Algebraic-coalgebraic specification in CoCasl

Till Mossakowski [a] Lutz Schröder [a] Markus Roggenbach [b]
Horst Reichel [c]

[a] *BISS, Department of Computer Science, University of Bremen*
[b] *Department of Computer Science, University of Wales Swansea*
[c] *Institute for Theoretical Computer Science, Technical University of Dresden*

Abstract

We introduce CoCasl as a light-weight but expressive coalgebraic extension of the algebraic specification language Casl. CoCasl allows the nested combination of algebraic datatypes and coalgebraic process types. Moreover, it provides syntactic sugar for an observer-indexed modal logic that allows e.g. expressing fairness properties. This logic includes a generic definition of modal operators for observers with structured equational result types. We prove existence of final models for specifications in a format that allows the use of certain initial datatypes, as well as modal axioms. The use of CoCasl is illustrated by specifications of the process algebras CSP and CCS.

Key words: Algebraic specification; coalgebra; process algebra; Casl, CCS, CSP.

In recent years, coalgebra has emerged as a convenient and suitably general way of modelling the reactive behaviour of systems [42]. While algebraic specification deals with *inductive datatypes* generated by constructors, coalgebraic specification deals with *coinductive process types* that are observable by selectors, much in the spirit of automata theory. An important role is played here by *final coalgebras*, which are complete sets of possibly infinite behaviours, such as streams or even the real numbers.

For algebraic specification, the Common Algebraic Specification Language Casl [2] has been designed as a unifying standard, while for the much younger field of coalgebraic specification there is still a divergence of notions and notations. The idea pursued here is to obtain a fruitful synergy by extending Casl with coalgebraic constructs that dualize the algebraic constructs already present in Casl.

In more detail, CoCasl provides a basic co-type construct, cogeneratedness

constraints, and structured cofree specifications; moreover, coalgebraic modal logic is introduced as syntactical sugar. Co-types serve to describe reactive processes, equipped with observer operations whose role is dual to that of the constructors of a datatype. Cotypes can be qualified as being cogenerated or cofree, respectively, thus imposing full abstractness and realization of all observable behaviours, respectively.

The modal logic is introduced in two stages. Modal operators are indexed by observer operations, which are thought of as transitions in a state space. Thus, a formula such as $[f]\,\phi$ states that ϕ holds 'necessarily' for the result of observer f. This informal interpretation is easy to capture formally in case the result of f is just a state; the first stage of the modal logic treats precisely this case. It is quite common, however, to have observers with structured results, such as a finite set or a list of states. In the second stage, we give a generalized definition of modal operators for such observers, subject to a certain healthiness condition on the involved datatypes, and we argue that some such restriction is inevitable in the sense that not all datatypes are suitable for modal logic.

The most powerful new CoCasl construct are cofree specifications, which allow specifying final models of arbitrary specifications. Of course, this raises the question for what kinds of specifications such final models actually exist. We provide a sufficient existence condition which covers specifications that employ initially specified datatypes in observer functions and restrict behaviours by modal formulas. This, besides syntactic conciseness, is the main motivation for introducing the modal logic; essentially, our model existence theorem is further support for the claim that modal formulas play the same role in coalgebra as equations do in algebra [21,22].

Finally, we illustrate the use of CoCasl in a typical reactive setting by means of specifying the syntax and semantics of two prominent process algebras, namely CCS and CSP. These two examples serve the dual purpose of providing a proof of concept and giving an idea of how CoCasl relates to other reactive Casl extensions.

The paper is organised as follows. Section 1 introduces the Casl logic; Section 2 provides a brief overview of Casl and the duality between Casl and CoCasl. The various basic process type constructs are discussed in Sections 3, 4, and 5. The semantics of cofree specifications is given in Section 6. Sections 7 and 8 introduce the modal logic for simple and structured observations, respectively. In Section 9, we sketch an institution that hardwires the CoCasl modal logic, as an alternative to its encoding in modality-free CoCasl. Section 10 is devoted to the existence theorem for final models. The specifications of CCS and CSP are described in Section 11. This work is an extended version of [36]; the process algebra example has appeared in [28].

1 CASL

The specification language CASL (*Common Algebraic Specification Language*) has been designed by COFI, the international *Common Framework Initiative for Algebraic Specification and Development* [13]. Its features include first-order logic, partial functions, subsorts, sort generation constraints, and structured and architectural specifications. For the language definition cf. [5,29]; a full formal semantics is laid out in [1]. An important point here is that the semantics of structured and architectural specifications is independent of the logic employed for basic specifications, so that the language is easily adapted to the extension of the logic envisaged here.

We now briefly sketch the many-sorted CASL logic, which can be formalized as an institution [14]. Full details can be found in [29,27]; examples of actual CASL specifications will appear in later sections. A *many-sorted* CASL *signature* Σ consists of a preordered set S of sorts and sets of total and partial function symbols and predicate symbols. Function and predicate symbols are written $f : \bar{s} \to t$ and $p : \bar{s}$, respectively, where t is a sort and $\bar{s}$ is a list $s_1 \dots s_n$ of sorts, thus determining their *name* and *profile*. Symbols with identical names are said to be *overloaded*; they may be referred to by just their names in CASL specifications, but are always qualified by profiles in fully statically analysed sentences. Signature morphisms map the sorts and the function and predicate symbols in a compatible way, such that totality of function symbols is preserved.

Models are many-sorted partial first order structures, interpreting total (partial) function symbols as total (partial) functions and predicate symbols as relations. Homomorphisms between such models are so-called *weak homomorphisms*. That is, they are total as functions, and they preserve (but not necessarily reflect) the definedness of partial functions and the satisfaction of predicates. A homomorphism is called *closed* [9], if it not only preserves, but also reflects definedness and satisfaction of predicates.

A *congruence* R on a model is an equivalence relation that is compatible with the total and partial functions (in the latter case, compatible *on the domain* of the partial function). R is called *closed* [9], if additionally domains of partial functions and domains of satisfaction of predicates are closed under R.

Concerning *reducts*, if $\sigma : \Sigma_1 \to \Sigma_2$ is a signature morphism and M is a Σ_2-model, then $M|_\sigma$ is the Σ_1-model which interprets a symbol by first translating it along σ and then taking M's interpretation of the translated symbol. Reducts of homomorphisms are defined similarly.

Given a signature, *sentences* are built from atomic sentences using the usual features of first order logic. Here, an atomic sentence is either a definedness

assertion, a strong equation, an existence equation, or a predicate application; see [2,1] for details. There is an additional type of sentence that goes beyond first-order logic: a *sort generation constraint* states that a given set of sorts is generated by a given set of functions, i.e. that all the values of the generated sorts are reachable by some term in the function symbols, possibly containing variables of other sorts. Every CASL specification Sp generates, along with its signature Σ, a set Γ of sentences; together, these determine the *theory* (Σ, Γ) generated by Sp.

The *subsorted* CASL institution is defined on top of the many-sorted one. Here, signatures are equipped with a pre-order on the sorts, the subsorting relation, and terms of a subsort may be used in places where a term of the supersort is expected. Moreover, there are partial projection functions from supersorts to subsorts, and membership predicates detecting whether an element of a sort is in a given subsort. Such a signature is translated to a many-sorted one by adding total injection functions of subsorts into supersorts and compatibility axioms for the extra infrastructure. Sentences, models and satisfaction are then defined in terms of the translated (many-sorted) signature.

Based on this institution, CASL itself additionally provides, for the sake of conciseness, a number of abbreviated constructs, most prominently for defining algebraic datatypes. CASL's datatype features are briefly recalled below, in direct comparison to the corresponding COCASL constructs. At the level of structured specifications, CASL offers parametrized named specifications, unions of specifications (keyword **and**), extensions of specifications (keyword **then**), free specifications **free {** ... **}**, and renaming as well as hiding of symbols. Generally, CASL employs *linear visibility*, e.g. all symbols must be declared before they can be used. The signature provided for a particular item (declaration or sentence) in a specification is called its *local environment* — it consists of all the declarations that have been made before the item.

2 An overview of COCASL

As indicated in the introduction, COCASL extends CASL at two levels: it enriches the logic available for *basic specifications*, and it introduces an additional *structuring* concept, namely, cofree specifications. *Architectural specifications* remain as in CASL. Figure 1 contains a summary of the CASL concepts and their COCASL dualizations.

At the level of basic specifications, the duality addresses the various forms of the **types** construct in CASL. In its elementary form, its dual is the **cotypes** construct, which serves to specify process types with observers (we shall reserve the word 'datatype' for the algebraic **types**); c.f. Section 3. In CASL,

Algebra	**Coalgebra**
type = (partial) algebra	**cotype** = coalgebra
constructor	observer (=selector)
generation	observability
generated type	**cogenerated (co)type**
= no junk	= full abstractness
= induction principle	= coinduction principle
no confusion	all possible behaviours
free type	**cofree cotype**
= absolutely initial datatype	= absolutely final process type
= no junk +	= full abstractness +
no confusion	all possible behaviours
free { ... } = initial datatype	**cofree { ... }** = final process type

Fig. 1. Summary of dualities between CASL and COCASL.

a **type** declaration can be strengthened in two ways. In a **generated type**, junk is excluded, while a **free type** additionally forbids confusion. Dually, we introduce a **cogenerated cotypes** construct for fully abstract process types (Section 4), as well as a **cofree cotypes** construct, which additionally requires that all possible observable behaviours are realized in the process type; cf. Section 5. (Intercombinations such as **cofree types** etc. are not provided, and their emulation is expressly discouraged.) Moreover, we introduce a modal logic for axioms about state evolution in process types as syntactical sugar (Sections 7 and 8).

At the level of structured specifications, we dualize the structured **free** construct to a structured **cofree** construct (Section 6) which equips arbitrary specifications with a final semantics, thus capturing one of the central notions of coalgebra. Like its dual, this construct is powerful enough to introduce inconsistencies, since final models of arbitrary specifications may fail to exist (while a cofree cotype, like a free type, is a conservative extension as long as the declared types are fresh). We do however provide a rather general existence theorem which guarantees conservativity of cofree extensions for specifications that adhere to a certain format allowing in particular modal logic formulas and structured observations using free specifications nested within a cofree specification (Section 10); examples are provided to show that conservativity may fail for many other formats, in particular for cofree specifications nested

```
spec CONTAINER [sort Elem] =
  type
      Container ::= empty | insert(first :? Elem; rest :? List[Elem]);
spec NTREE [sort Elem] =
  types
      NTree ::= fork(Elem; Forest)
      Forest ::= null | grow(Tree; Forest)
```

Fig. 2. Some **type** definition in CASL .

within free specifications.

3 Type and cotype definitions

The basic CASL construct for type definitions is the **type** construct. It declares constructors and, optionally, selectors for a datatype (or several datatypes at once); both constructors and selectors may be partial. Such a type declaration is expanded into the declaration of the constructor and selector operations and axioms relating the selectors and constructors. Nothing else is said about the type; thus, there may not only be 'junk' and 'confusion', but there may also be rather arbitrary behaviour of the selectors outside the range of the corresponding constructors. Consider e.g. the specification of containers in Figure 2. It declares sorts *Elem* and *Container*. The type *Container* is declared to have two *alternatives*, one of them given by a total constructor constant $empty : Container$, the other one given by a total constructor function $insert : Elem \times Container \to Container$, together with two partial selector functions $first : Container \to? Elem$ and $rest : Container \to? Container$. Also, two axioms ensuring $first(insert(x, y)) = x$ and $rest(insert(x, y)) = y$ are generated. Note that even if one fixes the interpretation of the sort *Elem*, this specification is rather loose: the sort *Container* may be interpreted e.g. either as the set of finite lists or the set of infinite lists (over *Elem*). Further note that also declarations of mutually recursive types are possible (specification NTREE in Figure 2) — but only within a single **types** construct.

In COCASL, the **types** construct is complemented by the **cotypes** construct. The syntax of this construct is nearly identical to the **type** construct; e.g., one may write

```
cotype Process ::= cont(hd1 :?Elem; next :?Process)
                 | fork(hd2 :?Elem; left :?Process; right :?Process)
```

thus determining constructors and selectors as for types. However, for cotypes, the constructors are optional and the selectors (which we henceforth call observers) are mandatory. The latter requirement rules out CASL's sort alternatives making a given sort a subsort of the declared type, as in

$$\textbf{type}\ \ Int ::= \textbf{sort}\ Nat \mid\ -\_\_(Pos)$$

Moreover, we also allow additional parameters for the observers. These have to come from the local environment (recall that this consists of all the declarations before the cotype):

spec MOORE =
  sorts In, Out
  cotype $State ::= (next : In \to State; observe : Out)$
end

The **cotype** definition in this case expands to

sort $State$
ops $next\ :\ In \times State \to State;$
  $observe\ :\ State \to Out$

Observers with additional parameters cannot have a corresponding constructor, since the constructor would need to have a higher-order type, e.g. in the abvoe example $(In \to State) \to State$.

Last but not least, the **cotype** construct introduces a number of additional axioms concerning the domains of definition of the observers, besides the axioms relating constructors with their observers as for types:

- definedness of observers is independent of the additional parameters,
- the domains [1] of two observers in the same alternative are the same,
- the domains of two observers in different alternatives are disjoint, and
- the domains of all observers of a given sort are jointly exhaustive.

Thus, the alternatives in a cotype are to be understood as parts of a disjoint sum, so that cotypes, unlike types, correspond directly to coalgebras (see Prop. 2 below).

Definition 1 A cotype in COCASL is given by the local environment sorts and the family of observers

$$CT = (S, (obs_{i,j,k} : T_i \to T_{i,j,k})_{i=1...n, j=1...m_i, k=1...r_{i,j}}).$$

[1] Since definedness of observers is independent of the additional parameters, by "domain" we here mean the set of values of the cotype for which the observer is defined.

Here, S is a set of sorts (the local environment sorts, also called observable sorts), $T_1 \ldots T_n$ are the newly declared process types (or non-observable sorts) in the cotype (which possibly involve mutual recursion like in Figure 8), and $obs_{i,j,k}$ is the k-th observer of the j-th alternative in the **cotype** definition of T_i. $T_{i,j,k}$ is the result sort of the observer; it may be either one of the T_i or one of the local environment sorts in S. Next, consider observers with additional parameters. In a **cotype** declaration, they are written $obs_{i,j,k} : s_1 \times \cdots \times s_m \to s$, where $s_1 \ldots s_m$ come from S and s either is one of the T_i or comes from S as well. In order to keep the format $obs_{i,j,k} : T_i \to T_{i,j,k}$ for the type of the observer, the corresponding $T_{i,j,k}$ is not simply a sort, but a function space

$$s_1 \times \cdots \times s_m \to s,$$

and the observer, normally having type $obs_{i,j,k} : s_1 \times \cdots \times s_m \times T_i \to s$, by currying can be equivalently considered to have the higher order type

$$obs_{i,j,k} : T_i \to (s_1 \times \cdots \times s_m \to s),$$

which is just $T_i \to T_{i,j,k}$. Although higher-order functions are not available in CoCasl, we prefer this notation for uniformity reasons. Still, the signature $Sig(CT)$ of a cotype CT is a first-order signature consisting of the local environment sorts S, the cotype sorts $T_1 \ldots T_n$, and the first-order profiles of the observers.

