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 equationally specified initial datatypes as observations, 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 specifying the reactive behaviour of systems [52]. Generally, software specifications consist of collections of symbols called *signatures* together with axioms expressed in this signature, the idea being that the signature describes the components and operations of a software system and the axioms constitute requirements on its behaviour, which may e.g. have reactive and functional aspects. While algebraic specification deals with functional behaviour, typically using *inductive datatypes* generated by constructors, coalgebraic specification in concerned with reactive behaviour modelled by *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 [38].

For algebraic specification, the Common Algebraic Specification Language Casl [13] 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, 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 datatype-valued observers.

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 formulae. 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 formulae play the same role in coalgebra as equations do in algebra [23,24].

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 define an institution for COCASL's modal logic. 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 [45]; the process algebra example has appeared in [33].

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*. Its features include first-order logic, partial functions, subsorts, sort generation constraints, and structured and architectural specifications. For the language definition and a full formal semantics cf. [8,34]. 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 [31,34]; examples of actual CASL specifications will appear in later sections. A *many-sorted* CASL *signature* $\Sigma = (S, TF, PF, P)$ consists of a set S of sorts, two $S^* \times S$-indexed sets $TF = (TF_{w,s})$ and $PF = (PF_{w,s})$ of total and partial operation symbols, and an S^*-indexed set $P = (P_w)$ of 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 [9], i.e. whenever a partial function is defined on congruent tuples of arguments, then the results are congruent). 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 (stating that a partial function is defined for certain arguments), a strong equation (stating equivalence of definedness of its two sides and equality in case of definedness), an existence equation (stating definedness and equality of its two sides), or a predicate application; see [13,5] 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. Note that Γ does not only contain explicitly stated sentences, but also sentences that are generated e.g. by CASL's powerful datatype constructs (see below), like the statement that selectors are one-sided inverses of their constructor.

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. Models and satisfaction of sentences 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. Moreover, CASL allows *structured specification*, i.e. specification in the large by modular composition of basic specifications. In terms of structuring operations, CASL offers (possibly parametrized) named specifications (keyword **spec**), unions of specifications (keyword **and**), extensions of specifications (keyword **then**), free specifications **free** $\{\ldots\}$, and renaming as well as hiding of symbols. A specification Sp_1 **then** Sp_2 determines the signature Σ obtained by extending the signature of Sp_1 by the symbols of Sp_2; its class of models consists of all Σ-models M that reduce to a model of Sp_1 and satisfy the conditions imposed by Sp_2. Free specifications are recalled in more detail in Section 6.

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 (typically with equations)	**cofree { ... }** = final process type (typically with modal axioms)

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

Generally, CASL employs *linear visibility*, i.e. all symbols must be declared before they can be used (an exception is made for mutually recursive datatypes). Declared symbols can be *redeclared*. In particular, one may write a **type** declaration for previously declared sorts; e.g. the specification

sort *Nat*
free type $Nat ::= 0 \mid suc(Nat)$

is legal. 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 precede 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 dualizations of CASL concepts in COCASL

At the level of basic specifications, the duality addresses the various forms of the **types** construct that serves to define inductive datatypes 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, 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 cotypes** declaration, like a **free types** declaration, is a conservative extension as long as the sorts it declares 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 formulae 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 within free specifications.

3 Type and cotype definitions

The basic CASL construct for type definitions is the **type** construct. A type declaration of the form

$$\begin{aligned}\textbf{types}\ \ & t_1 := c_{11}(s_{111};\ldots;s_{11k_{11}}) \mid \cdots \mid c_{1r_1}(s_{1r_1 1};\ldots;s_{1r_1 k_{1r_1}}) \\ & \ldots \\ & t_n := c_{n1}(s_{n11};\ldots;s_{n1k_{n1}}) \mid \cdots \mid c_{nr_n}(s_{nr_n 1};\ldots;s_{nr_n k_{nr_n}})\end{aligned}$$

