translation of methods into methods via a natural transformation between the interface functors. In this case the translation should map val to val, neg to neg, and ignore aux. Further it should map the initial state new to new.

$$
\begin{array}{l}
\textbf{Begin}\ \text{DelayNeg} : \ \textbf{ClassSpec} \\
\quad \textbf{Method} \\
\qquad \mathsf{val}\colon \mathsf{Self} \longrightarrow 2 \\
\qquad \mathsf{neg}\colon \mathsf{Self} \times 2 \longrightarrow \mathsf{Self} \\
\quad \textbf{Assertion} \\
\qquad \forall b_1, b_2 \in 2.\ \mathsf{val}(\mathsf{neg}(\mathsf{neg}(x, b_1), b_2)) = \neg b_1 \\
\quad \textbf{Creation} \\
\qquad \mathsf{val}(\mathsf{new}) = 0 \\
\qquad \forall b \in 2.\ \mathsf{val}(\mathsf{neg}(\mathsf{new}, b)) = 0 \\
\textbf{End}\ \text{DelayNeg}
\end{array}
$$

Fig. 2. A second specification of the negator system, see Example 4.2

Formally, this done by a projection natural transformation:

$$
2 \times \mathsf{Self}^2 \times 2 \xRightarrow{\langle \pi_1, \pi_2 \rangle} 2 \times \mathsf{Self}^2 \tag{2}
$$

The DelayNeg assertions holds, if we assume the DelayAuxNeg assertions:

$$
\mathsf{val}(\mathsf{neg}(\mathsf{neg}(x, b_1), b_2)) \;=\; \mathsf{aux}(\mathsf{neg}(x, b_1) \;=\; \neg b_1.
$$

The first creation condition is immediate, for second we get

$$
\mathsf{val}(\mathsf{neg}(\mathsf{new}, b)) \;=\; \mathsf{aux}(\mathsf{new}) \;=\; 0
$$

We continue to discuss several modifications and generalisations of the definition of assertional refinement.

4.1 Multiple Initial States and Reachability

In Definition 3.2 we describe only one initial state 'new'. One may wish to have several initial states, for example, in the DelayAuxNeg specification, a second initial state with val and aux both equal to 1 could be useful. In general, one could allow an arbitrary collection $(\mathsf{new}_j\colon \mathsf{Self})_{j \in J}$ of initial states. This does not fundamentally change the picture: In an assertional refinement one then has to produce an appropriate number of initial states for the abstract specification. For simplicity, we will thus continue to assume that there is only one initial state.

In a computational interpretation of refinement one thinks of a concrete model behind the casing of an abstract interface. Every invocation of an abstract method is translated into some concrete method, then the concrete model does the work and the results are delivered. In particular, the creation of an object that provides the abstract interface should be translated to the creation of a concrete model plus a few initialisation steps. Therefore one expects that the translated initial state x'_0 is reachable from x_0.

Definition 4.1 does not formalise this aspect. One can fix this by adding the following reachability requirement: The translated initial state x'_0 must lay in the least invariant containing x_0. This least invariant contains precisely all states that are reachable from x_0. The reachability requirement is orthogonal to what we discuss in the following. It could be added to all following definitions of refinement.

4.2 The Nature of the Translation I

The translation from the concrete to the abstract interface in Definition 4.1 was handled by a natural transformation (like in [14]). This has turned out to be too restrictive. Here is a simple example, in which an "abstract" specification of quadrupling is refined by a "concrete" specification of duplicating:

Begin Quad : **ClassSpec**	**Begin** Dupl : **ClassSpec**
Method	**Method**
$\mathsf{out}\colon \mathsf{Self} \longrightarrow \mathbb{N}$	$\mathsf{out}\colon \mathsf{Self} \longrightarrow \mathbb{N}$
$\mathsf{quad}\colon \mathsf{Self} \longrightarrow \mathsf{Self}$	$\mathsf{dupl}\colon \mathsf{Self} \longrightarrow \mathsf{Self}$
Assertion	**Assertion**
$\mathsf{out}(\mathsf{quad}(x)) = 4 * \mathsf{out}(x)$	$\mathsf{out}(\mathsf{dupl}(x)) = 2 * \mathsf{out}(x)$
Creation	**Creation**
$\mathsf{out}(\mathsf{new}) = 1$	$\mathsf{out}(\mathsf{new}) = 1$
End Quad	**End** Dupl

It is obvious how to define a refinement here: the method dupl should be used twice to obtain the behaviour of quad. Both the interface functors of these specifications are $\mathbb{N} \times \mathsf{Self}$. There is no natural transformation $\mathbb{N} \times \mathsf{Self} \Rightarrow \mathbb{N} \times \mathsf{Self}$ which can do such a translation.

When we look at the involved coalgebras we can define a suitable translation, namely:

$$\begin{array}{ccc} \big(\mathsf{Self} \longrightarrow \mathbb{N} \times \mathsf{Self}\big) & \longrightarrow & \big(\mathsf{Self} \longrightarrow \mathbb{N} \times \mathsf{Self}\big) \\ \langle \mathsf{out}, \mathsf{dupl} \rangle & \longmapsto & \langle \mathsf{out}, \mathsf{dupl} \circ \mathsf{dupl} \rangle \end{array}$$

We find it convenient to combine the translation of the coalgebra with the translation of the initial state into a single function of the form:

$$\begin{array}{ccc} \big(\mathsf{Self} \longrightarrow \mathbb{N} \times \mathsf{Self}\big) \times \mathsf{Self} & \longrightarrow & \big(\mathsf{Self} \longrightarrow \mathbb{N} \times \mathsf{Self}\big) \times \mathsf{Self} \\ (\langle \mathsf{out}, \mathsf{dupl} \rangle, \mathsf{new}) & \longmapsto & (\langle \mathsf{out}, \mathsf{dupl} \circ \mathsf{dupl} \rangle, \mathsf{new}) \end{array} \tag{3}$$

This format works most of the time. But in certain situations it is still not general enough, see Subsection 4.4 below.

4.3 The Need for Invariants

Assertional refinement in Definition 4.1 requires that the abstract assertions hold for all states. Often, this is not the case: they hold only under certain well-behavedness conditions. The latter are usually expressed via invariants. Here is a simple example.

$$
\begin{array}{l}
\textbf{BEGIN}\ \mathrm{BitStore}:\ \textbf{CLASSSPEC}\\
\quad \textbf{METHOD}\\
\quad\quad \mathsf{list}\colon \mathsf{Self} \longrightarrow 2^*\\
\quad\quad \mathsf{add}\colon \mathsf{Self} \times 2 \longrightarrow \mathsf{Self}\\
\quad \textbf{ASSERTION}\\
\quad\quad \forall b \in 2.\ \mathsf{list}(\mathsf{add}(x, b)) = b \cdot \mathsf{list}(x)\\
\quad \textbf{CREATION}\\
\quad\quad \mathsf{list}(\mathsf{new}) = \langle\rangle\\
\textbf{END}\ \mathrm{BitStore}
\end{array}
$$

The operation $\cdot$ is used in the assertion for prefixing a bit to a finite list.