Note that within cotypes, also constructors may be declared. However, we ignore them here, since they do not contribute to the coalgebra structure. However, they *do* play a role when homomorphisms are concerned, which is why we exclude them in the next proposition:

Proposition 2 *To a given* CoCasl **cotype** *definition without constructors, which, say, induces the theory* (Σ, Γ)*, one can associate a functor* $F : \mathbf{Set}^n \to \mathbf{Set}^n$ *such that the category of partial* (Σ, Γ)*-algebras is isomorphic to the category of* F*-coalgebras. In particular, this implies that all homomorphisms between partial* (Σ, Γ)*-algebras are closed.*

PROOF. We begin with the parameterless case, without any local environment. Given a cotype

$$CT = (\emptyset, (obs_{i,j,k} : T_i \to T_{i,j,k})_{i=1\ldots n, j=1\ldots m_i, k=1\ldots r_{i,j}}),$$

by abuse of notation, we also treat the T_i as set variables in the definition of the functor $F : \mathbf{Set}^n \to \mathbf{Set}^n$:

$$F(T_1, \ldots, T_n) = (\coprod_j \prod_k T_{1,j,k}, \ldots, \coprod_j \prod_k T_{n,j,k}),$$

We now have to prove the stated isomorphism of categories. The above condition that the domains of two observers in the same alternative are the same ensures that we can group the observers of each alternative to a partial function into a product:

$$T_i \hookleftarrow dom\, f_{i,j} \xrightarrow{f_{i,j} :=< obs_{i,j,1}, \ldots, obs_{i,j,r_{i,j}} >} \prod_k T_{i,j,k}$$

Moreover, the other two conditions (the domains of two observers in different alternatives being disjoint, and the domains of all observers of a given T_i being jointly exhaustive) just ensure that

$$T_i = \coprod_j dom\, f_{i,j},$$

hence we can group the $f_{i,j}$ and obtain

$$T_i = \coprod_j dom\, f_{i,j} \xrightarrow{g_i := \coprod_j f_{i,j}} \coprod_j \prod_k T_{i,j,k}.$$

Hence, we arrive at a coalgebra on $\mathbf{Set}^n$

$$(T_1, \ldots, T_n) \xrightarrow{(g_1, \ldots, g_n)} F(T_1, \ldots, T_n).$$

It is easy to reverse this process: given the g_i, the $f_{i,j}$ are obtained by pulling back (i.e., taking the inverse image):

$$\begin{array}{ccc} T_i & \xrightarrow{g_i} & \prod_j \coprod_k T_{i,j,k} \\ \uparrow & & \uparrow \coprod_j \\ dom\, f_{i,j} & \xrightarrow{f_i, j} & \coprod_k T_{i,j,k} \end{array}$$

and the $obs_{i,j,k}$ are obtained by composing with the projections. Altogether, this leads to a bijective correspondence between (Σ, Γ)-algebras and F-coalgebras. Furthermore, we show that this correspondence is functorial: Given a homomorphism $h = (h_1, \ldots, h_n) : M \to N$ of partial (Σ, Γ)-algebras, let $h_{i,j,k}$ be the same selection among the $(h_1, \ldots, h_n)$ as $T_{i,j,k}$ is among the $(T_1, \ldots, T_n)$. Now the partial algebra homomorphism conditions

$$\begin{array}{ccc} dom\, f^M_{i,j} & \xrightarrow{sel^M_{i,j,k}} & T^M_{i,j,k} \\ \downarrow{\scriptstyle h_i|_{dom\, f^M_{i,j}}} & & \downarrow{\scriptstyle h_{i,j,k}} \\ dom\, f^N_{i,j} & \xrightarrow{sel^N_{i,j,k}} & T^N_{i,j,k} \end{array}$$

when grouped together along the products and coproducts yield just the coalgebra morphism conditions

$$\begin{array}{ccc} T_i^M & \xrightarrow{g_i^M} & \coprod_j \prod_k T_{i,j,k}^M \\ {\scriptstyle h_i}\downarrow & & \downarrow{\scriptstyle \coprod_j \prod_k h_{i,j,k}} \\ T_i^N & \xrightarrow{g_i^N} & \coprod_j \prod_k T_{i,j,k}^N \end{array}$$

Again, this process can be easily reversed. The crucial point here is to show that the restriction of h_i to $dom\, f_{i,j}^M$ acutally ends up in $dom\, f_{i,j}^N$. By construction of $dom\, f_{i,j}$, this means that $g_i^M(x)$ ending in the j-th coproduct summand implies that $g_i^N(h_i(x))$ ends in the j-th coproduct summand — but this follows easily from the coalgebra morphism condition, since $\coprod_j \prod_k h_{i,j,k}$ acts summand-wise. Moreover, the latter even entails the converse: $h_i(x) \in dom\, f_{i,j}^N$ implies $x \in dom\, f_{i,j}^M$, showing that the resulting homomorphism of partial algebras is closed.

The proposition easily generalizes to the case that some of the $T_{i,j,k}$ come from the local environment or are function spaces (cf. Def. 1). For both cases, the proof goes through with minor adaptations. □

Definition 3 Let a set S of sorts (called observable sorts) and a signature Σ such that S is contained in the sorts of Σ, and a Σ-model M be given. A binary relation R on M is called an (S, Σ)-*bisimulation*, if it

- is the equality relation on sorts in S, and
- satisfies the closed congruence property for the operations and predicates in Σ.

Two elements of M are called (S, Σ)-*bisimilar*, if they are in relation for some bisimulation.

These notions easily carry over to cotypes: A CT-bisimulation for a cotype

$$CT = (S, (obs_{i,j,k} : T_i \to T_{i,j,k})_{i=1\ldots n, j=1\ldots m_i, k=1\ldots r_{i,j}})$$

is just a $(S, Sig(CT))$-bisimulation.

Note that Prop. 2 above does *not* hold in presence of constructors. It would hold if one required that the homomorphism equations for constructors only hold up to bisimilarity — or if we restrict ourselves to fully abstract algebras in the sense defined below. For example, the above process cotype without constructors

cotype $Process ::= (hd1 :?Elem; next :?Process) \mid (hd2 :?Elem; left :?Process; right :?Process)$

has a category of partial algebras that is isomorphic to the category of coalgebras for the functor

$$F(P) = E \times P + E \times P \times P$$

where E is the fixed interpretation of the sort *Elem*. In presence of the constructors *cont* and $fork$ as in the original process specification above, we get a subcategory of the original category of partial algebras — namely that consisting only of those homomorphisms which preserve *cont* and $fork$. That said, one shold note that this does not make a difference in terms of final models (see below), since final models are fully abstract and hence the coalgebra homomorphisms into the final model are automatically compatible with the constructors.

4 Generation and cogeneration constraints

In order to exclude 'junk' from models of datatypes, CASL provides generatedness constraints that essentially introduce (higher order) implicit induction axioms. E.g., a typical specification of finite sets would require the type of finite sets to be generated by the constant denoting the empty set and an operation for addition of elements:

spec FINITESET [**sort** *Elem*] =
  generated type $FinSet[Elem] ::= \{\} \mid \_\_ + \_\_(Elem; FinSet[Elem])$;

Here, the generatedness constraints means that all finite sets can be constructed by applications of $\{\}$ and $\_\_ + \_\_$.

Dually to this, COCASL introduces *cogeneratedness constraints* that amount to an implicit coinduction axiom and thus restrict the models of the type to fully abstract ones. This means that equality is the largest congruence w.r.t. the introduced sorts, operations and predicates (excluding the constructors). Put differently, everything that cannot be distinguished by its behaviour, as determined by the observers and the predicates, is identified (where observations can only be made on sorts in the local environment, i.e. outside the type declaration itself). In the example in Figure 3, the STREAM-models are (up to isomorphism) the subsets of E^ω that are closed under tl, where E is the interpretation of the sort *Elem*. (Note: since there is only one alternative, there is no difference between a type and a cotype here.)

spec STREAM1 [**sort** *Elem*] =
  cogenerated cotype
    Stream ::= *cons*(*hd* : *Elem*; *tl* : *Stream*)
end

Fig. 3. Cogenerated specification of bit streams in CoCASL

.

A more complex example is the specification of CCS – see Section 11. States are generated by the CCS syntax, but they are identified if they are bisimilar w.r.t. the ternary transition relation. This can be expressed in CoCASL by stating that states are cogenerated w.r.t. the transition relation.

Formally, a *cogeneration constraint* over a signature $\Sigma = (S, TF, PF, P)$ is a subsignature fragment (i.e. a tuple of component-wise subsets that need not by itself form a complete signature) $\bar{\Sigma} = (\bar{S}, \bar{TF}, \bar{PF}, \bar{P})$ of Σ. In the above example, the cogeneration constraint is $(\{Elem\}, \{hd, tl\}, \emptyset, \emptyset)$. The constraint $\bar{\Sigma}$ is *satisfied* in a Σ-model M if each $(\bar{S}, (S, \bar{TF}, \bar{PF}, \bar{P}))$-bisimulation on M is the equality relation.

In duality to generated types in CASL, the construct **cogenerated cotype ...** abbreviates **cogenerated {cotype ...}**. No such abbreviation is provided for **cogenerated {type ...}**, the use of which is in fact expressly discouraged (as are **generated {cotypes ...}**). A particularly deterrent example for the use of types where cotypes are expected is given in Example 11.

A cogenerated cotype involving observers with additional parameters is that of fully abstract Moore automata:

spec FULLYABSTRACTMOORE =
  sorts *In*, *Out*
  cogenerated cotype *State* ::= (*next* : *In* → *State*; *observe* : *Out*)
end

Remark 4 Note that observers of cotypes always have exactly one non-observable argument. However, like the **generated { ... }** construct in CASL, the **cogenerated { ... }** construct allows the inclusion of arbitrary signature items in the cogeneratedness constraint, so that observers of arbitrary arity are also possible. In particular, full abstractness for binary observers in the sense of [48] (i.e. observers with two non-observable argument sorts) is expressible.

Remark 5 At the level of model homomorphisms, the duality between generatedness and cogeneratedness constraints becomes formally a lot clearer: a generatedness constraint essentially amounts to a weakened form of initiality in the sense that a model M of the corresponding specification is *pre-initial* in the fibre over its reduct to the local environment (cf. Definition 9 below)

```
%list [__], nil, __ :: __
%prec { __ :: __ } < { __ ++ __ }

spec LIST [sort Elem] =
  free type
      List[Elem] ::= nil | __ :: __(head :? Elem; tail :? List[Elem]);
  op __ ++ __ : List[Elem] × List[Elem] → List[Elem];
  forall e : Elem; K, L : List[Elem]
      • nil ++ L = L                                  %(concat_nil)%
      • (e :: K) ++ L = e :: K ++ L                   %(concat_cons)%
end
```

Fig. 4. Specification of lists over an arbitrary element sort in CASL.

— i.e. there is at most one morphism from M into any other model over the same reduct. Dually, a model M that satisfies a cogeneratedness constraint is *pre-final* in its fibre in the sense that there exists at most one morphism from any other model over the same reduct into M. This may also roughly be expressed as follows: generated models do not have proper substructures, and cogenerated models do not have proper quotients.

5 Free types and cofree cotypes

CASL allows the exclusion not only of 'junk' in datatypes, but also of 'confusion', i.e. of equalities between different constructor terms. To this end, it provides the (basic) **free type** construct. Free datatypes carry implicit axioms that state, beside term-generatedness, the injectivity of the constructors and the disjointness of their images. E.g., the specification of lists over an element sort given in Figure 4 gives rise to axioms that state that *nil* is not of the form $x :: l$, and that $x_1 :: l_1 = x_2 :: l_2$ implies $x_1 = x_2$ and $l_1 = l_2$. The most immediate effect of these axioms is that *recursive definitions on a free datatype are conservative*. The elements of a free datatype can be thought of as being the (finite) constructor terms, i.e. in a suitable sense finite trees.

In COCASL, we provide, dually, a **cofree cotypes** construct that specifies the absolutely final coalgebra of infinite behaviour trees (see Example 11 on why there is no **cofree types** construct). More concretely, this means that, in addition to cogeneratedness, there is also a principle stating that there are enough behaviours, namely all infinite trees [4] (with branching as specified by the observers). In contrast to its dual (no confusion among constructors), the latter principle cannot be expressed in first-order logic; however, a second-order specification is possible (see below). In the example in Figure 5, the STREAM2-models are isomorphic to E^ω, where E is the interpretation of the

```
spec STREAM2 [sort Elem] =
  cofree cotype
      Stream ::= (hd : Elem; tl : Stream)
end
```

Fig. 5. Cofree specification of bit streams in CoCASL.

```
spec FUNCTIONTYPE =
  sorts A, B
  cofree cotype
      Fun[A, B] ::= (eval : A → B)
end
```

Fig. 6. Cofree specification of function types.

```
spec FINALMOORE1 =
  sorts In, Out
  cofree cotype State ::= (next : In → State; observe : Out)
end
```

Fig. 7. Cofree specification of the final Moore automaton.

```
spec INFTREE [sort Elem] =
  cofree cotypes
    InfTree    ::= (label : Elem; children : InfForest)
    InfForest  ::= (first :? InfTree; rest :? InfForest)
end
```

Fig. 8. Cofree specification of trees of possibly infinite depth and branching.

sort *Elem*. An example with an extra parameter for the observer is the specification of function types in Figure 6 (actually, this shows that higher-order types can be easily encoded in CoCASL). Similarly, Figure 7 specifies the final Moore automaton. Finally, in Figure 8 we use mutually recursive cofree cotypes to specify trees of possibly infinite depth and branching, dualizing the *Ntree* example of Figure 2 (note that finite trees are included since the observers *first* and *rest* are partial).

Definition 6 Given a set of sorts S, an *S-colouring* is just an S-sorted family of sets (of colours).

We are now ready to dualize the important algebraic concept of term algebra.

Definition 7 Given a cotype

$$CT = (S, (obs_{i,j,k} : T_i \to T_{i,j,k})_{i=1\ldots n, j=1\ldots m_i, k=1\ldots r_{i,j}})$$

and an S-colouring C, the *behaviour algebra* $Beh_{CT}(C)$ is defined to be the following $Sig(CT)$-algebra:

- the carriers for observable sorts (i.e. in S) are those determined by C;
- the carriers for a non-observable sort T_{i_0} consist of all infinite trees of the following form:
  - each inner node is labelled with a pair (T_i, j), where T_i is a non-observable sort and $j \in \{1 \ldots m_i\}$ selects an alternative out of those for T_i;
  - the root is labelled with (T_{i_0}, j_0) for some j_0;
  - each leaf is labelled with an observable sort $s \in S$ and some colour from C_s;
  - each non-leaf node with label (T_i, j) has one child for each of the observers $obs_{i,j,k}$ $(k = 1 \ldots r_{i,j})$ and each tuple of colours for the extra parameters of the observer. The child node is labelled with the result sort of the observer.
- an observer operation $obs_{i_0,j,k}$ is defined for a tree with root (T_{i_0}, j_0) if and only if $j = j_0$, and in this case, it just selects the child tree corresponding to the observer and the argument colours for the extra parameters of the observer.