declares *constructors* $c_{ij} : s_{ij1} \times \cdots \times s_{ijk_{ij}} \to t_i$ for datatypes $t_1, \ldots, t_n$, where linear visibility is relaxed to allow the t_i to appear among the s_{ijk}; the parts of the declaration separated by vertical bars are called *alternatives*. Optionally, *selectors* can be specified: replacing a plain argument sort s_{ijk} by $sel_{ijk} : s_{ijk}$ in the above specification declares a selector $sel_{ijk} : t_i \to s_{ijk}$, for which an

spec CONTAINER [**sort** *Elem*] =
  type
    Container ::= *empty* | *insert*(*first* :? *Elem*; *rest* :? *Container*);

spec NTREE [**sort** *Elem*] =
  types
    NTree ::= *fork*(*Elem*; *Forest*)
    Forest ::= *null* | *grow*(*NTree*; *Forest*)

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

axiom $def\ c_{ij}(x_1, \ldots, x_{k_{ij}}) \implies sel_{ijk}(c_{ij}(x_1, \ldots, x_{k_{ij}})) = x_k$ is generated. Both constructors and selectors may be partial (indeed, for selectors this is even typical whenever a type has more than one constructor). 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 (the keyword **spec** is used to name specifications for later reference). 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 $first(insert(x, y)) = x$ 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*). The specification NTREE in Figure 2 illustrates the declaration of several mutually recursive types 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

type *Int* ::= **sort** *Nat* | $-\_\_(Pos)$

Moreover, we also allow additional parameters for the observers. These have to come from the local environment (recall that the latter 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 do not have a corresponding constructor, since the constructor would need to have a higher-order type — e.g. in the above example $(In \to State) \to State$ — which is unavailable in CASL.

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 *domain* of an observer can thus be defined as a subset of the associated cotype,
- the domains 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 Proposition 2 below).

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

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

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_{ijk} is the k-th observer of the j-th alternative in the **cotype** definition of T_i. The result sort T_{ijk} of the observer 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 $obs_{ijk} : s_1 \times \cdots \times s_m \to s$, 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 $obs_{ijk} : T_i \to T_{ijk}$ for the type of the observer, the corresponding T_{ijk} is not simply a sort, but a function space

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

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

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

which is just $T_i \to T_{ijk}$. 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.

The *induced theory* of the cotype consists of the signature $Sig(CT)$ and the axioms generated by the cotype declaration as described above. The induced theory is also referred to as CT. An *S-palette* is an S-sorted family $C = (C_s)$ of sets of *colours*; a *C-colouring* is a a family h of maps $(h_s : A_s \to C_s)_{s \in S}$). A CT-algebra A is called a *(CT, C)-algebra* if A interprets the non-observable sorts as prescribed by C, i.e. $A_s = C_s$ for all $s \in S$; a *homomorphism of (CT, C)-algebras* is a CT-algebra homomorphism that acts as the identity on the sorts in S.

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 with induced theory CT and set S of observable sorts, one can associate a functor $F : \mathbf{Set}^n \to \mathbf{Set}^n$ such that, for each S-palette C, the category of (CT, C)-algebras is isomorphic to the category of F-coalgebras. In particular, this implies that all homomorphisms between (CT, C)-algebras are closed.*

PROOF. We begin with the parameterless case, without any local environment, i.e. we have a cotype

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

By abuse of notation, we 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 axioms in Γ ensure that each T_i is the disjoint sum of sets X_{ij}, where X_{ij} is the domain of definition of the observers in the j-th alternative of the cotype declaration for T_i. Thus, we can regard CT-algebras as coalgebras

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

on $\mathbf{Set}^n$ by taking g_i to be defined on X_{ij} by $g_i(x) = (obs_{ij1}(x), \ldots, obs_{ijr_{ij}}(x))$. It is easy to reverse this process: given an F-coalgebra A, the tuple $\langle obs, \ldots, obs_{ijr_{ij}} \rangle$ of observers in the j-th alternative for T_i is defined as the restriction of the i-th component of the structure map of A to the preimage of the j-th summand $\prod_k T_{ijk}$ of the i-th component of $F(T_1, \ldots, T_n)$. Altogether, this leads to a bijective correspondence between CT-algebras and F-coalgebras. Furthermore, this correspondence is functorial, i.e. a tuple $h = (h_1, \ldots, h_n) : M \to N$ of maps is a homomorphism of CT-algebras iff it is a homomorphism of the corresponding F-coalgebras. This equivalence is due to the fact that homomorphisms of partial algebras preserve definedness and hence respect the disjoint decompositions of the T_i, so that h_i can be decomposed into m_i maps between the disjoint summands of T_i. It is straightforward to generalize these arguments to the case that S is non-empty. $\square$