We wish to use this BitStore specification to refine the DelayAuxNeg specification from Example 3.1. The idea is to use the first two bits from the store for val and aux. This is used in a translation:

$$
\begin{array}{ccc}
\big(\mathsf{Self} \longrightarrow 2^* \times \mathsf{Self}^2\big) \times \mathsf{Self} & \longrightarrow & \big(\mathsf{Self} \longrightarrow 2 \times \mathsf{Self}^2 \times 2\big)\\
(\langle \mathsf{list}, \mathsf{add}\rangle, y) & \longmapsto & (\langle \mathsf{val}, \mathsf{neg}, \mathsf{aux}\rangle, y')
\end{array}
$$

where:

$$
\begin{aligned}
\mathsf{val}(x) &= \begin{cases} 0 & \text{if } \mathsf{list}(x) = \langle\rangle \text{ or } \mathsf{list}(x) = \langle b\rangle \\ b_2 & \text{if } \mathsf{list}(x) = b_1 \cdot b_2 \cdot \ell \end{cases}\\
\mathsf{neg}(x)(b) &= \mathsf{add}(x)(\neg b)\\
\mathsf{aux}(x) &= \begin{cases} 1 & \text{if } \mathsf{list}(x) = \langle\rangle \\ b_1 & \text{if } \mathsf{list}(x) = b_1 \cdot \ell \end{cases}\\
y' &= \mathsf{add}(y)(0).
\end{aligned}
$$

The predicate $P(x) \stackrel{\text{def}}{=} \mathsf{length}(\mathsf{list}(x)) > 0$ describes the "good states" in which we are really interested for the refinement. There, we don't have to choose an arbitrary value for the aux method: the 1 for in the first case for aux above. It is not hard to see that the "abstract" DelayAuxNeg assertions are implied by P, and also that the translated initial state y' is contained in P—and that the DelayAuxNeg assertion $\mathsf{val}(\mathsf{neg}(x, b)) = \mathsf{aux}(x)$ does not hold outside P. Further, once P holds for the translated coalgebra, it will continue to hold. Thus P is an invariant for the translated coalgebra, which singles out the relevant states.

In general, we thus wish to modify Definition 4.1 in such a way that the abstract assertions are required not to hold universally, but only on an invariant on the translated coalgebra, and that the translated initial state is contained in this invariant. This is justified because by using the translated initial state and methods, one will never go outside such an invariant. If the abstract assertions hold there, the refinement should work. This is formalised in Definition 4.3 below.

4.4 The Nature of the Translation II

In the refinement example in the previous subsection we have chosen an arbitrary value for the methods aux outside the invariant. Once one takes the use of invariants seriously, it becomes natural to translate the concrete coalgebra into an abstract coalgebra which is only defined on an invariant. In certain situations this is even required. Consider for example the following specification fragments.

Begin Abs : **ClassSpec**	**Begin** Conc : **ClassSpec**
Method	**Method**
$\mathsf{a}\colon \mathsf{Self} \longrightarrow \{-1, 1\}$	$\mathsf{b}\colon \mathsf{Self} \longrightarrow \mathbb{R}$
...	...
End Abs	**End** Conc

Suppose the intended refinement involves a translation $a(x) = \frac{b(x)}{|b(x)|}$ where we have an invariant $b(x) \neq 0$. Clearly, this invariant is needed in order to have a well-defined translation of coalgebras.

The translation as in (3) thus must be modified into:

$$\big(X \longrightarrow \mathcal{C}(X)\big) \times X \quad \xrightarrow{\qquad\quad} \quad \coprod_{P \subseteq X} \big(P \longrightarrow \mathcal{A}(P)\big) \times P \tag{4}$$

for every set X. The fact that such translation yields a coalgebra $P \to \mathcal{A}(P)$ on a subset P means that P is closed under the translated operations.

Actually, this closure can sometimes only be proved if we know that the coalgebra to which the translation is applied satisfies the assertions. Therefore, the translation can be made more precise with the notation from Definition 3.2:

$$\coprod_{c\colon X \xrightarrow{E_{\mathcal{C}}} \mathcal{C}(X)} I_{\mathcal{C}}(c) \quad \xrightarrow{\qquad\quad} \quad \coprod_{P \subseteq X} \; \coprod_{c'\colon P \xrightarrow{E_{\mathcal{A}}} \mathcal{A}(P)} I_{\mathcal{A}}(c') \tag{5}$$

But this says it all: it takes all of the above aspects (except multiple initial states and reachability) into account.

Definition 4.3 [General case] An **assertional refinement** of an "abstract" coalgebraic specification $\mathfrak{A} = (\mathcal{A}, E_{\mathcal{A}}, I_{\mathcal{A}})$ by a "concrete" one $\mathfrak{C} = (\mathcal{C}, E_{\mathcal{C}}, I_{\mathcal{C}})$ consists of a translation map (5).

The following proposition is immediate—for all versions of assertional refinement we have discussed so far.

Proposition 4.4 *Assertional refinement is compositional: if $\mathfrak{A}$ is refined by $\mathfrak{B}$ and $\mathfrak{B}$ is refined by $\mathfrak{C}$ then $\mathfrak{A}$ is refined by $\mathfrak{C}$.* □

This is so-called vertical composition. What is called horizontal composition [6] may be defined in a setting with parameterised specifications—which we do not have.

5 Behavioural Refinement

In the previous section we have focused on refinement in terms of validity of assertions. Next we shall introduce refinement in terms of simulation of behaviour. This turns out to be more flexible.

Throughout this section we assume that assertions and creation conditions are stable under bisimilarity in the following sense: Let $c : X \to \mathcal{T}(X)$ and $d : Y \longrightarrow \mathcal{T}(Y)$ be two $\mathcal{T}$ coalgebras. Then for two arbitrary states $x \in X$ and $y \in Y$ and a formula F over $\mathcal{T}$ we have that if $x \;{}_{c}{\underline{\leftrightarrow}}_{d}\; y$, then $c, x \models F$ if and only if $d, y \models F$. This property can be enforced via the following two syntactic restrictions: First, formulae must not contain equality on types that contain Self. For such types only behavioural equality $\underline{\leftrightarrow}$ may be used. Second, also quantification is restricted to those types that do not involve Self. Lemma 2.4 (vi) shows that formulae that are stable under bisimilarity are also stable under coalgebra morphisms: If $f : c \to d$ is a coalgebra morphism, then $c, x \models F$ if and only if $d, f(x) \models F$.

Recall from Example 3.1 that the DelayAuxNeg specification involves a *private* method aux—a fact that has been ignored so far. Usually, this means that the method can not be called from the outside, but only within the methods of the class itself. Therefore it seems reasonable to say that a refinement of this DelayAuxNeg class need not to implement the private method aux, because the fact that it is used internally is irrelevant from an external point of view. But if we do not have an implementation of aux, we cannot talk about validity of the DelayAuxNeg assertions anymore—because they use aux. Therefore, in behavioural refinement the assertions do not play any rôle, and indistinguishability of behaviour is all that counts.

Before we proceed we need to be more precise about omitting certain methods. We shall do this in a fairly general way, like in Display (2) in Example 4.2, via a natural transformation $\mathcal{A} \Rightarrow \mathcal{A}'$ from the interface functor $\mathcal{A}$ of the abstract class to another interface functor $\mathcal{A}'$ which we think of as describing the public interface of the abstract class. Such a natural transformation $\mathcal{A} \Rightarrow \mathcal{A}'$ will sometimes be called a view. Usually it projects away a part of the $\mathcal{A}$-interface.

Definition 5.1 A **coalgebraic specification with a view** $(\mathcal{A}, E, I, \mathcal{A} \stackrel{\nu}{\Rightarrow} \mathcal{A}')$ consists of a coalgebraic specification $(\mathcal{A}, E, I)$ together with a natural transformation $\nu\colon \mathcal{A} \Rightarrow \mathcal{A}'$ between two polynomial functors.