Proposition 8 *Given a cotype*

$$CT = (S, (obs_{i,j,k} : T_i \to T_{i,j,k})_{i=1\ldots n, j=1\ldots m_i, k=1\ldots r_{i,j}})$$

and an S-colouring C, the behaviour algebra $Beh_{CT}(C)$ *is* final *in the following sense: for any $Sig(CT)$-algebra A equipped with a C-colouring h (that is, a family of maps $(h_s : A_s \to C_s)_{s \in S}$), we can extend h in a unique way to a $Sig(CT)$-homomorphism*

$$h^\natural : A \to Beh_{CT}(C).$$

PROOF. Using the characterization of Prop. 2, the result follows from the general construction of final coalgebras over the category of $\{T_1, \ldots, T_n\}$-sorted sets (this generalizes the well-known result for **Set** [4]). Intuitively, $h^\natural$ constructs the behaviour of an element, which is the infinite tree given by all possible observations that can be made successively applying the observers until a value of observable sort (i.e. in S) is reached. □

Given a signature Σ, we formally add cofreeness constraints of form $cofree(CT)$, where

$$CT = (S, (obs_{i,j,k} : T_i \to T_{i,j,k})_{i=1\ldots n, j=1\ldots m_i, k=1\ldots r_{i,j}})$$

is a cotype with $Sig(CT) \subseteq \Sigma$, as Σ-sentences to our logic. A cofreeness constraint $cofree(CT)$ holds in a Σ-algebra A, if the reduct of A to $Sig(CT)$ is isomorphic to the behaviour algebra $Beh_{CT}(C)$ over the set of colours C with $C_s := A_s$ for $s \in S$.

Note that this implies the satisfaction of the cogeneratedness constraint $(S, \{sel_{i,j,k} | sel_{i,j,k} \text{ total}\}, \{sel_{i,j,k} | sel_{i,j,k} \text{ partial}\}, \emptyset)$, i.e. each cofree cotype is also cogenerated. The converse does not hold, i.e. a cogenerated cotype need not be cofree. However, cogenerated *cotypes* still behave quite nicely (in contrast to arbitrary cogenerated *types*): the elements of carriers of the non-observable sorts (i.e. those outside S) are completely determined by their *behaviours*. Thus, the elements can be identified with their bevaviours, and up to isomorphism, we have a subalgebra of the cofree cotype. Hence, cofreeness essentially adds the requirement that each possible behaviour is actually represented by an element.

Full abstractness of cofree types implies that cofreeness is not destroyed in the presence of constructors. Normally, constructors are determined only up to bisimilarity and hence may destroy the homomorphism condition. However, in the cofree model, bisimilarity is just equality.

The main benefit of cofree cotypes (in comparison to cogenerated cotypes) is the principle

corecursive definitions in cofree cotypes are conservative.

This completes the definition of CoCasl constraint sentences. Note that in order to be able to translate the various constraints along signature morphisms in a way that the satisfaction condition for institutions is fulfilled, one has to equip the constraints with an additional signature morphism, as in [1,27].

6 Structured free and cofree specifications

Besides institution-specific language constructs, Casl also provides institution-independent structuring constructs. In particular, Casl provides the structured **free** construct that restricts the model class to *initial* or *free* models. That is, if Sp_1 is a specification with signature Σ_1, then the models of Sp_1 **then free** $\{Sp_2\}$ are those models M of Sp_1 **then** Sp_2 that are free over $M|_{\Sigma_1}$ w.r.t. the reduct functor $\_|_{\Sigma_1}$. This allows for the specification of datatypes that are generated freely w.r.t. given axioms, as, for example, in the specification of finite sets over an state sort which is part of the specification of nondeterministic automata in Figure 9. Here, the **assoc**, **comm**, **idem** and **unit** attributes specify the operation $\_ \cup \_$ to be associative, commutative,

```
spec NonDeterministicAutomata =
  sort In
  sort State
  then free {
      type FinSet ::= {} | {__}(State) | __ ∪ __(FinSet; FinSet)
      op __ ∪ __ : FinSet × FinSet → FinSet,
              assoc, comm, idem, unit {}  }
  then cotype State ::= (next : In → FinSet)
end
```

Fig. 9. Specification of non-deterministic automata.

idempotent and have unit {}.

The **cofree { ... }** construct dualizes the **free { ... }** construct by restricting the model class of a specification to the cofree, i.e. final ones. This generalizes the **cofree cotypes** construct to arbitrary specifications; in particular, final models may be restricted by axioms (e.g. as in Figure 11 below).

More precisely, the semantics of **cofree** is defined as follows:

Definition 9 If Sp_1 is a specification with signature Σ_1, then the models of Sp_1 **then cofree** $\{Sp_2\}$ are those models M of Sp_1 **then** Sp_2 that are *fibre-final* over $M|_{\Sigma_1}$ w.r.t. the reduct functor $\_|_{\Sigma_1}$. Here, fibre-finality means that M is the final object in the fibre over $M|_{\Sigma_1}$. The fibre over $M|_{\Sigma_1}$ is the full subcategory of $Mod(Sp_1$ **then** $Sp_2)$ consisting of those models whose Σ_1-reduct is $M|_{\Sigma_1}$.

This definition deviates somewhat from the semantics of **free** in that the latter postulates initiality, with the unit required to be the identity map, rather than fibre-initiality. (Actually, it might be worthwhile to redefine the CASL semantics for free specifications in terms of fibre-initial models.) We will see shortly that the more liberal semantics for **cofree** is essential in cases where sorts from the local environment occur as argument sorts of selectors. Call a sort from the local environment an *output sort* if it occurs only as a result type of selectors. In the cases of interest, a more general co-universal property concerning, in the notation of the above definition, morphisms of Σ_1-models that are the identity on all sorts except possibly the output sorts follows from fibre-finality.

The **cofree cotypes** construct is equivalent to **cofree { cotypes ... }**:

Proposition 10 *If DD is a sequence of selector-based cotype definitions without subsorting, then*

spec FINALMOORE2 =
  sorts *In*, *Out*
  then **cofree {**
  cotype *State* ::= (*next* : *In* → *State*; *observe* : *Out*)
  }
end

Fig. 10. Structured cofree specification of the final Moore automaton.

cofree { cotypes *DD* **}** *and* **cofree cotypes** *DD*

have the same semantics.

PROOF. Thanks to the fact that the semantics of the **cofree** construct is defined via *fibre*-finality, the interpretations of additional parameters for observers are fixed (in a given fibre). Hence, we can apply currying as in Def. 1. The result then follows from Props. 2 and 8. □

By contrast, the use of **cofree { types . . . }** should be avoided:

Example 11 The specification

free type *Bool* ::= *false* | *true*
then
cofree {type *T* ::= *c1*(*s1* :?*Bool*) | *c2*(*s2* :?*Bool*) **}**,

is inconsistent. Indeed, by applying the uniqueness part of finality to a model of the unrestricted type where T has an element on which both selectors are undefined (this is allowed for types but not for cotypes), one obtains that any model of the cofree type would be a singleton; however, singleton models fail to satisfy the finality property e.g. for the model of the unrestricted type where T is $Bool \times Bool$ and the selectors are the projections.

As an example for the significance of the relaxation of the cofreeness condition, consider the specification of Moore automata as given in Figure 10. Here, the observer *next* depends not only on the state, but additionally on an input letter.

In the standard theory of coalgebra, *next* would become a higher-order operation $next : State \to State^{In}$, and the cofree coalgebra indeed yields the final automaton showing all possible behaviours - but only for a *fixed* carrier for *In* (the inputs). The carrier for *Out* is also regarded as fixed; however, one can show that the co-universal property holds also for morphisms that act

```
spec BitStream3 =
  free type Bit ::= 0 | 1
  then cofree {
      cotype BitStream ::= (hd : Bit; tl : BitStream)
      ∀s : BitStream
      • hd(s) = 0 ∧ hd(tl(s)) = 0 ⇒ hd(tl(tl(s))) = 1 }
end
```

Fig. 11. Structured cofree specification of bit streams in CoCasl.

```
spec ListStreamTree [sort Elem] =
  free type
      Tree ::= EmtpyTree
             | Tree(left :? Tree; elem :? Elem; : Elem; right :? Tree)
  cofree cotype
      Stream ::= (hd : Tree; tl : Stream)
  free type
      List ::= Nil | Cons(head :? Stream; tail :? List)
end
```

Fig. 12. Nested free and cofree (co)types.

non-trivially on *Out*. If the semantics of **cofree** required actual cofreeness, i.e. a couniversal property also for morphisms that act non-trivially on *In*, the specification would be inconsistent!

Let us now come to a further modification of the stream example. If the axiom were omitted in the specification in Figure 11, the model class would be the same as that in Figure 3, instantiated to the case of bits as elements. *With* the axiom, the streams are restricted to those where two 0's are always followed by a 1. Again, this is unique up to isomorphism.

It is straightforward to specify iterated free/cofree constructions, similarly as in [35]. Consider e.g. the specification of lists of streams of trees in Figure 12. Alternatively, one could have used structured free and cofree constructs as well:

$$SP \textbf{ then free } \{SP_1\} \textbf{ then cofree } \{SP_2\} \textbf{ then free } \ldots$$

Note that also in the latter case, there won't be any **free** *within* a **cofree** or vice versa. An example for **free** *within* **cofree** is shown in Figure 13. Here, the inner **free** has to be a structured one, since finite sets cannot be specified as **free type** directly.

```
spec FinalNonDeterministicAutomaton =
  sort In
  then cofree {
      sort State
      then free {
          type FinSet ::= {} | {__}(State) | __ ∪ __(FinSet; FinSet)
          op __ ∪ __ : FinSet × FinSet → FinSet,
                  assoc, comm, idem, unit {}  }
      then cotype State ::= (next : In → FinSet)  }
end
```

Fig. 13. A free type *within* a cofree type.

7 Modal logic

We now define a multi-sorted modal logic for use with process types, the basic idea being that observer operations give rise to modalities that describe the evolution of the system upon application of the observer. Related work, to be discussed at the end of Section 8, includes [17,18,21,41]. The underlying intuition is that the non-observable sorts of a process type form a multi-sorted state space, and that observers either directly produce observable values or effect an evolution of the state. Now the idea of modal logic in general is to formulate statements about such systems without explicit reference to the states.

In CoCasl, this takes the following shape. We are defining modal formulas for a given **cotype** declaration. All the sorts defined in the cotype are called *non-observable*, and the selectors are called *observers*. Sorts from the local environment are called *observable*. These notions can also be reformulated in terms of a signature of the modal CoCasl institution, see Section 9.

Atomic formulas in the modal logic involve *observer terms*. These are built from unary observers with *observable* result sort (which are treated as flexible constants, i.e. constants that depend on the respective state), observers with additional parameters (which then need to be applied to sufficiently many observer terms) and variables and function symbols from the local environment. An observer t (possibly applied to extra parameters) with *non-observable* result sort leads to modalities $[t]$, $<t>$, $[t*]$, $<t*>$ (all-next, some-next, always, eventually). The modal logic then consists of such equations between observer terms as atomic sentences, which may be combined using the modalities above and the usual propositional connectives, as well as quantification over variables of observable sorts. Using this logic, we can write, in the example of Figure 11,

$$hd = 0 \ \wedge \ [tl]hd = 0 \ \Rightarrow \ [tl][tl]hd = 1$$

as syntactic sugar for

$$hd(s) = 0 \wedge hd(tl(s)) = 0 \Rightarrow hd(tl(tl(s))) = 1$$

More precisely, we define the meaning of a modal formula φ to be

$$\forall x : s [\![\varphi]\!]_{x:s},$$

where $[\![\_]\!]_\_$ is defined as follows:

- $[\![u]\!]_{t:s} \equiv u$, if u is an observer term consisting of variables and operation symbols from the local environement,
- $[\![f]\!]_{t:s} \equiv f(t)$ if $f : s \to s'$ is a unary observer with observable result,
- $[\![f(t_1, \ldots, t_n)]\!]_{t:s} \equiv f([\![t_1]\!]_{t:s}, \ldots, [\![t_n]\!]_{t:s}, t)$, if $f : s_1 \times \cdots \times s_n \times s \to s'$ is an observer with additional parameters and observable result, and t_i is an observer term of sort s_i $(i = 1, \ldots, n)$,
- $[\![u_1 = u_2]\!]_{t:s} \equiv [\![u_1]\!]_{t:s} = [\![u_2]\!]_{t:s}$,
- $[\![u_1 \stackrel{e}{=} u_2]\!]_{t:s} \equiv [\![u_1]\!]_{t:s} \stackrel{e}{=} [\![u_2]\!]_{t:s}$,
- $[\![def\ u]\!]_{t:s} \equiv def [\![u]\!]_{t:s}$,
- $[\![[f]\varphi]\!]_{t:s} \equiv def\ f(t) \Rightarrow [\![\varphi]\!]_{f(t):s'}$, if $f : s \to s'$ is a unary observer with non-observable result,
- $[\![[f(t_1, \ldots, t_n)]\varphi]\!]_{t:s} \equiv def\ f([\![t_1]\!]_{t:s}, \ldots, [\![t_n]\!]_{t:s}, t) \Rightarrow [\![\varphi]\!]_{f([\![t_1]\!]_{t:s}, \ldots, [\![t_n]\!]_{t:s}, t):s'}$, if $f : s_1 \times \cdots \times s_n \times s \to s'$ is an observer with additional parameters and non-observable result and t_i is an observer term of sort s_i $(i = 1, \ldots, n)$,
- $[\![[f]\varphi]\!]_{t:s} \equiv \forall x_1 : s_1, \ldots, x_n : s_n \,.\, def\ f(x_1, \ldots, x_n, t) \Rightarrow [\![\varphi]\!]_{f(x_1, \ldots, x_n, t):s'}$, if $f : s_1 \times \cdots \times s_n \times s \to s'$ is an observer with additional parameters and non-observable result.

The translation is extended to the logical connectives and quantifiers by structural rules which just copy these.

Note that each modal formula has a *sort*, which is the sort occuring in the subscript argument of the translation function. In particular, a modal formula is well-formed and the translation function $[\![\_]\!]_\_$ is defined only in case of correct sorting. One may switch to a different sort (i.e. a different state space) using the modalities, but only in a well-sorted way. If necessary (due to overloading), observers have to be provided with explicit types.

We extend the modal formulas to *generalized modal formulas* by additionally allowing predicates on the states:

$$[\![p(t_1, \ldots, t_n)]\!]_{t:s} \equiv p([\![t_1]\!]_{t:s}, \ldots, [\![t_n]\!]_{t:s}, t),$$

if $p : s_1 \times \cdots \times s_n \times s$ is a predicate having exactly one non-observable sort (i.e. not coming from the local environment) s, and t_i is an observer term of sort s_i $(i = 1, \ldots, n)$. Note that unlike equations with terms of observable sort,

predicates are only preserved, but *not* reflected by homomorphisms; hence the different name for formulas containing predicates.