Definition 3 Let S be a set of sorts called *observable* sorts, let Σ be a signature such that S is contained in the sorts of Σ, and let M be a Σ-model. 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. That is, for $(a_1, \ldots, a_n), (b_1, \ldots, b_n) \in M_w$ such that $a_i \; R \; b_i$, $i = 1, \ldots, n$, we have
  - for $f \in TF_{w,s} \cup PF_{w,s}$, $(f_{w,s})_M(a_1, \ldots, a_n)$ is defined iff $(f_{w,s})_M(b_1, \ldots, b_n)$ is defined, and then

$$(f_{w,s})_M(a_1, \ldots, a_n) \; R \; (f_{w,s})_M(b_1, \ldots, b_n).$$

  - for $p \in P_w$,

$$(a_1, \ldots, a_n) \in (p_w)_M \iff (b_1, \ldots, b_n) \in (p_w)_M.$$

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

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

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

Remark 4 It is easy to check that for cotypes, the notion of bisimulation defined above agrees with the notion arising from the coalgebraic definition given e.g. in [52] and the correspondence between coalgebras and cotypes according to Proposition 2.

Note that Proposition 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

$$\begin{aligned}\textbf{cotype}\ \mathit{Process} ::= {} & (\mathit{hd1} :? \mathit{Elem}; \mathit{next} :? \mathit{Process}) \\ & \mid (\mathit{hd2} :? \mathit{Elem}; \mathit{left} :? \mathit{Process}; \mathit{right} :? \mathit{Process})\end{aligned}$$

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 should 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.

In summary, **type** declarations provide useful abbreviations for signatures of algebras, while **cotype** declarations provide useful abbreviations for theories of coalgebras, the latter being formalized as partial algebras.

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:

$$\begin{aligned}&\textbf{spec}\ \textsc{FiniteSet}\ [\textbf{sort}\ \mathit{Elem}] = \\ &\quad \textbf{generated type}\ \mathit{FinSet}[\mathit{Elem}] ::= \{\} \mid \_\_ + \_\_(\mathit{Elem}; \mathit{FinSet}[\mathit{Elem}]);\end{aligned}$$

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

Fig. 3. Cogenerated specification of bit streams in CoCASL

.

Here, the generatedness constraint means that all finite sets can be constructed by applications of $\{\}$ and $\_\_ + \_\_$. More generally, a generatedness constraint consists of a set of sorts and a set of operation symbols (called constructors) in a signature. It is fulfilled in a model, if each element of each carrier for a constrained sort is the value of some constructor term with variables in non-constrained sorts (where the variables may be interpreted with arbitrary values from non-constrained sorts).

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.)

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.

Given a signature $\Sigma = (S, TF, PF, P, \leq)$, a *cogeneration constraint* over a signature is a subsignature fragment (i.e. a tuple of subsets of the respective signature components, which 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 . . . }**). Example 11 below is intended as a deterrent against the use of types where cotypes are expected.

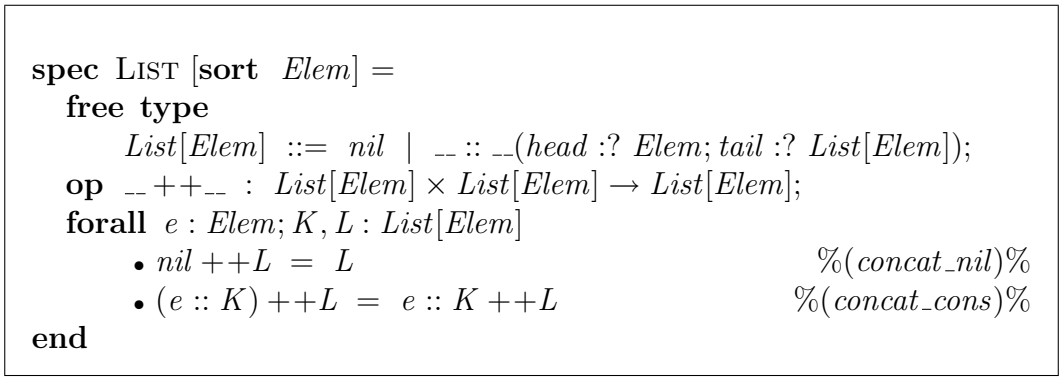

Fig. 4. Specification of lists over an arbitrary element sort in CASL.