A **behavioural model** of $(\mathcal{A}, E, I, \mathcal{A} \stackrel{\nu}{\Rightarrow} \mathcal{A}')$ is given by an $\mathcal{A}'$-coalgebra $c\colon X \to \mathcal{A}'(X)$ of the "public interface" together with an initial state $x_0 \in X$ which "behaves like" an actual $(\mathcal{A}, E, I)$-model: there is a $d\colon Y \xrightarrow{E} \mathcal{A}(Y)$ and $y_0 \in Y$ with $d, y_0 \models I$ such that x_0 and y_0 are "publicly indistinguishable", *i.e.* $\mathcal{A}'$-bisimilar: $x_0 \; {}_{c}\underline{\leftrightarrow}_{\nu\circ d} \; y_0$.

The next result shows that there is a canonical choice for the actual model.

Proposition 5.2 *Assume that there is a final coalgebra $e\colon Z \to \mathcal{A}(Z)$ among the coalgebras satisfying $E_{\mathcal{A}}$, with an initial state $z_0 \in Z$ such that $e, z_0 \models I$. Then, in the situation of the previous definition, the following two points are equivalent.*

(i) *There is a model $d\colon Y \xrightarrow{E} \mathcal{A}(Y)$ and $y_0 \in Y$ with $d, y_o \models I$ and $x_0 \; {}_{c}\underline{\leftrightarrow}_{\nu\circ d} \; y_0$.*

(ii) *For the final model $e\colon Z \xrightarrow{E} \mathcal{A}(Z)$ there is an initial state $z_0 \in Z$ such that $e, z_0 \models I$ and $x_0 \; {}_{c}\underline{\leftrightarrow}_{\nu\circ e} \; z_0$.*

These points are obviously a consequence of the next universally quantified statement, which is often convenient in actual refinement proofs (as in [13]).

(iii) *For any model model $d\colon Y \xrightarrow{E} \mathcal{A}(Y)$ and $y_0 \in Y$ satisfying $d, y_0 \models I$ one has $x_0 \; {}_{c}\underline{\leftrightarrow}_{\nu\circ d} \; y_0$.*

Proof. Clearly, (ii) $\Rightarrow$ (i), so we concentrate on the reverse implication. Let there be a $d\colon Y \xrightarrow{E} \mathcal{A}(Y)$ with $y_0 \in Y$ as in the definition. Then, by finality of $e : Z \to \mathcal{A}(Z)$ there is a coalgebra morphism $f : d \to e$. This yields $e, f(y_0) \models I$ (because creation conditions are stable under bisimilarity). Consider now the following diagram

$$\begin{tikzcd}
\mathcal{A}'(Y) \arrow[r, "\mathcal{A}'(f)"] & \mathcal{A}'(Z) \\
\mathcal{A}(Y) \arrow[u, "\nu_Y"] \arrow[r, "\mathcal{A}(f)"] & \mathcal{A}(Z) \arrow[u, "\nu_Z"'] \\
Y \arrow[u, "d"] \arrow[r, "f"] & Z \arrow[u, "e"']
\end{tikzcd}$$

Finality gives the lower square and naturality of ν the upper one. So f is also a morphism of $\mathcal{A}'$-coalgebras and by Lemma 2.4 (vi) its graph is an $\mathcal{A}'$-bisimulation. Therefore $y_0 \; {}_{\nu\circ d}\underline{\leftrightarrow}_{\nu\circ e} \; f(y_0)$, so that $x_0 \; {}_{c'}\underline{\leftrightarrow}_{\nu\circ e} \; f(y_0)$ by transitivity of bisimilarity (Lemma 2.4 (ii)). This shows that the final coalgebra also works. $\square$

Definition 5.3 Consider "abstract" and "concrete" coalgebraic specifications with a view $\mathfrak{A} = (\mathcal{A}, E_{\mathcal{A}}, I_{\mathcal{A}}, \mathcal{A} \overset{\nu}{\Rightarrow} \mathcal{A}')$ and $\mathfrak{C} = (\mathcal{C}, E_{\mathcal{C}}, I_{\mathcal{C}}, \mathcal{C} \overset{\mu}{\Rightarrow} \mathcal{C}')$. A **behavioural refinement** of $\mathfrak{A}$ by $\mathfrak{C}$ consists of a translation map for the public interfaces:

$$\big(X \to \mathcal{C}'(X)\big) \times X \xrightarrow{\quad\phi\quad} \big(P \longrightarrow \mathcal{A}'(P)\big) \times P$$

which maps a behavioural $\mathfrak{C}$-model to a behavioural $\mathfrak{A}$-model.

Example 5.4 In Example 4.2 we have seen how the DelayAuxNeg specification assertionally refines the DelayNeg specification by using its aux method. Here we like to show that, in the other direction, there is behavioural refinement. More precisely, the DelayNeg specification with identity view behaviourally refines the DelayAuxNeg with the view $\nu = \langle \pi_1, \pi_2 \rangle$ from (2) that projects away the aux method.

As translation map ϕ in Definition 5.3 we simply take the identity. And as DelayAuxNeg model we take the final one, described in Example 3.3, written as $d = \langle \mathsf{val}_d, \mathsf{neg}_d, \mathsf{aux}_d \rangle \colon X \to 2 \times X^2 \times 2$ in (1). Thus we have to show that for an arbitrary DelayNeg model $c = \langle \mathsf{val}_c, \mathsf{neg}_c \rangle \colon X \to 2 \times X^2$ with initial state x_0 we have a bisimilarity $x_0 \;{}_{c}\!\underline{\leftrightarrow}_{\nu \circ d}\; (0, 0)$.

A bisimulation R with $R(x_0, (0, 0))$ is for example:

$$R(x,\ (b_1, b_2)) \quad \Longleftrightarrow \quad \begin{cases} \mathsf{val}_c(x) = \mathsf{val}_d(b_1, b_2) \qquad \wedge \\ \forall b \in 2.\ \mathsf{val}_c(\mathsf{neg}_c(x, b)) = \mathsf{val}_d(\mathsf{neg}_d((b_1, b_2), b)) \end{cases}$$

This is indeed an appropriate bisimulation: if $R(x, (b_1, b_2))$, then $\mathsf{val}_c(x) = \mathsf{val}_d(b_1, b_2)$, and for each $b \in 2$ the pair $(\mathsf{neg}_c(x, b), \mathsf{neg}_d((b_1, b_2), b))$ is again in R: The equality of $\mathsf{val}_c(\mathsf{neg}_c(x, b)) = \mathsf{val}_d(\mathsf{neg}_d((b_1, b_2), b))$ is immediate and

$$\begin{array}{rcll} \mathsf{val}_c(\mathsf{neg}_c(\mathsf{neg}_c(x, b), b')) & = & \neg b & \text{by the DelayNeg assertion} \\ & = & \mathsf{val}_d(\neg b, \neg b') & \\ & = & \mathsf{val}_d(\mathsf{neg}_d((b_2, \neg b), b')) & \\ & = & \mathsf{val}_d(\mathsf{neg}_d(\mathsf{neg}_d((b_1, b_2), b), b')) & \end{array}$$

This shows that DelayNeg is a behavioural refinement of DelayAuxNeg: the DelayAuxNeg behaviour is realised without implementing the private method aux.

The alert reader may have noticed that in this simple example there is actually an implementation of aux, which works appropriately using an invariant. Define:

$$\begin{array}{rcl} P(x) & \Longleftrightarrow & \mathsf{val}_c(\mathsf{neg}_c(x, 0)) = \mathsf{val}_c(\mathsf{neg}_c(x, 1)) \\ \mathsf{aux}_c(x) & = & \mathsf{val}_c(\mathsf{neg}_c(x, 0)). \end{array}$$

Clearly, P is closed under neg_c (so it is an invariant) and contains the initial state new. Also, the DelayAuxNeg equations hold on P. Thus we have an assertional refinement too.

The next result shows that assertional refinement is the same as behavioural refinement with identity views.