The other modalities now can be defined as derived notions, where the starred forms $[t*]$, $<t*>$, being inspired by dynamic logic, need infinitary formulas. We here only treat the case of unary observers, the other cases being entirely analogous:

- $[\![<f>\ \varphi]\!]_{t:s} \equiv \neg[\![[f]\neg\varphi]\!]_{t:s}$
- $[\![[f*]\varphi]\!]_{t:s} \equiv [\![\varphi \wedge [f]\varphi \wedge [f][f]\varphi \wedge [f][f][f]\varphi \wedge \ldots]\!]$
  (here, argument and result sort of f must coincide)
- $[\![<f*>\ \varphi]\!]_{t:s} \equiv \neg[\![[f*]\neg\varphi]\!]_{t:s}$

Alternatively, one can define the starred modalities via **free** and **cofree**. E.g. $[f*]\varphi(x : s')$ can be replaced with $p(x)$ if the latter is defined to be the greatest predicate that implies φ and is closed under $[f]$. This can be expressed via

cofree {
  pred $p\ :\ s' \times s$
  - $\forall x : s' .\ p(x) \Rightarrow \varphi \wedge [f]p(x)$

}

Moreover, we provide a *global diamond* $\langle\mathsf{global}\rangle$ as suggested in [22], where $\langle\mathsf{global}\rangle\phi$ expresses the fact that ϕ holds in some state of the system, i.e.

$$\exists x : s [\![\varphi]\!]_{x:s},$$

For reasons laid out in [22], the global diamond is restricted to positive occurrences (see also the proof of Theorem 24 below). As explained in [22], the global diamond is, in terms of expressivity, equivalent to *modal rules* which state implications between *validities* of modal formulas.

The modal logic introduced above allows expressing safety or fairness properties. For example, the model of the specification BitStream4 of Figure 14 consists, up to isomorphism, of those bitstreams that will always eventually output a 1. Here, the 'always' stems from the fact that the modal formula is, on the outside, implicitly quantified over all states, i.e. over all elements of type *BitStream*.

Remark 12 The modal μ-calculus [20], which provides a syntax for least and greatest fixed points of recursive modal predicate definitions, is expressible using free and cofree specifications: μ is expressible by free recursively defined predicates, while ν is expressible by cofree recursively defined predicates. We have refrained from including syntactical sugar for the μ-calculus in CoCasl, because this would involve higher order variables and hence appear to be against the grain of CoCasl, which is first-order in spirit (although higher-order types can be emulated).

```
spec BitStream4 =
  free type Bit ::= 0 | 1
  then cofree {
      cotype  BitStream ::= (hd : Bit; tl : BitStream)
      • <tl*> hd = 1
      }
end
```

Fig. 14. Specification of a fairness property.

8 Modalities for structured observations

The limitation of the simple modal operators introduced in the previous section is that they are defined only for observers whose result sort is a non-observable sort, such as $tl : Stream \to Stream$. We now extend the concept to also cover observers into datatypes over the non-observable sorts, the leading example being the observer $next : In \times State \to FinSet$ from Figure 9.

In this example, the difference between the associated box and diamond operators becomes much clearer than before: $[next(i)]\phi$ will be intended to hold in a state s if ϕ holds for all successor states of s on input i, i.e. for all elements of $next(i, s)$, while $<next(i)> \phi$ will express that there *exists* a successor state that satisfies ϕ. Both predicates on sets are easily defined by recursion. We shall presently identify a class of datatypes that admits an analogous interpretation of modal operators.

We begin with a heuristic example showing that not all datatypes are suitable for modal logic:

Example 13 Let T denote the free abelian group monad; i.e. TX is the set of maps $X \to \mathbb{Z}$ with finite support (alternatively, think of the elements of TX as multisets with possibly negative multiplicities), with componentwise zero and addition. While maybe not immediately appealing, this datatype is specifiable in CASL and certainly within the scope of standard coalgebra. Now assume that we have a non-observable sort S and an observer $next : S \to TS$. Even without a precise definition of what the modal operator $[next]$ would mean (such a definition is possible; cf. [46]), it is intuitively clear that $[next]false$ should hold in a state $s : S$ iff $next(s) = 0$. Now let us assume that, in a state s in some coalgebra A, $next(s) = u - v$, where the states $u, v : S$ are distinct but bisimilar, i.e. identified by some coalgebra morphism h. Then s satisfies $\neg[next]false$, but $h(s)$ does not. Thus, the modal operator $[next]$ is at variance with the underlying premise that modal logic describes the observable behaviour of a state, which should be invariant under coalgebra morphisms. Moreover, we cannot expect that classes of coalgebras defined by

modal formulas containing $[next]$ have final models.

Thus, we need some sort of restriction on the way datatypes are defined in order to guarantee invariance of modal formulas under coalgebra morphisms. The obvious flaw of the abelian groups monad exposed in the above example was that variables in a term can cancel each other out, so that there is no immediately obvious well-defined notion of what variables are contained in a term. A sufficient condition for well-behavedness of modal operators is obtained by exluding precisely this phenomenon:

Definition 14 A functor $T : \mathbf{Set}^n \to \mathbf{Set}$ is called a *non-cancellative datatype* if it is given in terms of parameter sorts, (total) constructor operations, possibly involving mutual recursion with other datatypes, and equations between constructor terms, subject to the *variable restriction* stating that in each equation, the two sides have the same free variables.

It is easy to show that *any* equation holding between constructor terms in a non-cancellative datatype satisfies the variable restriction.

Example 15 All absolutely free datatypes such as lists, trees, option types etc. are non-cancellative; so are finite sets and multisets ('bags'). E.g., finite subsets of X are built from the elements of X by means of three constructors, namely empty set, singleton, and union, with equations stating associativity, commutativity, and idempotence of the union operator and neutrality of the empty set; all these equations satisfy the variable restriction. The datatype of finitely branching trees is a non-cancellative datatype whose definition requires mutual recursion with a second (also non-cancellative) datatype of 'forests' (i.e. lists of trees). A typical example of a cancellative datatype is the abovementioned free abelian group monad, whose equation $x - x = 0$ violates the variable restriction.

Theorem and Definition 16 *Let $T : \mathbf{Set}^n \to \mathbf{Set}$ be a non-cancellative datatype, let $\bar{X} = (X_i)$ be a family of n sets, let $t \in T\bar{X}$, and let ϕ be a predicate on X_i, read as a function from X into a type Bool of boolean truth values. Let $\bar{T}_{\bar{X},i}$ denote the functor $\mathbf{Set} \to \mathbf{Set}$ obtained from T by fixing all arguments except the i-th argument to $\bar{X}$. Then the following are equivalent:*

(i) *t can be expressed by a constructor term α such that every element of X_i contained in α satisfies ϕ;*
(ii) *for every representation of t by a constructor term α, every element of X_i contained in α satisfies ϕ;*
(iii) $(\bar{T}_{\bar{X},i}\phi)\, t = (\bar{T}_{\bar{X},i}\top)\, t$
(iv) $(\bar{T}_{\bar{X},i}\langle id, \phi\rangle)\, t = (\bar{T}_{\bar{X},i}\langle id, \top\rangle)\, t$

(where $\top$ denotes the constant true predicate, and $\langle f, g\rangle$ denotes the function mapping x to $(f(x), g(x))$). These equivalent conditions are expressed by the

notation $[t]_G\,\phi$ *(w.r.t.* T *and* i*).*

PROOF. *(i)* $\Longleftrightarrow$ *(ii):* By the variable restriction, all representations of t by constructor terms use the same set of elements of X_i.

(ii) $\Longrightarrow$ *(iv):* Just note that, for any predicate ψ, $(\bar{T}_{\bar{X},i}\langle id, \psi\rangle)\,t$ is obtained by replacing every element x of X_i in a representation of t by $(x, \psi(x))$.

(iv) $\Longrightarrow$ *(iii):* Compose both sides of (iv) with $\bar{T}_{\bar{X},j}\pi_2$, where π_2 denotes the projection $X_i \times \mathit{Bool} \to \mathit{Bool}$.

(iii) $\Longrightarrow$ *(i):* Let x occur in a representation α of t. Then $\phi(x)$ occurs in a representation of the right side of (iii), hence by the variable restriction also in the representation obtained by substituting each element of X_i occuring in α by $\top$. Thus, $\phi(x) = \top$.

The theorem shows in particular that the interpretation of $[t]_G\,\phi$ is independent of the representation of T by constructors and equations.

Example 17 In typical datatypes, the above definition explicates as follows.

- If T is the finite power set functor, then $[t]_G\,\phi$ holds iff all elements of t satisfy ϕ.
- If T is the list functor, then $[t]_G\,\phi$ holds iff all entries of t satisfy ϕ.
- If T is the functor 'multiplication with 2', i.e. $TX = X + X$, then $[t]_G\,\phi$ holds iff $\delta_X(t)$ satisfies ϕ, where δ_X is the codiagonal $[id, id] : X + X \to X$.

Remark 18 The concept of non-cancellative datatype relates closely to notions employed in [45,46] in the context of computational logic for monadic functional programming. In particular, Condition (iv) in Theorem 16 shows that, in the notation of [46], $[t]_G\,\phi$ is equivalent to $[x \leftarrow t]_G\,\phi(x)$, and the equivalence of Conditions (iii) and (iv) specializes, in the terminology of [45] and for the case of monads, to the statement that non-cancellative monads are simple. This explains also the use of the index G for 'global', indicating that, unlike in [45,46], we do not inquire further into a notion of state that might come with the datatype T itself, the idea underlying this work being that the states are explicit in the shape of the non-observable sorts.

Given these definitions, the need arises for CoCasl language constructs that deal with non-cancellative datatypes. Both in order to enforce the variable restriction and to identify datatypes to be equipped with modal operators, we therefore introduce a new semantic annotation %**modal** for free specifications. Explicitly, the annotation

Sp₁ **then free** %**modal** *Sp₂*

is correct iff

- Sp_2 is a basic specification consisting only of a type declaration (possibly declaring several mutually recursive types) and equational axioms;
- the types declared in Sp_2 are fresh, i.e. not already declared in Sp_1;
- the equations are only between terms of the newly declared types and have the same free variables on both sides;
- the type declaration contains selectors only if there are no equations.

(Note that generally, the use of selectors in type declarations other than absolutely free types tends to produce specification errors.) Datatypes declared by a %**modal** free extension are called *derived*; every derived datatype gives rise to a non-cancellative datatype $T : \mathbf{Set}^n \to \mathbf{Set}$, where the n arguments correspond to the sorts declared in Sp_1 and used as argument sorts for constructors in Sp_2; the latter are called the *type parameters* of the datatype.

We can now make the extended syntax of modal logic explicit: If $f : R_1 \times \ldots R_n \times S \to W$ is an observer with parameter sorts R_i, where W is a derived datatype, then f gives rise to a family of modal operators $[f(r_1, \ldots, r_n)]$, indexed over the elements of the parameter sorts. Such modal operators are called *structured modal operators*, to be distinguished from the *simple modal operators* introduced in the previous section. A modal formula of the form $[f(r_1, \ldots, r_n)]\phi$ has type S. The type W is, for purposes of the modal logic, firmly connected with the %**modal** free extension that defines it; in particular, defining the same type in two different such extensions is expressly forbidden. Thus, the modal operator $[f(r_1, \ldots, r_n)]$ is unambiguously associated with a non-cancellative datatype $T : \mathbf{Set}^n \to \mathbf{Set}$ to be used in its interpretation. Now let ϕ be a modal formula of type V, where V is the i-th type parameter of T. Then the semantics of $[f(r_1, \ldots, r_n)]\phi$ is given as follows:

Definition 19 Let s be of type S. Then

$$s \models [f(r_1, \ldots, r_n)]\phi \quad \text{iff} \quad [f(r_1, \ldots, r_n, s)]_G\, \phi,$$

where the right hand formula is to be read according to Theorem and Definition 16 (w.r.t. T and i).

This definition is easily seen to be compatible with the semantics for simple modal operators given in Section 7, so that we can regard simple modal operators as a special case of structured modal operators. The diamond modality $<f(r_1, \ldots, r_n)>$ is defined as $\neg[f(r_1, \ldots, r_n)]\neg$. The definition of iterated modalities $[f(r_1, \ldots, r_n)*]$, as well as implicit quantification by omission of parameters, carries over directly from Section 7.

spec NonRepetitiveNonDeterministicAutomata =
  sort *In*
  sort *State*
  then free %**modal{**
    type $FinSet ::= \{\} \mid \{\_\_\}(State) \mid \_\_ \cup \_\_(FinSet; FinSet)$
    op $\_\_ \cup \_\_ : FinSet \times FinSet \to FinSet$,
      assoc, **comm**, **idem**, **unit** $\{\}$ **}**
  then cotype $State ::= (next : In \to FinSet)$
  • $\forall i : In . \ <next(i)*> \ [next(i)] \ false$
end

Fig. 15. Specification of non-repetitive non-deterministic automata using modalities for structured observations.

Example 20 For the case where there is only one type parameter, the semantics of $[f]\phi$ is illustrated in Example 17 — just note that, in the notation used there, the element $t \in TX$ is now parametrized by a state s. As a simple example with several type parameters, consider relations, i.e. sets of pairs:

sorts S, V
then free %**modal {**
  type $Rel ::= \{\} \mid ins(S, V, Rel)$
  forall $s1, s2 : S; t1, t2 : V; r : Rel$
  • $ins(s1, t1, ins(s2, t2, r)) = ins(s2, t2, ins(s1, t1, r))$
  • $ins(s1, t1, ins(s1, t1, r)) = ins(s1, t1, r)$
}

Suppose that we have observers

ops $next : In \times S \to Rel$
  $out : V \to Nat$

where *In* is a parameter sort of inputs. Then the formula $out = 0$ has type V. Hence, the formula $[next(i)]out = 0$, of type S, holds in $s : S$ iff $t1$ satisfies $out = 0$ for each element $(s1, t1)$ of $next(i, s)$. Contrastingly, the formula $[next(j)][next(i)]out = 0$ holds in $s : S$ iff $s1$ satisfies $[next(i)]out = 0$ for each element $(s1, t1)$ of $next(j, s)$.

Example 21 Consider the specification of non-repetitive non-deterministic automata in Figure 15. Here, we express that no input letter i may occur all the time, that is, when the letter i's is input non-stop ($<next(i)*>$), the automaton will eventually get stuck ($[next(i)]\, false$).

Remark 22 Like the simple modal logic of Section 7, the structured modal

operators can be regarded as syntactical sugar and thus do not add expressivity to CoCasl. The encoding is slightly more complicated than for simple modal operators; moreover, one has to introduce auxiliary operations which later have to be hidden — i.e. while the simple modal logic can be translated directly into first order logic, the structured operators require structured specifications for their translation. It should be stressed, however, that the required symbol hiding is comparatively harmless, since the hidden operations are not only definitional, but also do not impose extra conditions on model morphisms, so that hiding them induces an isomorphism of model categories. In fact, for precisely this reason, hiding the auxiliary symbols is really unnecessary from a theoretical perspective; it serves only syntactical convenience in that it avoids overburdening the signature.

Explicitly, the encoding works as follows. Let S be a non-observable sort with observer $f : R_1 \times \cdots \times R_n \times S \to W$, where W is a derived datatype. By Theorem 16, W has a well-defined elementhood predicate $\_\_eps\_\_ : U \times W$ for each type parameter U of W. This predicate can be specified by primitive recursion over the constructors (possibly by mutual recursion involving further datatypes needed to construct W). Note that this is only possible thanks to the fact that W is non-cancellative — a similar recursive definition e.g. for the free abelian group monad would fail to be well-defined and hence introduce an inconsistency! If ϕ is a modal formula of type U, then satisfaction of $[f(r_1, \ldots, r_n)]\phi$ in a state $s : S$ is encoded as

$$\forall u : U \bullet u \; eps \; f(r_1, \ldots, r_n, s) \implies \phi(u).$$

This formula is built recursively into the translation of the ambient modal logic formula. Finally, the elementhood predicate *eps* is hidden.