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 5 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 [56] (i.e. observers with two non-observable argument sorts) is expressible.

Remark 6 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) — 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

```
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.

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 [3] (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 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 infinite depth and branching, dualizing the *Ntree* example of Figure 2.

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

Definition 7 Given a cotype

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

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.

and an S-palette C, the *behaviour algebra* $Beh_{CT}(C)$ is defined to be the following (CT, C)-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 inner node with label (T_i, j) has one child for each of the observers obs_{ijk} $(k = 1 \ldots r_{ij})$ 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*

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

and an S-palette C, the behaviour algebra $Beh_{CT}(C)$ *is final in the category of (CT, C)-algebras (note that the latter correspond to coalgebras).*

PROOF. Using the characterization of Proposition 2, the result follows from the general construction of final coalgebras for polynomial functors over the category of $\{T_1, \ldots, T_n\}$-sorted sets (this generalizes the well-known result

for **Set** [2,3]). Intuitively, the morphism from a given (CT, C)-algebra into $Beh_{CT}(C)$ 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

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

is a cotype with $Sig(CT) \subseteq \Sigma$, as Σ-sentences to the CoCasl 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 $(S, \{obs_{ijk} | obs_{ijk} \text{ total}\}, \{obs_{ijk} | obs_{ijk} \text{ partial}\}, \emptyset)$, 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 behaviours, and up to isomorphism, we have a submodel of the cofree model. Hence, cofreeness essentially adds the requirement that each possible behaviour is actually represented by an element.

Full abstractness of cofree cotypes 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 such a way that the satisfaction condition for institutions is fulfilled, one has to equip the constraints with an additional signature morphism, as in [5,31].

```
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.

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 (cf. Section 1 for CASL's notion of model, which is used also for CoCASL). 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 (cf. Section 1 for the meaning of **then**) that are free over $M|_{\Sigma_1}$ w.r.t. the reduct functor $\_|_{\Sigma_1}$ associated to the inclusion of Σ_1 into the signature of Sp_1 **then** Sp_2. 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 a 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, 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, i.e. that M is free over $M|_{\Sigma_1}$ with $\_|_{\Sigma_1}$-universal arrow

$id : M|_{\Sigma_1} \to M|_{\Sigma_1}$, which is stronger than fibre-initiality of M. 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 cotype declarations, then*

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 $Bool ::= false \mid true$
then
cofree { type $T ::= c1(s1 :?Bool) \mid c2(s2 :?Bool)$ **}**

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

```
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.

```
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.