Theorem 5.5 *A concrete specification* $(\mathcal{C}, E_{\mathcal{C}}, I_{\mathcal{C}})$ *is an assertional refinement of an abstract specification* $(\mathcal{A}, E_{\mathcal{A}}, I_{\mathcal{A}})$ *if and only if* $(\mathcal{C}, E_{\mathcal{C}}, I_{\mathcal{C}}, \mathcal{C} \stackrel{id}{\Rightarrow} \mathcal{C})$ *is a behavioural refinement of* $(\mathcal{A}, E_{\mathcal{A}}, I_{\mathcal{A}}, \mathcal{A} \stackrel{id}{\Rightarrow} \mathcal{A})$.

Proof. First, assume $(\mathcal{C}, E_{\mathcal{C}}, I_{\mathcal{C}})$ is an assertional refinement of $(\mathcal{A}, E_{\mathcal{A}}, I_{\mathcal{A}})$, say via translation map ϕ as in (5). A concrete model c, x_0 is also a behavioural model with identity view, and so we get an abstract (behavioural) model $\phi(c, x_0) = (P, c', x'_0)$. Thus we can take $d = c'$ and $y_0 = x'_0$ in Definition 5.1.

In the other direction, assume $(\mathcal{C}, E_{\mathcal{C}}, I_{\mathcal{C}}, \mathcal{C} \stackrel{id}{\Rightarrow} \mathcal{C})$ is a behavioural refinement of $(\mathcal{A}, E_{\mathcal{A}}, I_{\mathcal{A}}, \mathcal{A} \stackrel{id}{\Rightarrow} \mathcal{A})$, via a map ϕ as in Definition 5.3. For a concrete model c, x_0 we get an abstract coalgebra $\phi(c, x_0) = (P, c', x'_0)$, for which there is an abstract model $d\colon Y \to \mathcal{A}(Y)$, $y_0 \in Y$ with $x'_0 \;{}_{c'}\!\underline{\leftrightarrow}_d\; y_0$. Define the predicate $Q(x) = P(x) \wedge \exists y \in Y.\; x \;{}_{c'}\!\underline{\leftrightarrow}_d\; y$. We claim that (Q, c', x'_0) is then also an abstract model. Clearly, Q contains x'_0. And it is a c'-invariant by Lemma 2.4 (vii). Because assertions are stable under bisimilarity, we get $Q(x) \Rightarrow E_{\mathcal{A}}(c')(x)$. Similarly, $I_{\mathcal{A}}(c')(x'_0)$ follows from $I_{\mathcal{A}}(d)(y_0)$ and $x'_0 \;{}_{c'}\!\underline{\leftrightarrow}_d\; y_0$. □

There is also the analogue of Proposition 4.4.

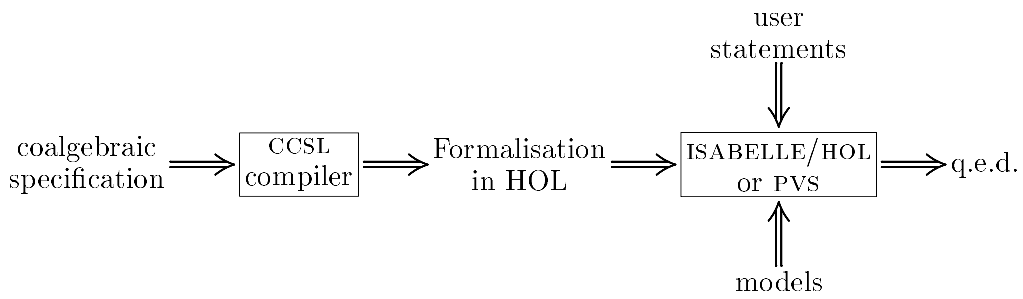
Proposition 5.6 *Behavioural refinement is compositional.* □

6 The Coalgebraic Class Specification Language CCSL

In this section we shortly describe the Coalgebraic Class Specification Language CCSL. CCSL combines algebraic and coalgebraic specification in one specification language (see [29] for algebraic specification). The advantage is, that in CCSL, one can combine algebraic and coalgebraic specifications in so-called *iterated specifications* [8]. One can, for example, first describe the abstract data type of binary trees and then use this data type in a class specification. All the example specifications from the preceding sections are valid CCSL specifications (modulo the use of symbols like ∀). However, CCSL allows more general coalgebraic signatures. Currently it supports signatures that correspond to the *higher-order polynomial functors* from [28]. CCSL uses a higher-order logic extended with infinitary modal operators. The main reference on CCSL is [26], but see also the older reference [9].

In order to get theorem proving support, CCSL can be translated into the higher-order logics of PVS [24] and ISABELLE/HOL [25]. The translation is performed by the CCSL compiler that acts as a frontend to either PVS or

ISABELLE. The working environment looks as follows:



The CCSL compiler reads files containing (co)algebraic specifications and produces output for either PVS or ISABELLE/HOL (depending on a command line switch). The produced output can be directly loaded into PVS or ISABELLE. The formulae one wants to prove and the models must be formulated in the logic of the target theorem prover.

CCSL and its compiler are research tools developed jointly at the universities of Nijmegen and Dresden within the LOOP project [3] on formal methods of object-oriented programming. A public release is planned in the foreseeable future. Because of the research character of CCSL, it contains some features that are not fully elaborated at this stage. However, all the examples in this paper are within the well-understood fragment of CCSL that could be paraphrased as *first order coalgebraic specification with infinitary modal operators.*

Coalgebraic specifications in the sense of Definition 3.2 are valid CCSL specifications. The concrete syntax of CCSL is similar to PVS and mostly self explaining. The example in the next section employs a special extension of equational logic: methodwise infinitary modal operators. These modal operators have the form

$$(\Box_x^A\ F)(t) \qquad \text{and} \qquad (\Diamond_x^A\ F)(t)$$

where F is a formula with a free variable $x : \mathsf{Self}$, A is a set of method identifiers, and t is a term of type Self. The operators are pronounced as *henceforth* F (for $\Box F$) and *eventually* F (for $\Diamond F$). The intuition is the following: The term t determines some state $y \in X$. By (recursively) applying all methods that are listed in A to y one can form the set $X' \subseteq X$ of states that are reachable from y by methods in A. Now, $(\Box_x^A\ F)(t)$ is true, if F is true for all $x \in X'$. And $(\Diamond_x^A\ F)(t)$ holds if there is at least one $x \in X'$ fulfilling F.

In a formal treatment we only have to consider $\Box$, as usual, $\Diamond$ can be defined as syntactic sugar as $\neg\Box\neg$. Assume in the following a coalgebraic specification with a set M of method declarations $m_i : \mathsf{Self} \longrightarrow \mathcal{T}_i(\mathsf{Self})$. In this case the interface type of the whole specification is

$$\mathcal{T}(\mathsf{Self}) \quad = \quad \mathcal{T}_1(\mathsf{Self}) \times \cdots \times \mathcal{T}_i(\mathsf{Self}) \times \cdots$$

[3] LOOP stands for Logic of Object-Oriented Programming. See URL `http://www.cs.kun.nl/~bart/LOOP/`.

Note that for any subset $A \subseteq M$ there is an interface type $\mathcal{T}_A$ describing just the methods in A together with a projection natural transformation $\pi_A : \mathcal{T}(\mathsf{Self}) \Longrightarrow \mathcal{T}_A(\mathsf{Self})$ that 'forgets' all the methods that are not in A. If $d : X \longrightarrow \mathcal{T}(X)$ is a coalgebra realizing all methods in M, then $\pi_A \circ d : X \longrightarrow \mathcal{T}_A(X)$ is a coalgebra restricted to the methods in A.