We conclude the section with the announced discussion of related work on modal logic for coalgebra (omitting the logic developed in the seminal paper [24], which is not immediately suitable for use in a specification language due to the presence of infinitary conjunction). The syntax chosen here is largely in the spirit of [17,21] in that modalities are indexed by observer terms. The syntax of [18], inherited from [40], differs in that it uses instead modal operators built along the structure of the signature functor, plus a single modality for the coalgebra structure. For the functors covered in [18], this choice does not affect expressivity at the level of state formulas. (The syntax of [18] allows formulating modal statements also at the level of the functor ingredients such as products and sums; however, the main interest is still in state formulas.) The syntax of the modal operators in CCSL, in turn, deviates from the others in that state variables are kept explicit; moreover, CCSL has an explicit bisimilarity relation (which can be emulated in CoCasl using cogeneratedness constraints). Among the pre-existing modal logics for coalgebra, [18] is unique (along with CoCasl, of course) in admitting several non-observable

sorts. Iterative modalities as in CoCasl are found only in CCSL.

The main novel feature of the modal logic introduced here is the generality of the datatypes admitted as observations. At this point, CCSL and the logic of [18] are incomparable in terms of generality: CCSL covers only absolutely free datatypes, while [18] admits, besides simple and parametrized observations, only (finite or infinite) power sets (under the heading of Kripke polynomial functors). Our notion of non-cancellative datatype includes both these cases.

9 An institution for modal CoCasl

We now sketch briefly how modal formulas are incorporated into an extended CoCasl institution (see [14] for a detailed definition for the notion of institution). We have shown that the modal logic can be regarded as syntactical sugar over the remaining language. However, for some purposes it will be desirable to retain the modal formulas explicitly, e.g. in order to pass them on to a modal theorem prover or in order to incorporate CoCasl into a heterogeneous framework such as the language of the heterogeneous tool set [25].

Since the syntax of modal logic relies to some extent on syntactical mechanisms that are not normally explicitly retained in the semantics of CoCasl specifications, such as in particular cotypes and free extensions, we need to extend the notion of signature in order to keep track of the required information. To wit, an *extended* CoCasl *signature* consists of a standard signature together with the following data:

- two transitive relations *constructs* and *sees* on the sorts;
- for each *derived datatype* S, i.e. each sort in the codomain of *constructs*, a set of operations with result sort S called its *constructors*.

A sort is called a *cotype* if it is in the domain of *sees*. Two sorts that see each other are called *siblings*. We say that S *strictly sees* T if S *sees* T but not conversely. Signature morphisms σ are required to preserve the *constructs* and *sees* relations and reflect the *constructs* relation and the strict *sees* relation; moreover, for a derived datatype S, the constructor set of $\sigma(S)$ must be the image of the constructor set of S.

The intuition for this structure is that a cotype S sees exactly the local environment at the time of its declaration, including all sorts declared in the same cotype declaration, which are, then, its siblings. All other sorts that it sees are regarded as observable for purposes of observing S; i.e. the *sees* relation gives rise to a *local* notion of observability. Moreover, the sorts that construct

a derived datatype are intended to be its type parameters.

The distinction between the *sees* relation and its strict subrelation is thus quite natural: the first names all possible outcome types for observers of a cotype, and the second names the observable ones among these. Note that the somewhat subtle relaxation of the reflection condition for the *sees* relation in the definition of signature morphisms enables us to instantiate observable parameter sorts with non-observable argument sorts in parametrized specifications, so that, e.g., a generic specification of lists over a non-observable sort of elements can be instantiated to obtain, say, lists of streams.

Such an extended signature gives rise to the following modal syntax: one has a flexible constant for each function symbol $f : R_1 \times \cdots \times R_n \times S \to T$, where S is a cotype that strictly sees T and the R_i. If, on the other hand, S strictly sees the R_i and T is a sibling of S or constructed by a sibling of S, then we obtain a family of modal operators for f, simple in the former case and structured in the latter case. The definition of signature morphisms is designed in such a way that such formulas can indeed be translated along them.

The semantics of flexible constants and simple modal operators is exactly as described in Section 7. The semantics of structured modal operators is determined by the constructors of the derived datatype T, using Condition (i) of Theorem 16 (this being the most sensible choice for datatypes that are not yet known to be non-cancellative) — i.e. $[f(r_1, \ldots, r_n)]\phi$, where ϕ has type U and U *constructs* T, holds in a state $s : S$ iff $f(r_1, \ldots, r_n, s)$ can be expressed by a term using only the constructors of T and elements of sorts that construct T, employing from U only elements that satisfy ϕ. In general, this semantics may produce pathologies (e.g. if T constructs itself, but also generally if T fails to be a non-cancellative datatype). However, the satisfaction condition holds due to the fact that signature morphisms do not remove or add constructors of derived datatypes, so that we have indeed arrived at an institution. Moreover, the mentioned pathologies do not occur in theories of actual CoCasl specifications, since the CoCasl syntax guarantees that derived datatypes are actually non-cancellative.

More precisely, a CoCasl specification generates an extended signature as follows. Every cotype declaration expands the *sees* relation as indicated above (so that redeclaring a cotype does increase the number of observable sorts from its vantage point). The declaration of a derived datatype by a **free** %**modal** specification has the effect that all sorts from the local environment of the free specification (i.e. from *outside* its scope) *construct* the declared datatypes, provided they actually appear as arguments in their constructors. (Thus, a derived datatype does not construct itself, and mutually recursive datatypes do not construct each other!) Moreover, the constructors of derived datatypes are explicitly recorded in the signature.

10 Existence of cofree models

We now turn to the problem of establishing a general format for structured cofree specifications that guarantees consistency; essentially, this amounts to asking which subcategories of the category $\mathbf{CoAlg}(\Sigma)$ of coalgebras for a given functor Σ have final coalgebras. For the dual case, the answer is given in [47]: free models exist for specifications with universally quantified Horn axioms. Part of a corresponding coalgebraic result has been obtained in [22]. In summary, it is known that

(i) Cofree coalgebras exist for bounded functors Σ on **Set**, more generally for accessible functors on locally presentable categories [6,31]. Here, a functor is called *(κ-)accessible* if it preserves κ-filtered (equivalently: κ-directed) colimits for some regular cardinal κ. The category $\mathbf{Set}^n$ is locally presentable.

(ii) Let Σ be a **Set**-valued functor that has a final coalgebra. Then every subcategory of $\mathbf{CoAlg}(\Sigma)$ defined by modal axioms or, more generally, axioms that are stable under coproducts and quotients, has a *fully abstract* final coalgebra, i.e. a final object that is a subobject of the final Σ-coalgebra [21,22].

The second statement has to generalized slightly in order to cope with specifications with several non-observable sorts, i.e. for coalgebras over $\mathbf{Set}^n$. Even more generally, we have

Proposition 23 *Let* $\mathbf{C}$ *be a category equipped with a factorization system* $(\mathbf{E}, \mathcal{M})$ *for sinks [3], and let* $\Sigma : \mathbf{C} \to \mathbf{C}$ *be a functor that preserves* $\mathcal{M}$*, i.e.* $\Sigma[\mathcal{M}] \subset \mathcal{M}$*. Then*

(i) $(\mathbf{E}, \mathcal{M})$ *lifts to a factorization structure*

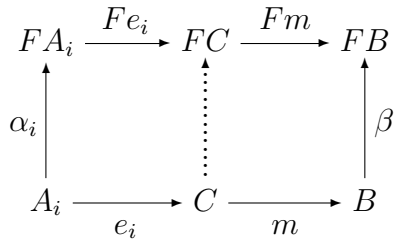
$$(U^{-1}[\mathbf{E}], U^{-1}[\mathcal{M}]),$$

also denoted $(\mathbf{E}, \mathcal{M})$*, on* $\mathbf{CoAlg}(\Sigma)$*, where* U *is the forgetful functor* $\mathbf{CoAlg}(\Sigma) \to \mathbf{C}$*.*

(ii) If $\mathbf{B}$ *is a full subcategory of* $\mathbf{CoAlg}(\Sigma)$ *that is closed under* $\mathbf{E}$*-sinks, and* Σ *has a final coalgebra, then* $\mathbf{B}$ *has a final coalgebra that is* fully abstract*, i.e. an* $\mathcal{M}$*-subobject of the final* Σ*-coalgebra.*

PROOF. *(i):* The diagonal property follows from the fact that $\mathbf{E}$-sinks are epi-sinks and hence final w.r.t. U. Now let $(g_i : (\alpha_i, A_i) \to (\beta, B))_I$ be a sink in $\mathbf{CoAlg}(\Sigma)$, with coalgebras $\alpha : A \to FA$ written in the form (α, A). The factorization of (g_i) is obtained by lifting the $(\mathbf{E}, \mathcal{M})$-factorization $(g_i) = m(e_i)$

in **C**, using the diagonal of the square $Fm(Fe_i\alpha_i) = (\beta m)e_i$:



(ii): The closure condition implies that **B** is $\mathcal{M}$-coreflective in $\mathbf{CoAlg}(\Sigma)$ [3]. The coreflection of the final Σ-coalgebra is final in **B**. □

For functors Σ on $\mathbf{Set}^n$, equipped with the componentwise factorization structure (jointly surjective, injective), the preservation condition is always *almost* satisfied, since injective maps in $\mathbf{Set}^n$ are sections and hence preserved by all functors, provided that all components of the domain are non-empty. For the case $n = 1$, it is shown in [6] (Proof of Theorem 3.2) that one can always modify Σ in such a way that it preserves injective maps and such that both its behaviour on non-empty sets and its category of coalgebras remain essentially unchanged. It is easy to check that the given construction works mutatis mutandis for arbitrary n; we shall thus silently assume that Σ preserves monomorphisms.

If, in the notation of Proposition 23, **C**, and hence $\mathbf{CoAlg}(\Sigma)$, is cocomplete, then closure under **E**-sinks is equivalent to closure under **E**-quotients and coproducts, provided that every **E**-sink contains a small **E**-sink (this is the case in $\mathbf{Set}^n$); this fact relates closely to the results of [22] for the **Set**-valued case. However, it is often just as easy to argue directly via **E**-sinks, as will be seen in the proof of our main existence result:

Theorem 24 *Let Sp be a specification of the form*

Sp₁ **then cofree** *Sp₂* .

Call the sorts from Sp_1 observable *sorts. Let the specification* Sp_2 *consist of (no more than)*

- *declarations of (new)* non-observable *sorts;*
- *(new) derived datatypes (cf. Section 8);*
- *a redeclaration of the non-observable sorts as cotypes, with only observable sorts as parameters: and*
- *modal logic formulas (excluding generalized modal formulas, but including the global diamond).*

Then Sp is conservative (model-extensive) over Sp_1, provided that Sp_1 **then** *Sp_2 is conservative over Sp_1.*

PROOF. Since the derived datatypes depend functorially on the non-observable sorts, one sees as in Proposition 2 that the non-observable sorts and their observers form a Σ-coalgebra for a functor $\Sigma : \mathbf{Set}^n \to \mathbf{Set}^n$, with n being the number of non-observable sorts. Thus, the category $\mathbf{B}$ of Sp_2-models over a given Sp_1-model is equivalent to a full subcategory of $\mathbf{CoAlg}(\Sigma)$. The functor Σ is κ-accessible, with κ being the largest cardinality of a parameter sort if there is an infinite parameter sort, and $\kappa = \omega$ otherwise; hence, Σ admits a final coalgebra (see above).

By Proposition 23, it now suffices to show that subcategories determined by modal logic formulas are closed under componentwise jointly surjective sinks. Thus, let ϕ be a modal formula of type S, a non-observable sort. We have to show that

$$s \models \phi \quad \text{implies} \quad h(s) \models \phi$$

for each coalgebra homomorphism h in $\mathbf{B}$ and each s in S.

To this end, we replace global diamonds in ϕ by explicit existential quantifiers, modal operators with omitted parameters by suitable (universal or existential) quantifications over observable sorts, and iterated modalities by infinitary conjunctions or disjunctions. We then transform ϕ into prenex normal form. (The latter is possible thanks to the fact that carrier sets are assumed to be nonempty. That quantifiers indeed commute with modal operators is seen by a simple case distinction over whether or not the inner quantified formula holds and does not require expanding the definition of the modal operator.) Denote the quantifier-free part of the resulting formula by $\bar{\phi}$. We will show that satisfaction of $\bar{\phi}$ is preserved by h (when applied to all free variables in $\bar{\phi}$). This implies $(*)$: all quantifiers in the prenex normal form are either over observable sorts (stemming from quantifiers already explicit in ϕ or from modal operators with omitted parameters) or over non-observable sorts (stemming from global diamonds). Since global diamonds are restricted to positive occurences, the latter quantifiers are necessarily existential. Quantifiers over observable sorts remain unaffected since the observable sorts are fixed under h (due to the fact that attention is restricted to fibres). Truth (but not falsity) of existential quantifiers over non-observable sorts is preserved along h since witnesses are preserved.

It remains to be shown that h preserves satisfaction of $\bar{\phi}$. We prove the stronger statement that h preserves and reflects satisfaction by induction over the formula structure, recalling that $\bar{\phi}$ consists of equations between terms of observable sort, boolean connectives, statements of the form $s \models \psi$ (arising from the explication of global diamonds), and (fully parametrized) modal operators.

Explicitly, this means that, if $\bar{\phi}$ is of type S and contains free variables v_i of non-observable sorts V_i, then for all $s : S$ and all $v_i : V_i$,

$$s \models \bar{\phi} \quad \text{iff} \quad h_S(s) \models \bar{\phi}[h_{V_i}(v_i)/v_i],$$

(free variables of observable sort remain unchanged under h), where h_U denotes the action of h on a sort U. The base case of the induction is straightforward: the values of observable terms, and hence the truth values of equations and definedness assertions involving such terms, are left unchanged under h. The induction steps for the boolean connectives and for statements of the form $s \models \psi$ are trivial.

The remaining case is that $\bar{\phi}$ is an application of a structured modal operator (this case subsumes the case of simple modal operators; cf. Section 8). Making the state variable explicit, we obtain that $\bar{\phi}$ is of the form $[t]_G\,\psi$, where ψ is a modal formula of type U with free variables v_i as above for which preservation and reflection are already established, and $t : W$ is a term formed from observers, operations from the local environment, free variables of observable type (the latter two stemming from parameters), and a single free variable $s : S$ of non-observable type, with W a derived type with type parameter U. We have to show

$$[t]_G\,\psi \quad \text{iff} \quad [t[h_S(s)/s]]_G\,\psi[h_{V_i}(v_i)/v_i]. \qquad (*)$$

Since h is a coalgebra homomorphism, we have $t[h_S(s)/s] = (Th)\,t$. Thus, the left side of $(*)$ holds iff all elements u of U occurring in the value of $t : W$ satisfy ψ, and the right side holds iff for all these elements u, $h(u)$ satisfies $\psi[h_{V_i}(v_i)/v_i]$. Hence, the equivalence of the two sides follows from the inductive assumption. □

Remark 25 The last proviso in the theorem is needed because the given modal formulas may be inconsistent in the sense that they are false for all states. In the full category of coalgebras over $\mathbf{Set}^n$, such formulas do of course have a model, namely the empty coalgebra (in a sense, this observation is dual to the fact that the equation $x = y$ equating two free variables is consistent because it is modeled by the singleton). However, in CASL and hence in CoCASL, carrier sets are explicitly required to be non-empty, so that the model class of, say, the modal formula *false* is indeed empty.