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 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 [44]. 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. This specification extends the specification of non-deterministic automata of Figure 9 by an outer (structured) cofreeness constraint, so that its model class now consists only of models where the cotype *State* is 'the' final non-deterministic

```
spec ListStreamTree [sort Elem] =
  free type
      Tree ::= EmtpyTree
             | Tree(left :? Tree; 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.

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

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

automaton (determined uniquely up to isomorphism) over the interpretation of *In* rather than the class of all non-deterministic automata. Here, like in Figure 9, the inner **free** has to be a structured one, since finite sets cannot be specified as **free type** directly. In principle, **free** and **cofree** can be nested arbitrarily; however, care must be taken to ensure that this does not lead to inconsistencies. A general consistency criterion that covers nestings of the type used in Figure 13 is given in Section 10.

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 [18,19,23,51]. 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. Modal logic allows formulating statements about

$$
\begin{aligned}
\varphi ::= \quad & t_1 = t_2 \\
| \; & t_1 \stackrel{e}{=} t_2 \\
| \; & def\ t \\
| \; & [t]\,\varphi \\
| \; & \langle t \rangle\,\varphi \\
| \; & [t*]\,\varphi \\
| \; & \langle t* \rangle\,\varphi \\
| \; & [\{t_1, \ldots, t_n\}]\,\varphi \\
| \; & \langle \{t_1, \ldots, t_n\} \rangle\,\varphi \\
| \; & [\{t_1, \ldots, t_n\}*]\,\varphi \\
| \; & \langle \{t_1, \ldots, t_n\}* \rangle\,\varphi
\end{aligned}
$$

Fig. 14. Syntax of CoCasl's modal logic.

such systems without explicit reference to the states. The effect is that axioms formulated in modal logic indeed describe only the observable behaviour of a system, formally: satisfaction of modal formulae is bisimulation invariant (see Section 10 and e.g. [22,35]). Methodologically, this means that the state space is appropriately encapsulated; a technical advantage is that restriction by modal formulae preserves existence of final models (cf. Section 10).

In CoCasl, this takes the following shape. We define modal formulae 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.

The full syntax of CoCasl's modal logic is given in Fig. 14 and explained successively in the sequel. Note that the syntax does not include propositional variables, since these would violate invariance under bisimulation. Atomic formulae 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. The modal logic has (existential or strong) equations between as well as definedness assertions of observer terms as atomic sentences. Sentences may be combined using the usual propositional connectives, the quantification over variables of observable sorts, as well as the following modalities: An observer t (possibly applied to extra parameters) with *non-observable* result sort leads to modalities $[t]$, $\langle t \rangle$, $[t*]$, $\langle t* \rangle$ (all-next, some-next, always, eventually). 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 the meaning of the formula

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

Here, $[\![\_]\!]_{\_}$ takes a modal formula (or an observer term) and a sorted term to an ordinary formula (or ordinary term). Intuitively, the sorted term, which is written as a subscript, carries the current state. This is defined as follows:

- $[\![u]\!]_{t:s} \equiv u$, if u is an observer term consisting of variables and operation symbols from the local environment,
- $[\![f]\!]_{t:s} \equiv f(t)$ 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 occurring 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.

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

- $[\![\langle f \rangle \varphi]\!]_{t:s} \equiv \neg [\![[f] \neg \varphi]\!]_{t:s}$

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

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

}

The starred modalities have the limitation that only *one* specific observation can be repeated arbitrary often. However, sometimes it is desirable to express that a group of observations can be repeated. We hence allow for grouping observers with braces: $[\{f_1, \ldots, f_n\}]$ and $\langle\{f_1, \ldots, f_n\}\rangle$ denote the conjunction and the disjunction, respectively, of the modal formulae obtained for the individual observers. Note that for the unstarred versions, this also can be expressed explicitly as a conjunction (disjunction), while this is not possible for the starred versions. This machinery allows us to express that a buffer eventually outputs all elements that are read in as follows:

$$\forall a : Elem \,.\, [next(input(a))]\, \langle\{next(input), next(output)\}*\rangle\, \langle next(output(a))\rangle\, true$$

Note that in the example of Sect. 11.5, $\langle\{next(input), next(output)\}*\rangle$ can also be expressed as $\langle next*\rangle$. However, in general there may be operations beyond *input* and *output*, such that this is not a semantically equivalent abbreviation.

The modal logic introduced above allows expressing safety or fairness properties. For example, the model of the specification BitStream4 of Figure 15 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

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

Fig. 15. Specification of a fairness property.

against the grain of CoCasl, which is first-order in spirit (although higher-order types can be emulated).