Syntactically the methodwise modal operator $\Box^A$ is introduced with the following rule

$$\frac{x : \mathsf{Self} \;\vdash\; F : \mathsf{Prop}}{y : \mathsf{Self} \;\vdash\; (\Box_x^A\; F)\; (y) : \mathsf{Prop}}\; A \subseteq M$$

That is, if F is a formula with a free variable x, then $(\Box_x^A\; F)(y)$ is a formula with a free variable y (the x is bound in $(\Box_x^A\; F)$, provided that A is a subset of the available method identifiers M. The variable binding of $\Box_x^A$ leads to the following substitution rules:

$$((\Box_x^A\; F)\; t)\; [s/y] \quad = \quad \begin{cases} (\Box_x^A\; F)\; (t[s/y]) & \text{for } x = y \\ (\Box_x^A\; F[s/y])\; (t[s/y]) & \text{for } x \neq y \end{cases}$$

For the semantics of $\Box^A$, note that F has a free variable x, so the semantics of F is a predicate $[\![\, F \,]\!] \subseteq X$. Then the semantics of $\Box_x^A$ with respect to a model $d : X \to \mathcal{T}(X)$ is as follows.

$$[\![\, \Box_x^A\; F \,]\!] \subseteq X \quad \stackrel{\text{def}}{=} \quad \underline{([\![\, F \,]\!])}_{\,\pi_A \circ d}$$

where $\underline{(-)}_{\,\pi_A \circ d}$ denotes the greatest invariant with respect to the restricted coalgebra $\pi_A \circ d$. Note that from Lemma 2.4 (iv) it follows that the greatest invariant contained in a predicate can be characterised as

$$\underline{P}_{\,c} \quad = \quad \bigcup \{Q \mid Q \subseteq P \text{ and } Q \text{ is an invariant for } c\}$$

For $\Diamond$ we define

$$(\Diamond_x^A\; F)(t) \quad \stackrel{\text{def}}{=} \quad \neg\, (\Box_x^A\; \neg F)(t)$$

In the concrete syntax of CCSL, a formula $(\Box_x^{\{m_1,\ldots,m_n\}}\; F)(t)$ translates to

Always (**Lambda**(x : **Self**) . F) **For** $\{\; m_1, \ldots, m_n \;\}$ (t)

and $(\Diamond_x^{\{m_1,\ldots,m_n\}}\; F)(t)$ to

Eventually (**Lambda**(x : **Self**) . F) **For** $\{\; m_1, \ldots, m_n \;\}$ (t)

Besides coalgebraic specifications CCSL also allows abstract data type definitions. This allows the definition of types like $\mathsf{list}[\tau]$, representing the finite lists over some type τ. For an abstract data type definition one has to give the constructor signature of the abstract data type. The semantics is the initial model of the given signature. Figure 3 shows as an example the parametric abstract data type InterruptOrSucceed. This type is used in the following

```
Begin InterruptOrSucceed[ X : Type ] : ADT
  Constructor
    Interrupt  :  X -> Carrier;
    Succeed    :  X -> Carrier;
End InterruptOrSucceed
```

Fig. 3. The labelled coproduct InterruptOrSucceed

section. An element of InterruptOrSucceed[τ] is an element of τ tagged with either Interrupt or Succeed. So InterruptOrSucceed[τ] is a labelled version of $\tau + \tau$.

7 Refining a Transaction Mechanism

This section presents coalgebraic specification at work. It introduces a specification of an abstract transaction-commit mechanism, with two standard refinements. Such "two-phase update" mechanisms are used to make certain designated sequences of actions atomic, for example via commands like `StartTransaction(); action_1; ... action_n; CommitTransaction();` These mechanisms are commonly found in the context of databases or operating systems (see the standard reference [18]), but also in the world of smart cards—where there is always the possibility that a card is removed before an appropriate sequence of actions is completed, see [5, Chapter 5].

In this section we drastically simplify the possible actions that may occur, and concentrate on the behaviour of the start and commit of transactions. In fact, we consider only one possible action: addition. Its result should be visible via a value attribute: After addition of $n \in \mathbb{Z}$ the value is the original value plus n. Several of such additions can be surrounded by start and commit actions. What we wish to specify is that after a start of a transaction, the effect of (successful) additions is not visible until after a (successful) commit. In the abstract specification we allow possible interruptions both of addition and of commit. This is represented, in typically coalgebraic style, via the structured output type InterruptOrSucceed from Figure 3.

7.1 The Abstract Specification

The CCSL source code for the coalgebraic signature of the abstract transaction interface is in Figure 4. In the following we go through it line by line. CCSL keywords are set in **bold sans serif**; comments start with a percent sign %, they are set in *italics*. Class specification start with the keyword **Begin** followed by the name of the specification followed by the keyword **ClassSpec**. The keyword **Method** on line 2 in Figure 4 starts the declaration of methods.

The method val is viewed as an attribute that returns the current integer value. The method add can be used to change the attribute val. It takes

```
Begin TransactionCommitAbstract : ClassSpec
  Method
    val    : Self -> int;                              % current value
    add    : [Self, int] -> InterruptOrSucceed[Self];  % the action
    start  : Self -> Self;                             % start of transaction
    commit : Self -> InterruptOrSucceed[Self];         % end of transaction

  Assertion
        ⋮    % the assertions are explained in the text

  Constructor
    new : Self;

  Creation
        ⋮    % the creation condition is in the text

End TransactionCommitAbstract
```

Fig. 4. The signature of the abstract specification

an additional argument of type int. The product of Self and int is written as [**Self**, int] in CCSL, similar to PVS. The assertions of the specification ensure that the effect of the add operations becomes only visible after applying commit. The method add can fail because it gets interrupted. In this case it returns a value of the form Interrupt(x'), where x' is the state of the machine after interruption. Similarly, if add is not interrupted, then it returns Succeed(x') with a successor state x'. The methods start and commit begin and end a transaction.

The keyword **Constructor** starts a section that can declare several parametrised initial states. Here we only require one initial state, denoted by new.

The specification TransactionCommitAbstract has six assertions and one creation condition. All assertions formalise different aspects of the behaviour of the methods. In the following we explain them one by one. In a CCSL specification assertions are in the assertion section, started with the keyword **Assertion**. The first assertion is

```
val_start : x.start.val = x.val;
```

Here val_start is the name of the assertion and all text between the colon and the final semicolon constitutes an assertion over the signature of Transaction-CommitAbstract in the sense of Section 3. In CCSL we use the *dot* notation for method invocation, so x.start.val is the result (of integer type) of first applying the method start to the variable x and then retrieving the val attribute from the resulting state. So the assertion val_start expresses that for any state x the following holds: If you start a transaction via the method start, then the value does not change.

The next assertion defines the behaviour of add

```
val_add : Forall( n : int ) .
    Cases x.add( n ) OF
      Interrupt( y ) : y.val = x.val;
      Succeed( z )   : z.val = x.val;
    Endcases;
```

It uses a case expression to discriminate the result of the add method. In case that the addition is interrupted the equation y.val = x.val must hold, where y is the resulting state from the interrupted adding attempt. All together the assertion val_add states that for all integers that one might want to add, the value does not change, regardless whether the method add succeeds or gets interrupted.

The assertion val_commit describes the value after an interrupted commit attempt: As described before, it is required not to change. The CCSL source is

```
val_commit : Cases x.commit OF
                Interrupt(y) : y.val = x.val;
                Succeed(z)   : True;
             Endcases;
```

Note that in case the attempt to commit succeeds, we have the trivial requirement of **True**. This has the effect that the assertion val_commit is trivially fulfilled if committing succeeds.