Remark 26 It should be emphasized that the above theorem, although worded specifically for CoCASL, really applies in a much wider context — namely, in any setting with coalgebras over an explicit signature formed as a polynomial combination of non-cancellative datatypes.

Remark 27 The redeclaration of the non-observable sorts as cotypes serves mainly to incorporate the axioms for cotypes ensuring that partial observers

can be combined into a single total observer into a sum type. In case there are only total observers, the cotype declaration can be replaced by declarations of the observers.

We conclude the section with a few warnings concerning cofree specifications that deviate from the form sanctioned by Theorem 24:

Example 28 Restricting non-observable sorts by equational axioms, rather than modal formulas, may lead to inconsistencies. An extreme example is

spec FINALELEMENT =
BOOL
then cofree {
  sort *Unit*
  forall $x, y : Unit \bullet x = y$
  op $el : Unit \to Bool$
}

— the specification in brackets has precisely two models (three if empty carriers are admitted), none of which is final.

Moreover, observe that the initiality constraint for derived datatypes is essential. E.g., the specification of final nondeterministic automata (Figure 13) becomes meaningless if the initiality constraint for the type of sets is omitted — the model it describes then has the singleton set as its state space, with a singleton 'power set' that equates all subsets. In other words, enough of a handle must be provided to actually prove distinctness of observations.

Theorem 24 can be read as supporting the nesting of certain free specifications within cofree specifications. Nesting *cofree* specifications within either free or cofree specifications is more risky, essentially due to the fact that final coalgebras may be rather large. E.g. one can specify the full powerset functor $\mathfrak{P}$ by a cofree specification (in fact even as a **cofree cotype**), as shown in Section 11.1. In a surrounding free or cofree specification, one could then specify the initial algebra or final coalgebra, respectively, for $\mathfrak{P}$ — an inconsistency due to Russell's paradox.

11 Modelling Process Algebra in CoCASL

As a comprehensive example, we now show how to model central concepts of process algebra in CoCASL. Among the various frameworks for the description and modelling of reactive systems, process algebra plays a prominent rôle. It has proven to be suitable at the level of requirement specification, at the level of design specifications, and also for formal refinement proofs [7]. Almost all of the underlying concepts of process algebra can be found in the languages CCS

[23] and CSP [16,39]: a type system on the communications; synchronous as well as asynchronous communication; operational semantics; and also various notions of process equivalence like strong and weak bisimulation, observation congruence, and trace equivalence. Thus, when proposing a new framework which aims at the specification of reactive systems in general, it is worthwile to study if these process algebras and their semantic concepts are covered.

11.1 Elements of process algebra syntax

Process algebras observe reactive systems by means of communications. While CSP requires the communcations just to be a set, CCS has a small type system, which we model using CASL subtyping.

Both process algebras involve higher order types constructed on top of their set of communications, namely *sets* for hiding symbols and as synchronization sets, and *functions* as well as (binary) *relations* for renamings. These type constructions are not available in CASL, but they can be modelled coalgebraically.

Based on communications and the above mentioned higher order types, the syntax of processes can be specified as a free datatype. This allows also for an inductive definition of substitution on processes, a construction necessary to describe the semantics of recursive processes.

11.1.1 Datatypes of communications

The language CSP is defined relative to an alphabet Σ of all communications. At the semantical level, this alphabet Σ is extended by an 'invisible action' τ and and a 'termination signal' $\surd$ (*tick*). This can be specified in CASL as

sort *Sigma*
free type *ExtSigma* ::= **sort** *Sigma* |*tau* |*tick*

The effect of the **free type** declaration is that each element of *ExtSigma* is either an element of *Sigma* or one of the two distinct new elements *tau* and *tick*.

CCS processes communicate *names*. Each name n has a *co-name* $\bar{n}$, where the function $bar : n \mapsto \bar{n}$ is involutive. Names and co-names together form the set of *labels*. Adding to this set the *silent action* τ results in the set of *actions*.

sort *Name* %% Names
free type *Label* ::= **sort** *Name* |*bar*(*Name*) %% Labels
free type *Act* ::= **sort** *Label* |*tau* %% Actions
op *bar* : *Label* → *Label*
∀ *a*:*Name* . *bar*(*bar*(*a*)) = *a*

Note that we have the subsort relations *Name* < *Label* < *Act*. The operation *bar* is introduced twice: as constructor from *Name* into *Label* and as function on *Label*.

11.1.2 Sets, relations, and function spaces: higher order via cofreeness

As mentioned above, process algebras need higher order types constructed on their respective alphabet of communications. In CASL, it is not possible to specify these types monomorphically, while COCASL captures them in terms of the structured cofree construct.

The syntax of CCS requires arbitrary sets of labels for restrictions. Since the powerset, being isomorphic to the set of boolean-valued maps, enjoys a couniversal property, we can easily specify it in COCASL: building upon a specification of a type *Bool* of booleans and the type *Label* as above,

cofree cotype *Set*[*Label*] ::= (_ *isIn* _ : *Label* → *Bool*)

specifies *Set*[*Label*] as the powerset of the set of labels (compare this to the specification of function types in Figure 6). Concerning COCASL syntax, note that *Set*[*Label*] is a so-called compound identifier, which can, for the purposes of this paper, be regarded as a sort name like any other (in instantiations of the parametrized syntax specification that assign particular label sets to the parameter *Label*, the part of the name in square brackets will be syntactically replaced by the name of the concrete label set). Corresponding comments hold for other uses of this mechanism further below, e.g. *Fun*[*Label*] or *Relation*[*Sigma*].

Similarly, one specifies the function spaces needed for relabelling. Since only bijections that commute with the 'bar' operation are admissible as CCS relabellings, the actual type of relabellings is defined as a subtype:

cofree cotype *Fun*[*Label*] ::= (*eval* : *Label* → *Label*)
then
sort *Relabelling* = { *f* : *Fun*[*Label*] .
∀ *l*:*Label* . *eval*(*bar*(*l*), *f*) = *bar*(*eval*(*l*, *f*))
∧ ∀ *l*, *k*:*Label* . (*eval*(*l*, *f*) = *eval*(*k*, *f*) ⇒ *k* = *l*
∧ ∀ *l*:*Label* . ∃ *k*:*Label* . *l* = *eval*(*k*, *f*)) }

free type
AgentExpression ::= **sort** *AgentVariable*
| **sort** *AgentConstant*
| *0* %% inactive agent
| _ _→_ _(*Act*; *AgentExpression*) %% Prefix
| _ _+_ _(*AgentExpression*; *AgentExpression*) %% Sum
| _ _||_ _(*AgentExpression*; *AgentExpression*) %% Parall.
| _ _−_ _(*AgentExpression*; *Set*[*Label*]) %% Restriction
| _ _<_ _>(*AgentExpression*; *Relabelling*) %% Relabelling
| *fix*(*AgentVariable*; *AgentExpression*) %% Recursion

Fig. 16. The CCS Syntax as a free type.

Note the combination of coalgebraic and algebraic modelling: while the type of functions is specified using a structured cofree, the properties of 'relabellings' are described by classical algebraic constructs.

Sets of communications are also needed for the hiding and generalized parallel operators of CSP. Finally, the relational renaming operator of CSP requires a type of binary relations on the communication alphabet Σ:

cofree cotype *Relation*[*Sigma*] ::= (*holds* : *Sigma* × *Sigma* → *Bool*)

11.1.3 Process syntax and substitution: inductive types

Using the higher order types introduced above, the respective syntaxes of CCS and CSP can be specified as free types, c.f. Figures 16 and 17. The freeness constraint on the type declarations means that the elements of the types are precisely the terms formed from the parameter sorts (e.g. in Figure 16 the sorts *AgentVariable*, *AgentConstant*, *Act*, *Set*[*Label*] and *Relabelling*) and the constructor operations.

In [23], Milner introduces CCS as a *class* of agent expressions. The crucial point is that the summation operator (non-deterministic choice) involves arbitrary index sets. This is beyond the scope of CASL and COCASL, as the specified models interpret sorts by carrier *sets*. Therefore, and also in order to capture bisimulation via a final object in a suitably chosen category, we restrict the language to finite nondeterminism — this is expressive enough to retain full computational power (cf. [23], p. 135).

Also CSP is sometimes defined as a class of processes. Again, it is an operator for nondeterminism which is beyond set theory, namely the internal choice operator over an arbitrary set of processes. The main purpose of this operator is of theoretical nature. It is required as part of the language to prove full

free type

$$
\begin{array}{lll}
\mathit{Process} ::= & \mathit{Skip} & \\
| & \mathit{Stop} & \\
| & \mathit{Omega} & \\
| & \textbf{sort}\ \mathit{ProcessVar} & \\
| & \_\_\rightarrow\_\_(\mathit{Sigma};\ \mathit{Process}) & \%\% \text{ Prefix} \\
| & \_\_\mathit{seq}\_\_(\mathit{Process};\ \mathit{Process}) & \%\% \text{ Sequential Composition} \\
| & \_\_[]\_\_(\mathit{Process};\ \mathit{Process}) & \%\% \text{ External Choice} \\
| & \_\_\ |\tilde{}|\_\_(\mathit{Process};\ \mathit{Process}) & \%\% \text{ Internal Choice} \\
| & \_\_-\_\_(\mathit{Process};\ \mathit{Set}[\mathit{Sigma}]) & \%\% \text{ Hiding} \\
| & \_\_[[\_\_]](\mathit{Process};\ \mathit{Relation}[\mathit{Sigma}]) & \%\% \text{ Relational Renaming} \\
| & \_\_[\_\_]\_\_(\mathit{Process};\ \mathit{Set}[\mathit{Sigma}];\ \mathit{Process}) & \%\% \text{ Generalized Parallel} \\
| & \mathit{mu}(\mathit{ProcessVar};\ \mathit{Process}) & \%\% \text{ Recursion}
\end{array}
$$

Fig. 17. The CSP Syntax as a free type.

op $\_\_\{\_\_/\_\_\}$:
$\mathit{AgentExpression} \times \mathit{AgentExpression} \times \mathit{AgentVariable} \rightarrow \mathit{AgentExpression}$
$\forall\ P{:}\mathit{AgentExpression};\ X{:}\mathit{AgentVariable}$
$\bullet\ \forall\ Y{:}\mathit{AgentVariable}\ .\ \ Y\ \{\ P\ /\ X\ \} = P\ \mathit{when}\ Y = X\ \mathit{else}\ Y$
$\bullet\ \forall\ C{:}\mathit{AgentConstant}\ .\ \ C\ \{\ P\ /\ X\ \} = C$
$\bullet\ \mathit{0}\ \{\ P\ /\ X\ \} = \mathit{0}$
$\bullet\ \forall\ a{:}\mathit{Act};\ E{:}\mathit{AgentExpression}\ .\ \ (a \rightarrow E)\ \{\ P\ /\ X\ \} = a \rightarrow E\ \{\ P\ /\ X\ \}$
$\bullet\ \forall\ E,\ F{:}\mathit{AgentExpression}\ .$
$\quad (E + F)\ \{\ P\ /\ X\ \} = E\ \{\ P\ /\ X\ \} + F\ \{\ P\ /\ X\ \}$
$\ldots$

Fig. 18. Inductive definition of substitution in CCS.

abstraction results for the various denotational CSP semantics.

While CCS uses environments that bind agent constants to agent expressions, the version of CSP in [39], which we specify here, is restricted to a core language without environments. The full language including e.g. the various CSP parallel operators can be recaptured as a definitional extension.

Thanks to the free type construct of the process syntax it is straightforward to introduce substitution operators, as carried out for the case of CCS in Figure 18.

11.2 Structural Operational Semantics

For both process algebras, their semantics as a transition system is defined by structural operational semantics. A node of the transition system is an *AgentExpression* or a *Process*, respectively. The transitions are defined to

free { **pred** $\_\_ - \_\_ \rightarrow \_\_ : AgentExpression \times Act \times AgentExpression$
%% (Act):
$\forall$ a:Act; E:$AgentExpression$
• $(a \rightarrow E) - a \rightarrow E$
%% (Sum1):
$\forall$ E, E', F:$AgentExpression$; a:Act
• $E - a \rightarrow E' \Rightarrow (E + F) - a \rightarrow E'$
...
%% (Rec):
$\forall$ X:$AgentVariable$; E, E':$AgentExpression$; a:Act
• $E\{fix(X, E)/X\} - a \rightarrow E' \Rightarrow fix(X, E) - a \rightarrow E'$ }

Fig. 19. Part of the CCS Semantics.

be the smallest relation satisfying a certain set of inference rules. This relation is modelled by a structured free specification, which has the effect that the introduced predicate, e.g. $pred\_\_ - \_\_ \rightarrow \_\_ : AgentExpression * Act * AgentExpression$, holds on a minimal subset. Figures 19 and 20 show (part of) the operational semantics of CCS and CSP, respectively. Note, that [39] introduces the CSP process *Omega* only in the context of the operational semantics. Its purpose is to deal properly with termination. Within the structured free construct of both CoCasl specifications, only positive Horn clauses appear, so that the specifications are consistent (note that due to the definition of *Act* as free type, axioms with premise $\neg a = tau$ can be replaced by two axioms with equational premise). Figure 19 includes the CCS inference rule for recursion, which makes use of the substitution operator described above. CSP models recursion in the same way. Note how the rules for external choice in CSP are formulated along the type system of CSP communications on the semantical level. It is interesting to observe the difference between CCS and CSP in the modelling of nondeterminism. While CCS directly procedes with an action, the CSP semantics uses an invisible action τ. This inference rule among other, similar ones, is the reason why it is necessary to carefully extract the transitions with observable actions from the specified transition system. The advantage of the — at first sight slightly complicated — transition system for CSP is that it can also be taken as the basis for working out the denotations of processes in the failures and failures/divergences semantics of CSP.

11.3 Process Equivalences

Milner introduces strong bisimulation, weak bisimulation, and observation congruence as notions of equivalence on CCS agent expressions, which we model in a uniform way. For CSP, we study trace equivalence and show that it is essentially of algebraic nature although there exists a characterization in

free { **pred** $\_\_ - \_\_ \rightarrow \_\_ : Process \times ExtSigma \times Process$
. . .
%% External Choice:
$\forall\ P,\ P',\ Q{:}Process$
$\bullet\ P - tau \rightarrow P' \Rightarrow (P [] Q) - tau \rightarrow (P' [] Q)$
$\forall\ P,\ Q,\ Q'{:}Process$
$\bullet\ Q - tau \rightarrow Q' \Rightarrow (P [] Q) - tau \rightarrow (P [] Q')$
$\forall\ a{:}ExtSigma;\ P,\ P',\ Q{:}Process$
$\bullet\ \neg\ a = tau \Rightarrow P - a \rightarrow P' \Rightarrow (P [] Q) - a \rightarrow P'$
$\forall\ a{:}ExtSigma;\ P,\ Q,\ Q'{:}Process$
$\bullet\ \neg\ a = tau \Rightarrow Q - a \rightarrow Q' \Rightarrow (P [] Q) - a \rightarrow Q'$
%% Internal Choice:
$\forall\ P,\ Q{:}Process$
$\bullet\ (P |\tilde{}| Q) - tau \rightarrow P$
$\forall\ P,\ Q{:}Process$
$\bullet\ (P |\tilde{}| Q) - tau \rightarrow Q$
. . . }

Fig. 20. Semantis of CSP External and Internal Choice.

terms of bisimulation.