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 $next : In \times State \to Set$ 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 $\langle next(i)\rangle\phi$ will express that there *exists* a successor state that satisfies ϕ.

In CoCasl, a standard way to describe functors is as algebraic datatypes:

Definition 13 A functor $T : \mathbf{Set}^n \to \mathbf{Set}$ is called a *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. More precisely, T is a datatype if it is of the form $P_i \circ S$, where P_i is the i-th projection $\mathbf{Set}^n \to \mathbf{Set}$ and $S : \mathbf{Set}^n \to \mathbf{Set}^n$ is the free algebra monad associated to an n-sorted (total) algebraic signature Σ and a set E of Σ-equations in the usual sense of universal algebra. The triple (Σ, E, i) is called a *presentation* of T. Given such a presentation, a *constructor term* over a family $\bar{X} = (X_j)$ of n sets is a Σ-term over the variable supply $\bar{X}$ (i.e. X_j is the set of variables for the j-th sort). Every constructor term over $\bar{X}$ represents an element of $T\bar{X}$. A datatype T is called *non-cancellative* if E can be chosen in such a way that each equation $\alpha = \beta$ in E satisfies the *variable restriction* stating that α and β 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 14 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 (cf. Figure 9); 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 datatype that fails to be non-cancellative is the free abelian group monad, which takes a set X to the set of maps $X \to \mathbb{Z}$ with finite support; such maps are written in the form $\sum_{i=1}^{k} n_i x_i$, with $k \in \mathbb{N}$, $n_i \in \mathbb{Z}$ and $x_i \in X$, $i = 1, \ldots, k$. Here, the equation $x - x = 0$ violates the variable restriction.

Definition 15 Let $T : \mathbf{Set}^n \to \mathbf{Set}$ be a functor. Let $\bar{X} = (X_j)$ 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_i 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 we put

$$[\![t]\!]\,\phi :\iff (\bar{T}_{\bar{X},i}\phi)\,t = (\bar{T}_{\bar{X},i}\top)\,t,$$

where $\top$ denotes the constant true predicate.

Example 16 In typical datatypes (cf. Example 14), the above definition is made explicit as follows.

- If T is the finite power set functor, then $[\![t]\!]\,\phi$ holds iff all elements of t satisfy ϕ.
- If T is the list functor, then $[\![t]\!]\,\phi$ holds iff all entries of t satisfy ϕ.
- If T is the functor 'multiplication with 2', i.e. $TX = X + X$, then $[\![t]\!]\,\phi$ holds iff $\delta_X(t)$ satisfies ϕ, where δ_X is the codiagonal $[id, id] : X + X \to X$.
- If T is the finite multiset functor, then $[\![t]\!]\,\phi$ holds iff all elements occurring in t with non-zero multiplicity satisfy ϕ.
- If T is the free abelian groups functor, then $[\![t]\!]\,\phi$ holds iff the elements violating ϕ have combined multiplicity 0 in t; e.g. one has $[\![a - b]\!]\,\mathit{false}$.

For datatypes in general, the action of the arising functor T on maps can in principle be defined in CoCasl by primitive recursion; e.g. for the datatype of finite sets specified as in Figure 9, instantiated for the sort *State* as in the specification and, additionally, for a type of booleans, we can define $T\phi$ for a predicate ϕ (written *phi* below) on *State* as the function *setphi* determined by

var $s : \mathit{State}, t1, t2 : \mathit{Set}$
- $\mathit{setphi}(\{\}) = \{\}$
- $\mathit{setphi}(\{s\}) = \{\mathit{phi}(s)\}$
- $\mathit{setphi}(t1 \cup t2) = \mathit{setphi}(t1) \cup \mathit{setphi}(t2)$

Analogously, one defines a function *settrue* representing $T\top$, and then $[\![t]\!]\,\phi$ can be written in CoCasl as the equation $\mathit{setphi}(t) = \mathit{settrue}(t)$.

We note

Proposition 17 *In the notation of Definition 15, we have*

$$[\![(\bar{T}_{\bar{X},i} f)t]\!]\,\phi \iff [\![t]\!]\,(\phi \circ f)$$

for every map $f : X_i \to Y$.

PROOF. Just note that $(\bar{T}_{X,i}\phi)(\bar{T}_{X,i}f)t = (\bar{T}_{X,i}(\phi \circ f)t$ and $(\bar{T}_{X,i}\top)(\bar{T}_{X,i}f)t = (\bar{T}_{X,i}\top)t$. □

In the terminology of [22,35], the above proposition states that the assignment $(t, \phi) \mapsto [\![t]\!]\,\phi$ is a predicate lifting.

The last item in the Example 16 shows that modal operators arising from Definition 15 (see below) will in general fail to be normal. However, we have

Theorem 18 *Let* $T : \mathbf{Set}^n \to \mathbf{Set}$ *be a non-cancellative datatype with presentation* (Σ, E, i). *In the notation of Definition 15, the following are equivalent:*

(i) *t can be represented by a constructor term* α *such that every element of* X_i *appearing in* α *as a free variable satisfies* ϕ*;*
(ii) *for every representation of t by a constructor term* α*, every element of* X_i *appearing in* α *as a free variable satisfies* ϕ*;*
(iii) $[\![t]\!]\,\phi$.

Corollary 19 *Let T be a non-cancellative datatype. In the notation of the above theorem,* $[\![t]\!]\,\phi$ *implies* $[\![t]\!]\,\psi$ *whenever* ϕ *implies* ψ. *Moreover,*

$$(\forall i.\, [\![t]\!]\,\phi_i) \iff [\![t]\!]\,(\forall i.\phi_i)$$

for every family of predicates ϕ_i.

Note that the corollary fails to generalize to arbitrary datatypes. E.g. for the free abelian groups functor (cf. Example 14), by the description of $[\![t]\!]\,\phi$ given in Example 16 we have $[\![a - b]\!]\,\mathit{false}$, but not $[\![a - b]\!]\,\{a\}$.

PROOF (Theorem 18). *(i)* $\Longleftrightarrow$ *(ii):* By the variable restriction, all representations of t by constructor terms use the same variables from X_i.