The next three assertions regulate the behaviour of successful commits. The problem here is that a successful commit changes the val attribute in a way that depends on all previous successful add's. But there is no way to access the previously executed methods. We decided therefore to specify the behaviour of a successful commit in an inductive way. The first assertion commit_start deals with the case that the method commit is directly invoked after starting a transaction.

```
commit_start : Eventually (Lambda( y : Self ) .
                  Cases y.commit OF
                     Interrupt(v) : False;
                     Succeed(w)   : w.val = x.val;
                  Endcases)
               For { commit } (x.start);
```

Here we use method-wise modal operator $\Diamond^{\mathsf{commit}}$ to express that the successive application of the method commit to the resulting state of x.start yields eventually a state y that makes the inner case expression true. In other words, the assertion commit_start expresses the liveness property that in state x.start the attempt to commit eventually succeeds with a state that holds a value identical to that of x.

The following assertion deals with the case that some successful add operations have been performed since the start of the transaction:

```
commit_add : Forall( n, m : int ) .
    Eventually (Lambda( y : Self ) .
        Cases y.commit OF
            Interrupt(v) : False;
            Succeed(z)   : z.val = m;
        Endcases)
    For { commit } (x)
  Implies
    Cases x.add(n) OF
      Interrupt(y) : True;
      Succeed(z)   : Eventually (Lambda( y : Self ) .
                         Cases y.commit OF
                             Interrupt(v) : False;
                             Succeed(w)  : w.val = m + n;
                         Endcases)
                     For { commit } (z);
    Endcases;
```

One can paraphrase this assertion in English as follows: Assume that in state x successive attempts to commit succeed eventually with the value attribute equal to m. Assume further that adding the integer n in state x succeeds and yields a state z. Then we require that successive attempts to commit starting in state z do eventually succeed with a value equal to m + n.

The last assertion of the abstract specification deals with the case that the commit method is further applied after a successful commit. It states that once a commit succeeded, the value field stays constant:

```
commit_commit :
    Cases x.commit OF
        Interrupt(y) : True;
        Succeed(z)   : Always (Lambda( v : Self ) . v.val = z.val)
                       For { commit } (z);
    Endcases;
```

Finally, the creation condition requires that in a newly created object the value attribute is zero:

```
val_new : new.val = 0;
```

From the specification TransactionCommitAbstract the CCSL compiler produces several PVS theories that formalise its semantics in the higher-order logic of PVS. From the method declarations the compiler generates the PVS type definition TransactionCommitAbstractSignature. Inhabitants of this type are coalgebras for the signature of the abstract transaction specification. Further the compiler performs predicate and relation lifting according to Definition 2.1 and generates definitions for invariants, bisimulations, and the modal operators. In addition to the definitions many lemmas are generated that are useful for rewriting purposes. Finally it translates the assertions and creation conditions into PVS and combines them in a predicate TransactionCommitAbstractModel?. This predicate holds for an inhabitant of type TransactionCommitAbstractSignature and a state new, precisely if they form a model of the abstract transaction specification according to Definition 3.2. The whole translation into PVS is a definitional extension, it does not use any axioms.

In PVS it is now possible to construct a model of the specification, for example with state space $\mathbb{Z} \times \mathbb{Z}$ for the visible value and the value of incorporating the as yet uncommitted additions. Also, one can deduce consequences from specifications. For instance, we used PVS to prove the following (intended) correctness property.

Lemma 7.1 *Let d be an arbitrary coalgebra on state space X of the interface type of the TransactionCommitAbstract specification that fulfils all assertions. Define the function* $\mathsf{adds} : X \times \mathsf{List}[\mathbb{Z}] \longrightarrow \mathsf{InterruptOrSucceed}[X]$ *recursively as*

$$\mathsf{adds}(x, l) \quad = \quad \begin{cases} \mathsf{Succeed}(x) & \textit{if } l \textit{ is the empty list} \\ \mathsf{Interrupt}(y) & \begin{array}{l} \textit{if } l = \mathsf{cons}(n, l') \textit{ and} \\ \mathsf{adds}(x, l') = \mathsf{Interrupt}(y) \end{array} \\ \mathsf{add}(x', n) & \begin{array}{l} \textit{if } l = \mathsf{cons}(n, l') \textit{ and} \\ \mathsf{adds}(x, l') = \mathsf{Succeed}(y) \end{array} \end{cases}$$

Assume that $\mathsf{adds}(x.\mathsf{start}, l) = \mathsf{Succeed}(y)$. *Then successive attempts to commit in state y eventually succeed with a value attribute equal to the sum of all elements of the list l plus the value in state x.* □

7.2 Refinement 1: New Value Logging

To implement a transaction mechanism as specified in the preceding subsection there are two standard approaches [23]: "new value logging" and "old value logging". The idea of new value logging is to keep a private copy of the current state and apply changes in the course of a transition only to that private copy. Then the commit action only needs to switch to the new state. In old value logging one logs the actions instead of doing them. In this approach the commit action does all the work, it walks through the list of actions and performs them one after each other.

In this subsection we refine the abstract transaction mechanism to use new value logging. The CCSL sources are in Figure 5 and Figure 6. Besides the methods that are already present in the specification TransactionCommitAbstract we have a new private method intent, that possibly holds a different state. The type constructor Lift is predefined in CCSL as $\mathsf{Lift}[\tau] = \tau + \mathbf{1}$ with

```
Begin NewValueLogTransactionCommit : ClassSpec
  Method
    val    : Self -> int;                                 % current value
    add    : [Self, int] -> InterruptOrSucceed[Self];     % the action
    start  : Self -> Self;                                % start of transaction
    commit : Self -> Self;                                % end of transaction

  Private Method
    intent : Self -> Lift[Self];          % attribute for holding the new state

  Assertion
    val_start : x.start.val = x.val;

    intent_intent : Cases x.intent OF
                        bot   : True;
                        up(v) : Cases v.intent OF
                                    bot   : True;
                                    up(w) : False;
                                Endcases;
                    Endcases;

    val_intent_start : Cases x.start.intent OF
                           bot   : False;
                           up(w) : w.val = x.val;
                       Endcases;

    commit_intent : Cases x.intent OF
                        bot   : x.commit ∼ x;
                        up(v) : x.commit ∼ v;
                    Endcases;

        ⋮    % the assertion val_add is in Figure 6

  Constructor
    new : Self;

  Creation
    val_new : new.val = zero;

    intent_new : Cases new.intent OF
                     bot   : True;
                     up(w) : False;
                 Endcases;

End NewValueLogTransactionCommit
```

Fig. 5. Part of the specification for new value logging

```
val_add : Forall( n : int ) .
  Cases x.add(n) OF
    Interrupt(y) :  y.val = x.val    And
                    Cases y.intent OF
                        bot   : True;
                        up(w) : False;
                    Endcases;
    Succeed(z)   :  z.val = x.val    And
                    Cases x.intent OF
                        bot   : Cases z.intent OF
                                    bot   : False;
                                    up(w) : w.val = x.val + n;
                                Endcases;
                        up(v) : Cases z.intent OF
                                    bot   : False;
                                    up(w) : w.val = v.val + n;
                                Endcases;
                    Endcases;
  Endcases;
```

Fig. 6. The assertion val_add for new value logging

the constant bot : Lift[τ] and the injection up : $\tau \to$ Lift[τ]. So x.intent equals either bot (holding no state) or up(v) for some v : Self. In the following the method intent is viewed more like a field that is either empty or holds a state with a partially applied transaction.

The assertion val_start expresses that —as in the abstract specification— starting a transaction does not change the value. The next assertion intent_intent specifies that, if the intent field holds a state v, then the intent field of v itself is always empty. So there is no chain of intents. The assertion val_intent_start requires that immediately after starting a transaction in state x the intent field must hold some object w and the value of x and that of w must be equal.