11.3.1 Strong Bisimulation

Modelling strong bisimulation is straightforward. We build up a new transition system, which — as a starting point — is a nearly identical copy of the CCS operational semantics. The difference is that the sort *Process* is introduced as a generated type, i.e. at this point the equivalence relation on its elements is left open. By choosing the transition predicate as observer for the sort *Process* in the cogenerated construct, the processes are identified by bisimulation. Finally, this notion is carried over to the sort *AgentExpression* via a predicate $\_\_ \sim \_\_$.

generated type $Process ::= [[\_\_]](AgentExpression)$
pred $\_\_ - \_\_ \rightarrow \_\_ : Process \times Act \times Process$
$\forall\ E,\ E'{:}AgentExpression;\ a{:}Act$
$\bullet\ E - a \rightarrow E' \Leftrightarrow [[E]] - a \rightarrow [[E']]$

cogenerated { **sort** *Process*
pred $\_\_ - \_\_ \rightarrow \_\_ : Process \times Act \times Process$ }

pred $\_\_\sim\_\_ : AgentExpression \times AgentExpression$
$\forall\ E,\ F{:}AgentExpression$
$\bullet\ E \sim F \Leftrightarrow [[\ E\ ]] = [[\ F\ ]]$

The cogeneratedness constraint guarantees full abstractness via a coinduction

axiom, which in this case amounts to stating that strong bisimulation is equality, cf. [42,23]. Note that the existence of strong bisimulation is guaranteed by the results of [23]; hence, the above specification is consistent. Moreover, since strong bisimulation is even a congruence, it is also consistent to shift the operations of the process syntax from the level of agent expressions to the level of processes. Also note that there are other abstraction principles on processes, like weak bisimulation as discussed below, that fail to be congruences.

11.3.2 Weak Bisimulation

In the specification of weak bisimulation in our setting, we make use of the following characterization in terms of strong bisimulation, reformulating a result of [10]:

Theorem 29 (Weak vs. Strong Bisimulation) *Let $\mathcal{T}_i = (S_i, s_i, Act, \rightarrow_i)$ be transition systems over Act with state sets S_i, initial states $s_i \in S_i$ and transition relations $\rightarrow_i$, $i = 1, 2$. Then*

$$\mathcal{T}_1 \approx \mathcal{T}_2 \iff W(\mathcal{T}_1) \sim W(\mathcal{T}_2),$$

where $\approx$ denotes weak bisimulation, and $\sim$ stands for strong bisimulation.

The operator W maps a transition system $\mathcal{T} = (S, s, Act, \rightarrow)$ to a transition system $W(\mathcal{T}) = (S, s, Act', \rightarrow_w)$ with Act' consisting of empty or one element lists over Act, $r \stackrel{\hat{\alpha}}{\rightarrow}_w r' :\iff r \stackrel{\hat{\alpha}}{\Rightarrow} r'$, where $\hat{} : Act \rightarrow Act'$ with

$$\hat{\alpha} := \begin{cases} \alpha \ ; \alpha \neq \tau \\ \epsilon \ ; \alpha = \tau \end{cases} \quad , \textit{ and } \quad \stackrel{\hat{\alpha}}{\Rightarrow} := \begin{cases} (\stackrel{\tau}{\rightarrow})^* \stackrel{\alpha}{\rightarrow} (\stackrel{\tau}{\rightarrow})^* \ ; \alpha \neq \tau \\ (\stackrel{\tau}{\rightarrow})^* \ ; \alpha = \tau. \end{cases}$$

PROOF. To prove '$\Rightarrow$', we claim that any weak bisimulation relation R between the transition systems $\mathcal{T}_i, i = 1, 2$, is also a strong bisimulation between $W(\mathcal{T}_i)$, $i = 1, 2$. This follows from the fact that for any weak bisimulation R the following holds: if $(r, s) \in R$ and $r \Rightarrow r'$ for some r', then $s \Rightarrow s'$ and $(r', s') \in R$ for some s'. This establishes the proof together with the observation that any step $r \stackrel{\alpha}{\rightarrow}_w r'$ with $\alpha \neq \tau$ in a transition system $W(\mathcal{T})$ corresponds to a derivation $r(\stackrel{\tau}{\rightarrow})^* r_1 \stackrel{\alpha}{\rightarrow} r_2(\stackrel{\tau}{\rightarrow})^* r'$ in $\mathcal{T}$. The reverse implication '$\Leftarrow$' holds because the transition systems $W(\mathcal{T}_i)$ have essentially $\stackrel{\hat{\alpha}}{\Rightarrow}_i$ as their transition relations.

Thus, in order to model weak bisimulation, it is necessary to specify the operator W, i.e. the transition relation $\stackrel{\hat{\alpha}}{\Rightarrow}$, in CoCasl. The specification below shows how to iterate *tau*-transitions on processes of type *AgentExpression* in

terms of a predicate $\_\_ \longrightarrow \_\_$. Then, a new transition system is defined. The state set remains, but the transition relation is $\stackrel{\hat{\alpha}}{\Rightarrow}$, which has *Act'* as labels.

pred $\_\_ - \_\_ \rightarrow \_\_ : AgentExpression \times Nat \times AgentExpression$
$\forall$ *E*, *E1*, *E3*:*AgentExpression*; *n*:*Nat*
• $E - 0 \rightarrow E$
• $E1 - (n + 1) \rightarrow E3 \Leftrightarrow$
$\exists$ *E2*:*AgentExpression* . $E1 - n \rightarrow E2 \wedge E2 - tau \rightarrow E3$

pred $\_\_ \longrightarrow \_\_ : AgentExpression \times AgentExpression$
$\forall$ *E1*, *E2*:*AgentExpression* . $E1 \longrightarrow E2 \Leftrightarrow \exists$ *n*:*Nat* . $E1 - n \rightarrow E2$

generated type $WProcess ::= [[[\_\_]]](AgentExpression)$
free type $Act' ::=$ **sort** $Label \mid epsilon$
pred $\_\_ - \_\_ - > \_\_ : WProcess \times Act' \times WProcess$
$\forall$ *E*, *E'*:*AgentExpression*; *l*:*Label*
• $[[[E]]] - l \rightarrow [[[E']]] \Leftrightarrow$
$\exists$ *E1*, *E2*:*AgentExpression* . $E \longrightarrow E1 \wedge E1 - l \rightarrow E2 \wedge E2 \longrightarrow E'$
• $[[[E]]] - epsilon \rightarrow [[[E']]] \Leftrightarrow E \longrightarrow E'$

Having this available, we can apply Theorem 29, i.e. strong bisimulation is defined as equality on *WProcess* and transferred to the CCS *AgentExpression*:

cogenerated { **sort** *WProcess*
pred $\_\_ - \_\_ - > \_\_ : WProcess \times Act' \times WProcess$ }
pred $\_\_ \approx \_\_ : AgentExpression \times AgentExpression$
• $\forall$ *E*, *F*:*AgentExpression* • $E \approx F \Leftrightarrow [[[\ E\ ]]] = [[[\ F\ ]]]$

Note that — as in the case of strong bisimulation — we obtain a fully abstract model, despite the fact that weak bisimulation fails to be a congruence for CCS, c.f. Milner's counterexample: $b.\mathbf{0} \approx \tau.b.\mathbf{0}$, but $a.\mathbf{0}+b.\mathbf{0} \not\approx a.\mathbf{0}+\tau.b.\mathbf{0}$. In the end, this means that the semantical operator $[[[\_\_]]]$ fails to be a homomorphism w.r.t. the CCS operations, here +.

11.3.3 Observation Congruence

With the notion of weak bisimulation available, we can express Milner's definition of observation congruence in [23], p.153, directly in CoCasl. The crucial point of this definition is that it involves a new transition relation $\_\_ == \_\_ \Longrightarrow \_\_$, which also takes the *tau* action into accout:

pred $\_\_ == \_\_ \Longrightarrow \_\_ : AgentExpression \times Act \times AgentExpression$
$\forall$ *E*, *E'*:*AgentExpression*; *alpha*:*Act*

$\bullet\ E == alpha ==> E' \Leftrightarrow$
$\quad \exists\ E1, E2{:}AgentExpression\ .\ E \longrightarrow E1 \wedge E1 - alpha \rightarrow E2 \wedge E2 \longrightarrow E'$
pred $\_\_==\_\_ : AgentExpression \times AgentExpression$
$\forall\ P, Q{:}AgentExpression; alpha{:}Act$
$\bullet\ P == Q \Leftrightarrow (\exists\ P'{:}\ AgentExpression\ .\ P - alpha \rightarrow P' \Rightarrow$
$\qquad (\exists\ Q'{:}AgentExpression\ .\ Q == alpha ==> Q' \wedge P' \approx Q'))$
$\qquad \wedge\ (\exists\ Q'{:}AgentExpression\ .\ Q - alpha \rightarrow Q' \Rightarrow$
$\qquad (\exists\ P'{:}AgentExpression\ .\ P == alpha ==> P' \wedge P' \approx Q'))$

Although this construction does not involve a 'copy' of the CCS transition system, it is easy to define a new process type *ObservationProcess*, which has observation congruence as equality:

generated type $ObservationProcess ::= Obs(AgentExpression)$
$\bullet\ \forall\ E, F{:}AgentExpression \bullet Obs(E) = Obs(F) \Leftrightarrow E == F$

11.3.4 Trace Equivalence on CSP

Similar to the modelling of weak bisimulation, it is possible to express trace equivalence in terms of bisimulation. This indicates once more the fundamental nature of bisimulation and, consequently, of the CoCasl **cogenerated** construct for the theory of concurrency.

Theorem 30 (Trace Equivalence vs. Strong Bisimulation) *Let* $\mathcal{T}_i = (S_i, s_i, \Sigma, \rightarrow_i)$ *be transition systems over* Σ *with state sets* S_i*, initial states* $s_i \in S_i$ *and transition relations* $\rightarrow_i$*,* $i = 1, 2$*. Then*

$$\mathcal{T}_1 =_{trace} \mathcal{T}_2 \iff P(\mathcal{T}_1) \sim P(\mathcal{T}_2),$$

where $=_{trace}$ *denotes trace equivalence, and* $\sim$ *stands for strong bisimulation.*

The operator P *describes the usual powerset construction. It maps a transition system* $\mathcal{T} = (S, s, \Sigma, \rightarrow)$ *to a transition system* $P(\mathcal{T}) = (2^S \backslash \emptyset, \{s\}, \Sigma, \rightarrow_P)$*, where*

$$X \xrightarrow{\alpha}_P Y :\iff Y = \{r' \in S \mid \exists\, r \in X\,.\, r \xrightarrow{\alpha} r'\}.$$

for all $X, Y \in 2^S \backslash \emptyset$*.*

PROOF.

"$\Leftarrow$": Is a direct consequence of the facts that (i) bisimular transition systems are trace equivalent and (ii) that the above described powerset construction yields a trace equivalent transition system.

"$\Rightarrow$": Let $R \subseteq RS(S_1) \times RS(S_2)$ be the smallest set such that

(1) $(\{s_1\}, \{s_2\}) \in R$ and
(2) if $(X, Y) \in R$, $X \xrightarrow{\alpha}_P X'$ in $P(\mathcal{T}_1)$, $Y \xrightarrow{\alpha}_P Y'$ in $P(\mathcal{T}_2)$, then $(X', Y') \in R$.

We claim that R is a bisimulation.

Let $(X, Y) \in R$, let $X \xrightarrow{\alpha}_P X'$ be a transition in $P(\mathcal{T}_1)$. As $(X, Y) \in R$, there exists a trace $u \in \Sigma^*$ such that $\{s_1\} \xrightarrow{u}_P X$ is a derivation of u in $P(\mathcal{T}_1)$ and $\{s_2\} \xrightarrow{u}_P Y$ is a derivation of u in $P(\mathcal{T}_2)$. As $X \xrightarrow{\alpha}_P X'$, also $u\alpha$ is a trace of $P(\mathcal{T}_1)$. As $P(\mathcal{T}_1)$ is trace equivalent to $P(\mathcal{T}_2)$, $u\alpha$ is also a trace of $P(\mathcal{T}_2)$. In $P(\mathcal{T}_2)$ any derivation for the prefix u ends in Y because $P(\mathcal{T}_2)$ is deterministic. Therefore, there exists a state Y' such that $Y \xrightarrow{\alpha}_P Y'$. As $(X, Y) \in R$, by definition of R we also obtain $(X', Y') \in R$.

Again, a similar result can be found in [10].

In order to apply this theorem to CSP processes, we first have to provide a powerset construction:

cofree cotype *Powerset[Process]* ::= (*eps* : *Process* → *Boolean*)
then
op {_} : *Process* → *PowerSet[Process]*
∀ *P*, *Q*:*Process*
• P = Q ⇒ *eps*(*P*, {*Q*}) = *True*
• P ≠ Q ⇒ *eps*(*P*, {*Q*}) = *False*
sort *NonEmtpyPS[Process]*= { *PS* : *PowerSet[Process]* .
∃ *P*:*Process* . *eps*(*P*, *PS*) = *True*}

The next step is to define the transition relation according to the operator *P*. To this end, it is necessary to extract all 'true' steps from the CSP transition system. The reason is that the CSP operational semantics introduces certain *tau* steps in order to deal with the different forms of non-determinism. Having the extracted relation of all true observations in *Sigma* available, we can apply the powerset construction.

generated type *TraceProcess* ::= *tr*(*NonEmptyPS[Process]*)
pred _ − _ → _ : *TraceProcess* × *Sigma* × *TraceProcess*
∀ *X*, *Y*:*NonEmptyPS[Process]*; *a*:*Sigma*
• *tr*(*X*) − *a* → *tr*(*Y*) ⇔
(∀ *Q*:*Process* . *eps*(*Q*, *Y*) = *True* ⇔
∃ *P*:*Process* . *eps*(*P*, *X*) = *True* ∧ *P* − *a* → *Q*)

According to Theorem 30, trace equivalence can now be defined in terms of strong bisimulation. Here, the embedding operation {_} of processes into the powerset of processes relates the two transition systems.