(ii) $\Longrightarrow$ *(iii):* Let α be a constructor term representing t. Then a representing term of $(\bar{T}_{X,i}\phi)t$ is obtained by replacing every variable $x \in X_i$ occurring in α by $\phi(x)$. Since by assumption $\phi(x) = \top$ for such x, the resulting term represents also $(\bar{T}_{X,i}\top)t$.

(iii) $\Longrightarrow$ *(ii):* Let x occur in a representation α of t. Then $\phi(x)$ occurs in a representation of $(\bar{T}_{X,i}\phi)t$, hence by the variable restriction also in the representation of $(\bar{T}_{X,i}\top)t = (\bar{T}_{X,i}\phi)t$ obtained by substituting each element of X_i occurring in α by $\top$. Thus, $\phi(x) = \top$. □

Remark 20 For non-cancellative datatypes T, the map predicate lifting $\phi \mapsto \lambda t.\, [\![t]\!]\, \phi$ arises from a natural relation in the sense of [36], i.e. (in the single-sorted case) from a natural transformation $\mu : T \to \mathcal{P}$ into the covariant power set functor $\mathcal{P}$; here, μ_X takes a term $t \in TX$ to the set of variables contained in t. Even for non-cancellative types, it is not in general the case that the arising modal logic is expressive, i.e. distinguishes non-bisimilar states. This fails e.g. for the finite multiset functor, i.e. the functor that takes a set X to the set of maps $X \to \mathbb{N}$ with finite support.

Given these definitions, the need arises for CoCasl language constructs that mark datatypes intended as functor definitions for modal operators. We therefore introduce a new semantic annotation %**modal** for free specifications. Explicitly, the annotation

Sp₁ **then free** %**modal { ***Sp₂* **}**

is well-formed 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;
- 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 datatype functor $T : \mathbf{Set}^n \to \mathbf{Set}$, where the n arguments, called the *type parameters* of the datatype, correspond to the sorts declared in Sp_1.

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 in the above sense, 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; here, ϕ can be a modal formula of any type V that appears as a type parameter of W. 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 impossible due to the freshness condition above. Thus, the modal operator $[f(r_1, \ldots, r_n)]$ is unambiguously associated with a datatype functor $T : \mathbf{Set}^n \to \mathbf{Set}$ 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 21 Let s be of type S. Then

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

where the right hand formula is to be read according to Definition 15 (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 $\langle f(r_1, \ldots, r_n)\rangle$ 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. E.g. given the specification of non-deterministic automata of Figure 9, $\langle next(i)\rangle\phi$ holds in a state s iff $next(s)$ contains an element satisfying ϕ, $[next]\phi$ holds iff for all inputs i, all elements of $next(i)(s)$ satisfy ϕ, and $\langle next*\rangle\phi$ holds iff there exists a sequence of inputs through which a state satisfying ϕ can be reached from s.

Remark 22 It should be noted that, by the above, the structured modal operator $[f(r_1, \ldots, r_n)]$ is overloaded in the sense that it applies to modal formulae of type V for all type parameters V of T. In practice, this means that parsing modal formulae requires overloading resolution in much the same style as for overloaded CASL terms. In the case of ambiguities, atomic modal formulae can always be disambiguated by providing explicit type annotations for flexible constants, the only exception to this being the atomic formulae *true* and *false*. Beyond this, COCASL allows disambiguating modal operators by annotating them explicitly with the intended type of the following modal formula in the form $[f(r_1, \ldots, r_n) : V]$. (Confusion of this notation with the annotation of operators with explicit profiles, which is also denoted by the colon ':' in CASL, is unlikely since the profiles of observers of cotypes always contain the function arrow.)

Example 23 For the case where there is only one type parameter, the

semantics of $[f]\phi$ is illustrated in Example 16 — 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 the following.

sorts S, V
then free %**modal {**
types $Sum ::= left(S) \mid right(V)$
$Set ::= \{\} \mid ins(Sum, Set)$
forall $x1, x2 : Sum; s : Set$
- $ins(x1, ins(x2, s)) = ins(x2, ins(x1, s))$
- $ins(x1, ins(x1, s)) = ins(x1, s)$

}

(I.e. *Set* is specified as the finite powerset of $S + V$.) Suppose that we have observers

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

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 v satisfies $out = 0$ for each element $right(v)$ 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 $s1 = left(s)$ of $next(j, s)$. The formula $[next(i)]false$ is ambiguous and thus has to be disambiguated as explained in Remark 22: $[next(i) : S]false$ holds in a state $s : S$ iff $next(i)$ does not contain an element of the form $left(s1)$, and $[next(i) : V]false$ holds iff $next(i)$ does not contain an element of the form $right(v)$.

Example 24 Consider the specification of non-repetitive non-deterministic automata in Figure 16. Here, we express that no input letter i may occur all the time, that is, when the letter i's is input non-stop ($\langle next(i)* \rangle$), the automaton will eventually get stuck ($[next(i)]\, false$).

Remark 25 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 sorts and 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 sorts and operations are not only monomorphic, but also do not impose extra conditions on model morphisms, so that hiding them induces an equivalence of model categories. In fact, for precisely this reason, hiding the auxiliary symbols