The assertion commit_intent regulates committing. The tilde $\sim$ is the ASCII version of the bisimilarity relation $\underline{\leftrightarrow}$ in CCSL. The assertion says that in case the intent field is empty (no transaction has been started) then there is no visible change. Otherwise the value of the intent field is copied into the current state. This is the heart of the matter. Note, that this implicitly resets the intent field. Note further, that we assume that the copying is atomic (it might be implemented by changing one pointer), so it can never fail. This is also reflected in the type of commit.

As in the abstract specification we assume one initial state new and require additionally that the intent field of the initial state is empty.

The assertion val_add (in Figure 6) regulates the behaviour of add. It is the longest assertion in the NewValueLogTransactionCommit specification. It looks complicated because of the nested case distinction, but there is nothing deep going on here: Independent of success or failure of add we require that the value field does not change. Further, if the method add fails then the intent must get reset. Otherwise the value of the intent field is increased appropriately. If the user forgot to start a transaction before the first add, then the intent field is silently initialised.

Proving the Refinement

It is quite obvious that every model of the NewValueLogTransactionCommit specification implements a correct transaction mechanism with respect to the TransactionCommitAbstract specification. To prove the refinement formally in PVS we use assertional refinement as described in Subsection 4.2 (that is, for Definition 4.3 we use truth as an invariant). The translation of the concrete methods in the abstract ones is obvious, we only have to adopt the type of commit. In PVS this looks as follows:

```
abstract(c) : TransactionCommitAbstractSignature[Self] = (#
        val    := val(c),
        add    := add(c),
        start  := start(c),
        commit := (Lambda(x:Self) :
                       Cases intent(c)(x) OF
                         bot   : Interrupt(x),
                         up(w) : Succeed(w)
                       Endcases)    #)
```

As explained before TransactionCommitAbstractSignature is the type of coalgebras that match the signature of the abstract specification. In the context of the listed PVS source c is declared as a variable that holds a coalgebra of a model of NewValueLogTransactionCommit. (In particular the type of c contains the assumption that c fulfils the assertions of NewValueLogTransactionCommit.) The above PVS source defines the (higher-order) function abstract that transforms a coalgebra of type NewValueLogTransactionCommit into one of type TransactionCommitAbstract. When the CCSL compiler computes the semantics of a specification it does not combine all methods into one coalgebra. Instead it uses a record with one field for each method. The special brackets (# $\cdots$ #) are PVS syntax for constructing an element of record type. We set the val field to the val field of c, which holds the coalgebra for the method val, and so on. For the commit method the types of the abstract and concrete specification do not match, therefore we use a lambda expression that adds appropriate tags. The translation of the initial states is the identity.

It remains to prove (in PVS) that the function abstract has the required dependent type, that is that the abstract assertions hold on abstract(c). This

```
Begin OkOrFail : ADT
  Constructor
    Ok : Carrier;
    Fail : Carrier;
End OkOrFail
```

Fig. 7. The two element type OkOrFail

can be done by proving that the formula

TransactionCommitAbstractModel?(abstract(c), new)

is true. This refinement is fairly complex because of the many cases that may occur. Still, the PVS proof of this statement is quite easy: altogether it consists of less than 100 PVS prover commands.

7.3 Refinement 2: Old Value Logging

This section describes a refinement of the abstract transaction mechanism that follows the old value logging approach. The refinement in the last section was quite abstract: it added only a few details to the abstract specification. The refinement of this section is more concrete: it fills in much more implementation details. First we specify a simple automaton that can hold one integer (for the value field) and an integer list (to save the add's since the start of the transaction). On top of this automaton we implement the methods that are required for the abstract transaction mechanism. This kind of extension of a specification is directly supported in CCSL through inheritance. A second difference is that we use now the type OkOrFail from Figure 7 to model possible failures.

Let us start with the simple automaton, its specification is in Figure 8. Via the methods val and args one can access the two fields that hold an integer and a list of integers, respectively. There is a method set_val to set the integer field, a method add_arg to add one integer to the list, and a method reset_args to set the list to nil. We take the point of view that the methods set_val and reset_args are atomic, because they require one word of memory to be written. The method add_arg must allocate a cons cell on the heap. This is more complex operation, which can be interrupted. In this case add_arg returns a pair consisting of the tag Fail and a successor state.

The assertions of TransactionState are the obvious ones. For instance reading the integer field after setting it to n yields n. For add_args we specify that in case it fails with Fail then the integer list obtained by args does not change. Otherwise the provided argument n is added at the front of the list.

The specification OldValueLogTransactionCommit extends the specification TransactionState. In CCSL this can be accomplished by inheriting from Transac-

```
Begin TransactionState : ClassSpec
  Private Method
      val        : Self -> int;                        % value field
      set_val    : [Self, int] -> Self;                % set value field
      args       : Self -> list[int];                  % intended adds
      add_arg    : [Self,int] -> [OkOrFail,Self];      % cons to args field
      reset_args : Self -> Self;                       % set args to nil

  Assertion
      val_set_val : Forall(n : int) . x.set_val(n).val = n;

      args_set_val : Forall(n : int) . x.set_val(n).args = x.args;

      args_add_arg : Forall(n : int) .
        Cases Proj_1( x.add_arg(n) ) OF       % Proj_1/2 are the projections
          Ok   : Proj_2( x.add_arg(n) ).args = cons(n, x.args);
          Fail : Proj_2( x.add_arg(n) ).args = x.args;
        Endcases;

      val_add_arg : Forall(n : int) .
        Proj_2( x.add_arg(n) ).val = x.val;

      args_reset_args : x.reset_args.args = null;        % null is the empty list

      val_reset_args : x.reset_args.val = x.val;

  Constructor
      new_state : Self;

  Creation
      val_new : new_state.val = 0;

      args_new : new_state.args = null;

end TransactionState
```

Fig. 8. The automaton that implements old value logging

tionState. Inheritance in CCSL is very similar to inheritance in object-oriented programming languages: The material of the ancestor specification is included in the heir without copying it.

The source code of OldValueLogTransactionCommit is in Figure 9. The inheritance clause renames two methods to avoid name clashes. Through inheritance all method declarations and all assertions from TransactionState are also present in OldValueLogTransactionCommit (constructors and creation conditions are not propagated). Next the specification declares the methods for the transaction mechanism. Their behaviour is now defined in terms of the TransactionState automaton. However, CCSL does not (yet) support definitions in this sense. Therefore we add axioms, one for each method, that describe their behaviour. For instance, the method start should simply reset the internal args attribute via the method reset_args. The assertion start_def

```
Begin OldValueLogTransactionCommit : ClassSpec
  Inherit From TransactionState
    Renaming val AS state_val And set_val AS state_set_val

  Method
    val    : Self -> int;                          % current value
    add    : [Self, int] -> [OkOrFail, Self];      % the action
    start  : Self -> Self;                         % start of transaction
    commit : Self -> [OkOrFail,Self];              % end of transaction
  Assertion
    val_def : x.val = x.state_val;

    start_def  : x.start ∼ x.reset_args;

    add_def : Forall(n : int) .
        x.add(n) ∼ Cases Proj_1( x.add_arg(n) ) of
                      Ok   : x.add_arg(n);
                      Fail : (Fail, Proj_2( x.add_arg(n) ).reset_args);
                   Endcases;

    commit_def :
          ⋮    % the assertion commit_def is in the text
  Constructor
    new : Self;
  Creation
    val_new : new.val = zero;
    args_new : new.args = list[int].null;
End OldValueLogTransactionCommit
```

Fig. 9. The signature for the old value logging refinement

uses bisimilarity to express this.

In the assertion add_def the type on both sides of the tilde is OkOrFail$\times$Self. In this case the CCSL compiler uses the relation lifting from Section 2 to lift bisimilarity to the required type. The effect is as follows. If the inherited method add_arg succeeds then also add succeeds and the result state of add must be bisimilar to that of add_arg. If add_arg fails then also add fails and to obtain the successor state of add we reset the list field via reset_args.