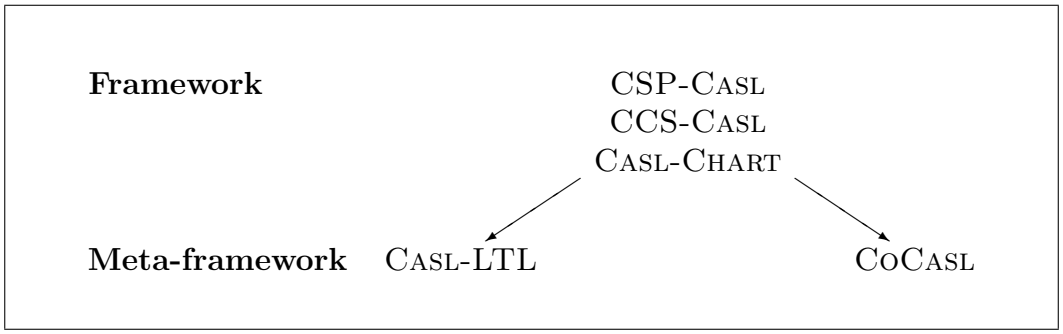


Fig. 21. Relationship between COCASL and other reactive CASL extensions

cogenerated
{ **sort** *TraceProcess*
pred $\_\_ - \_\_ \rightarrow \_\_ : \mathit{TraceProcess} \times \mathit{Sigma} \times \mathit{TraceProcess}$ }
pred $\_\_ =_{trace} \_\_ : \mathit{Process} \times \mathit{Process}$
$\forall\ P, Q: \mathit{Process}\ .\ P =_{trace} Q \Leftrightarrow tr(\{\ E\ \}) = tr(\{\ F\ \})$

Note that trace equivalence it not only of coalgebraic nature. The extraction of CSP process traces, as described by [39], can also be formulated in COCASL directly. This extraction uses essentially algebraic constructs. Thus, COCASL provides a framework which captures both approaches and — via the concept of a **view** — even allows relating them.

11.4 Related Approaches in Modelling Process Algebra

We have presented a general scheme for specifying models of concurrency: a clear destinction between syntax, operational semantics, and a (fully abstract) domain representing the chosen notion of equivalence has turned out to be the most adequate design.

There are various proposals of reactive CASL extensions – see Figure 21 for a small selection. Our definition of COCASL differs from CASL extensions like CSP-CASL [38], CCS-CASL [43,44] or CASL-CHART [33] These CASL extensions combine CASL with reactive systems of a particular kind, the semantics of which is defined in terms of set theory. We use COCASL (being much simpler than full set theory) as a meta-framework suitable for the formalization of (the semantics of) different frameworks for reactive systems. Hence, the proof support presented here can be used to prove meta-properties about these frameworks.

CASL-LTL [32] is similar to COCASL inasmuch as it is suitable as a meta-framework: for example, CCS has been formalized in CASL-LTL. However, the formalization in [32] has important drawbacks: only the transition relation is modelled, but the various forms of bisimulation are not covered, nor are

infinite state systems and recursion. It is unclear whether these shortcomings can be repaired in CASL-LTL.

12 Conclusion and related work

We have introduced CoCASL as a light-weight but expressive extension of CASL. CoCASL allows algebraic and coalgebraic specification to be mixed. CoCASL has a multi-sorted modal logic for reasoning with implicit states, partly modeled on predecessors from the literature but equipped with the crucial new feature of modal operators for structured observations in non-cancellative datatypes such as finite sets or lists of states. We have given a sufficient criterion for the existence of cofree models for specifications using non-cancellative initial datatypes and modal formulas. Moreover, we have shown how the modal logic can, alternatively to its encoding in modality-free CoCASL, be cast in the framework of an institution that incorporates a local notion of observability.

As an application, we have presented CoCASL specifications for the process algebras CCS and CSP including established notions of process equivalence, namely strong bisimulation, weak bisimulation, observation congruence, and trace equivalence, in the latter case illustrating how algebraic and coalgebraic notions interact in CoCASL. In general, our specifications deal with the concepts involved in a natural way, indicating that CoCASL is an expressive language which is able to deal with reactive systems at an appropriate level.

CoCASL is more expressive than other algebra-coalgebra combinations in the literature: [11] uses a simpler logic, CCSL [41] has fewer datatypes available, while hidden algebra such as in BOBJ [37] and reachable-observable algebra such as in COL [8] do not support cofree types. If, for example, streams are specified not as the final (=cofree) model, then there are stream models which do not contain some corecursively definable functions (like the flipping of streams). Hence, corecursive definitions in general fail to be conservative.

By contrast, cofree cotypes in CoCASL support a style of specification separating the basic process type (with its data sorts, observers and other operations) from further, derived operations defined on top of this in a conservative way. Note that this is not a purely theoretical question: programming languages such Charity [12] and Haskell [19] support infinite datastructures that correspond to the infinite trees in the behaviour algebras, and one should be able to specify that as many infinite trees as needed for all programs over some datastructure expressible in these languages are present in the models of a specification. The Haskell semantics for lazy datastructures (at least for the non-left-$\rightarrow$-recursive case) indeed comprises *all* infinite trees, i.e. is captured

exactly by a behaviour algebra.

The logic COL *Constructor-based observational logic* [8] combines reachability induced by constructors with observational equality induced by observers. CoCasl does not directly support observational equality or bisimilarity, but it can be expressed via cogeneration constraints, as shown in the process algebra examples. In COL, observability is a global notion and required to be preserved and reflected by signature morphisms. CoCasl's local notion of observability provides an extra degree of flexibility — in particular, it allows instantiating observable sorts with non-observable ones. Unlike COL, CoCasl does not simultaneously support a glass-box and a black-box view on a specification. However, we plan to develop a notion of behavioural refinement between CoCasl specifications. Then, the black-box/glass-box view of [8] could be expressed in CoCasl as a refinement of a black-box specification into a glass-box one, thus also providing a clear separation of concerns.

The *Coalgebraic Class Specification Language* CCSL [41], developed in close cooperation with the LOOP project [49], is based on the observation of [34] that coalgebras can give a semantics to classes of object-oriented languages. CCSL provides a notation for parametrized class specifications based on final coalgebras. Its semantic is based on a higher-order equational logic and it provides theorem proving support by compilers that translate CCSL into the higher-order logic of PVS and Isabelle. In its current version, CCSL does not support data type specifications with partial constructors, axioms or equations, i.e. it only supports free types without axioms in the sense of Casl. This also implies that, in contrast to CoCasl, CCSL does not support modalities for coalgebras mapping states to finite sets of states (since finite sets are defined by a structured free specification using equational axioms). Recently, CCSL has been extended by *binary methods* [48] (i.e. observers with two non-observable arguments). These are also available in CoCasl and can be used in connection with cogeneration (= full abstraction) constraints, while cofree models usually do not exist. For example, we specify the bisimulation relation in CSP as a binary method.

Future work will concentrate in particular on the development of tools for CoCasl. The *heterogeneous tool set* [25] already provides a parser and static analysis for Casl and CoCasl structured specification; the extension of the analysis tools for Casl basic specifications to CoCasl will be easy. Concerning proof support, it is planned to extend the coding of Casl into Isabelle/HOL [26] to CoCasl. While cogenerated and cofree cotypes are easily expressible (and partly already available in Isabelle/HOL), structured cofree specifications will be a challenge. Moreover, we expect that recent research about circular coinduction [15] and terminal sequence induction [30] will provide useful tactics for the encoding of CoCasl into Isabelle/HOL.

Acknowledgements

The authors would like to thank the participants of an informal CoCasl and observability meeting, Hubert Baumeister, Michel Bidoit, Rolf Hennicker, Bernd Krieg-Brückner, Don Sannella, Andrzej Tarlecki, and Martin Wirsing, for intensive feedback to a draft version of this work. We also thank Hendrik Tews for useful discussions on the relation between CCSL and CoCasl.

References

[1] Casl – *The* CoFI *Algebraic Specification Language – Semantics*, Note S-9 (Documents/CASL/Semantics, version 0.96), in [13], July 1999.

[2] Casl – *The* CoFI *Algebraic Specification Language – Summary, version 1.0.1*, Documents/CASL/Summary, in [13], March 2001.

[3] J. Adámek, H. Herrlich, and G. E. Strecker, *Abstract and concrete categories*, Wiley Interscience, 1990.

[4] M. Arbib and E. Manes, *Parametrized data types do not need highly constrained parameters*, Inform. Control **52** (1982), 139–158.

[5] E. Astesiano, M. Bidoit, B. Krieg-Brückner, H. Kirchner, P. D. Mosses, D. Sannella, and A. Tarlecki, Casl – *the Common Algebraic Specification Language*, Theoret. Comput. Sci. **286** (2002), 153–196.

[6] M. Barr, *Terminal coalgebras in well-founded set theory*, Theoret. Comput. Sci. **114** (1993), 299–315.

[7] J.A. Bergstra, A. Ponse, and S.A. Smolka, *Handbook of process algebra*, Elsevier, 2001.

[8] M. Bidoit and R. Hennicker, *On the integration of observability and reachability concepts*, Foundations of Software Science and Computation Structures, LNCS, vol. 2303, Springer, 2002, pp. 21–36.

[9] P. Burmeister, *Partial algebras — survey of a unifying approach towards a two-valued model theory for partial algebras*, Algebra Universalis **15** (1982), 306–358.

[10] Allan Cheng and Mogens Nielsen, *Open maps (at) work*, Tech. Report RS-95-23, BRICS, 1995.

[11] C. Cîrstea, *On specification logics for algebra-coalgebra structures: Reconciling reachability and observability*, LNCS **2303** (2002), 82–97.

[12] R. Cockett and T. Fukushima, *About Charity*, Yellow Series Report 92/480/18, Univ. of Calgary, Dept. of Comp. Sci., 1992.

[13] CoFI, *The Common Framework Initiative, electronic archives*, Notes and documents accessible from http://www.cofi.info.

[14] J. Goguen and R. Burstall, *Institutions: Abstract model theory for specification and programming*, J. ACM **39** (1992), 95–146.

[15] J. Goguen, K. Lin, and G. Rosu, *Conditional circular coinductive rewriting*, Automated Software Engineering, IEEE Press, 2000, pp. 123–131.

[16] Charles Antony Richard Hoare, *Communicating sequential processes*, Prentice Hall, 1985.

[17] B. Jacobs, *Towards a duality result in the modal logic of coalgebras*, Coalgebraic Methods in Computer Science, Electron. Notes Theoret. Compt. Sci., vol. 33, 2000.

[18] _____, *Many-sorted coalgebraic modal logic: a model-theoretic study*, Theor. Inform. Appl. **35** (2001), 31–59.

[19] S. P. Jones, J. Hughes, L. Augustsson, D. Barton, B. Boutel, W. Burton, J. Fasel, K. Hammond, R. Hinze, P. Hudak, T. Johnsson, M. Jones, J. Launchbury, E. Meijer, J. Peterson, A. Reid, C. Runciman, and P. Wadler, *Haskell 98 language and libraries: The revised report*, (2002), http://www.haskell.org/onlinereport.

[20] D. Kozen, *Results on the propositional μ-calculus*, Theoret. Comput. Sci. **27** (1983), 333–354.

[21] A. Kurz, *Specifying coalgebras with modal logic*, Theoret. Comput. Sci. **260** (2001), 119–138.

[22] _____, *Logics admitting final semantics*, Foundations of Software Science and Computation Structures, LNCS, vol. 2303, Springer, 2002, pp. 238–249.

[23] Robin Milner, *Communication and concurrency*, Prentice Hall, 1989.

[24] L. Moss, *Coalgebraic logic*, Ann. Pure Appl. Logic **96** (1999), 277–317.

[25] T. Mossakowski, *Implementing logics: from genericity to heterogeneity*, Technical report, University of Bremen.

[26] _____, CASL*: From semantics to tools*, Tools and Algorithms for the Construction and Analysis of Systems, LNCS, vol. 1785, Springer, 2000, pp. 93–108.

[27] _____, *Relating* CASL *with other specification languages: the institution level*, Theoret. Comput. Sci. **286** (2002), 367–475.

[28] T. Mossakowski, M. Roggenbach, and L. Schröder, COCASL *at work — modelling process algebra*, Coalgebraic Methods in Computer Science, Electron. Notes Theoret. Comput. Sci., vol. 82, 2003.

[29] P. D. Mosses (ed.), CASL *— the Common Algebraic Specification Language*, Springer, To appear.

[30] D. Pattinson, *Expressive logics for coalgebras via terminal sequence induction*, Tech. report, LMU München, 2002.

[31] J. Power and H. Watanabe, *An axiomatics for categories of coalgebras*, Coalgebraic Methods in Computer Science (CMCS 98) (B. Jacobs, L. Moss, H. Reichel, and J. Rutten, eds.), ENTCS, vol. 11, Elsevier, 2000.

[32] G. Reggio, E. Astesiano, and C. Choppy, CASL-LTL — *a* CASL *extension for dynamic reactive systems — summary*, Tech. Report DISI-TR-99-34, Università di Genova, 2000.

[33] G. Reggio and L. Repetto, CASL-CHART*: a combination of statecharts and of the algebraic specification language* CASL, Algebraic Methodology and Software Technology, LNCS, vol. 1816, Springer, 2000,, pp. 243–257.

[34] H. Reichel, *An approach to object semantics based on terminal co-algebras*, Math. Struct. Comput. Sci. **5** (1995), 129–152.

[35] ———, *A uniform model theory for the specification of data and process types*, Workshop on Algebraic Development Techniques, LNCS, vol. 1827, Springer, 2000, pp. 348–365.

[36] H. Reichel, T. Mossakowski, M. Roggenbach, and L. Schröder, *Algebraic-coalgebraic specification in* COCASL, Recent Developments in Algebraic Development Techniques, 16th International Workshop (WADT 02), LNCS, Springer, 2003, to appear.

[37] G. Roşu, *Hidden logic*, Ph.D. thesis, Univ. of California at San Diego, 2000.

[38] M. Roggenbach, CSP-CASL — *a new integration of process algebra and algebraic specification*, Third AMAST Workshop on Algebraic Methods in Language Processing (AMiLP-3), TWLT, University of Twente, 2003, to appear.

[39] A.W. Roscoe, *The theory and practice of concurrency*, Prentice Hall, 1998.

[40] M. Rößiger, *Coalgebras and modal logic*, Coalgebraic Methods in Computer Science, Electron. Notes Theoret. Compt. Sci., vol. 33, 2000.

[41] J. Rothe, H. Tews, and B. Jacobs, *The Coalgebraic Class Specification Language CCSL*, J. Universal Comput. Sci. **7** (2001), 175–193.

[42] J. Rutten, *Universal coalgebra: A theory of systems*, Theoret. Comput. Sci. **249** (2000), 3–80.

[43] G. Salaün, M. Allemand, and C. Attiogbé, *A formalism combining CCS and* CASL, Tech. Report 00.14, University of Nantes, 2001.

[44] ———, *Specification of an access control system with a formalism combining CCS and* CASL, Parallel and Distributed Processing, IEEE, 2002,, pp. 211–219.

[45] L. Schröder and T. Mossakowski, *Monad-independent dynamic logic in* HASCASL, J. Logic Comput., to appear. Short Version to appear in Recent Developments in Algebraic Development Techniques, 16th International Workshop (WADT 02), LNCS, Springer, 2003.

[46] ———, *Monad-independent Hoare logic in* HasCasl, Fundamental Approaches to Software Engineering, LNCS, vol. 2621, Springer, 2003, pp. 261–277.

[47] A. Tarlecki, *On the existence of free models in abstract algebraic institutions*, Theoret. Comput. Sci. **37** (1985), 269–304.

[48] H. Tews, *Coalgebraic methods for object–oriented languages*, Ph.D. thesis, Technical Univ. of Dresden, 2002.

[49] J. van den Berg and B. Jacobs, *The LOOP compiler for Java and JML*, LNCS **2031** (2001), 299–312.