The definition of commit is a bit more complicated. To implement commit we have to walk through the list that is saved in the args field and add them one by one to the the integer in state_val. We do that with the functional reduce (which is sometimes also called fold_right). For a constant $a : \sigma$ and a function $f : \tau \times \sigma \longrightarrow \sigma$ it is defined as:[4]

$$\mathsf{reduce}(a, f)\ (l) \quad = \quad \begin{cases} a & \text{if } l \text{ is the empty list} \\ f\big(b,\ \mathsf{reduce}(a, f)\ (l')\big) & \text{if } l = \mathsf{cons}(b, l') \end{cases}$$

For simplicity we assume that the list processing can not be interrupted. The CCSL source code is then

```
commit_def :
    Let fun = Lambda( n : int, y : Self ) .
                    y.state_set_val( n + y.val ),
        z   = reduce (x, fun) (x.args)
    IN
        x.commit ~ (Ok, z.reset_args);
```

We use a let binding for the function fun that does the iteration over the list x.args. In the state z all the integers have been added to the field state_val. As last we reset the argument list and return Ok.

Proving the Second Refinement

Also the second refinement is not hard to prove. However, it is a bit more complicated to establish it in PVS because it involves some non-trivial reasoning with bisimilarities. The decision procedures of PVS are tuned for equality, and it is not possible to use them with a different congruence relation. Therefore the mechanical proof of the second refinement cannot exploit powerful prover commands.

The translation of the interfaces is as simple as the one for the new value logging refinement. We only have to adopt the types for add and commit:

```
abstr(c) : TransactionCommitAbstractSignature = (#
    val := val(c),

    add := Lambda(x : Self, n : int) :
              Cases Proj_1( add(c)(x,n)) OF
                  Ok   : Succeed( Proj_2( add(c)(x,n)) ),
                  Fail : Interrupt( Proj_2(add(c)(x,n)) )
              Endcases,
    start := start(c),
    commit := Lambda(x:Self) : Succeed( Proj_2( commit(c)(x)) )      #)
```

As before c is a coalgebra that fulfils the assertions of the specification OldValueLogTransactionCommit.

To conclude, this section described a transaction mechanism in the specification language CCSL. It discussed two possible refinements, old value logging

[4] The pair (a, f) can be viewed as a list algebra over σ. Then $\mathsf{reduce}(a, l)$ is the unique algebra morphism $\mathsf{list}[\tau] \longrightarrow \sigma$.

and new value logging. The two refinements have been proved in the theorem prover PVS. For the translation of the CCSL source we used a preliminary version of the CCSL compiler. The source code (including all proofs) is available at the URL `http://wwwtcs.inf.tu-dresden.de/~tews/Transaction`. In addition to the material shown here it includes models for all specifications and also some general results, which have been derived from the specifications.

8 Conclusion

The present paper describes two approaches to define a notion of refinement in the context of coalgebraic specification. The first approach—assertional refinement—works relative to the assertions that restrict the behaviour of the models of a specification: When constructing an assertional refinement one has to show that the assertions of the abstract specification are fulfilled. The second approach—behavioural refinement—works relative to the visible behaviour: In proving a behavioural refinement one has to show that the behaviour of the concrete specification is allowed by the abstract specification.

Assertional refinement has a simpler definition. However, it requires that the full abstract interface is refined. This is sometimes inconvenient, for example, if the abstract interface contains private methods. Behavioural refinement allows to refine only parts of the abstract interface. The generalisation comes not for free: The definition of behavioural refinement leads to more complicated proof obligations. If the complete abstract interface is to be refined then we show (under sensible assumptions on the logic used in the assertions) that both notions of refinement coincide. Further, both notions of refinements are compositional.

The notion of refinement described in the present paper has been applied in several case studies in the context of the coalgebraic specification language CCSL, see for instance [21,16]. Section 7 demonstrates assertional refinement for two different approaches to implement a transaction mechanism: old value logging and new value logging.

References

[1] P. Aczel and P. F. Mendler. A final coalgebra theorem. In D. H. Pitt, D. E. Rydeheard, P. Dybjer, A. M. Pitts, and A. Poigné, editors, *Proceedings of the Conference on Category Theory and Computer Science*, volume 389 of *Lect. Notes Comp. Sci.*, pages 357–365, Berlin, September 1989. Springer.

[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] R.-J. Back and J. von Wright. *Refinement Calculus: A Systematic Introduction.* Graduate Texts in Computer Science. Springer-Verlag, 1998.

[4] J.A. Bergstra, A. Ponse, and S.A. Smolka, editors. *Handbook of Process Algebra*. North-Holland, Amsterdam, 2001.

[5] Z. Chen. *Java Card Technology for Smart Cards*. The Java Series. Addison-Wesley, 2000.

[6] J.A Goguen and R. Burstall. The semantics of CLEAR, a specification language. In *Abstract Software Specification*, number 86 in Lect. Notes Comp. Sci., pages 292–332. Springer, Berlin, 1980.

[7] R. Gorrieri and A. Rensink. Action refinement. In [4].

[8] U. Hensel. *Definition and Proof Principles for Data and Processes*. PhD thesis, Univ. of Dresden, Germany, 1999.

[9] 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.

[10] U. Hensel and B. Jacobs. Coalgebraic theories of sequences in PVS. *Journ. of Logic and Computation*, 9(4):463–500, 1999.

[11] C. Hermida and B. Jacobs. Structural induction and coinduction in a fibrational setting. *Inf. & Comp.*, 145:107–152, 1998.

[12] C.A.R. Hoare. Proof of correctness of data representations. *Acta Informatica*, 1:271–281, 1972.

[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. Exercises in coalgebraic specification. To appear in the proceedings of the *Mathematics for Information Technology* Summer School and Workshop on *Algebraic and Coalgebraic Methods in the Mathematics of Program Construction*, 2000.

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

[17] 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.

[18] B. Lampson and H. Sturgis. Reflections on an operating system design. *Communications of the ACM*, 19(5):251–266, 1976.

[19] N. Lynch and F. Vaandrager. Forward and backward simulations. I. Untimed systems. *Inf. & Comp.*, 121(2):214–233, 1995.

[20] S. Mac Lane. *Categories for the Working Mathematician*, volume 5 of *Graduate Texts in Mathematics*. Springer-Verlag, Berlin, 2nd edition, 1997. (1st ed., 1971).

[21] D. Meyer. A case study in object oriented specification and verification: The MSMIE protocol. Graduation thesis, Catholic University of Nijmegen, January 1999.

[22] R. Milner. *Communication and Concurrency.* Prentice Hall, 1989.

[23] M. Oestreicher. Transactions in Java Card. In *15th Annual Computer Security Applications Conference (ACSAC'99)*, pages 291–298. IEEE, 1999.

[24] 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.

[25] L. C. Paulson. *Isabelle: A Generic Theorem Prover.* Number 828 in Lect. Notes Comp. Sci. Springer, Berlin, 1994.

[26] J. Rothe, H. Tews, and B. Jacobs. The coalgebraic class specification language CCSL. *Journal of Universal Computer Science*, 7(2):175–193, March 2001.

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

[28] H. Tews. Coalgebras for binary methods: Properties of bisimulations and invariants. *Theoretical informatics and applications*, 35(1):83–111, February 2001.

[29] M. Wirsing. Algebraic specification. In J. van Leeuwen, editor, *Formal Models and Semantics*, volume B of *Handbook of Theoretical Computer Science*, chapter 13, pages 673–788. Elsevier/MIT Press, 1990.