```
spec NonRepetitiveNonDeterministicAutomata =
  sort In
  sort State
  then free %modal{
      type Set ::= {} | {__}(State) | __ ∪ __(Set; Set)
      op __ ∪ __ : Set × Set → Set,
              assoc, comm, idem, unit {}  }
  then cotype State ::= (next : In → Set)
  • ∀i : In . ⟨next(i)∗⟩[next(i)] false
end
```

Fig. 16. Specification of non-repetitive non-deterministic automata using modalities for structured observations.

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 with associated datatype functor T, and let ϕ be a modal formula of type U, already translated into a function $phi : S \to Bool$ for a sort $Bool$ of booleans. Reusing the notation of Definition 15, we can write $W = \bar{T}_{X,i}S$. Moreover, one can specify the type $TB := \bar{T}_{X,i}Bool$ by repeating the associated **free** %**modal** block with S replaced by $Bool$ and W replaced by TB. As laid out in Example 16, one then recursively defines functions $tphi$ and $ttrue$ representing the functions $\bar{T}_{X,i}\phi, \bar{T}_{X,i}\top : W \to TB$. A truth function $boxphi : S \to bool$ which encodes satisfaction of $[f(r_1, \ldots, r_n)]\phi$ in a state $s : S$ is defined by

- $boxphi(s) = true \Leftrightarrow tphi(f(r_1, \ldots, r_n, s)) = ttrue(f(r_1, \ldots, r_n, s))$

The translation of the ambient modal logic formula is then continued using $boxphi$. At the outermost level, validity of a modal formula ϕ, translated into a function $phi : S \to Bool$, is translated into the formula

- $\forall s : S \bullet phi(s) = true$

Finally, the auxiliary sorts TB and all function symbols introduced along the way are 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 [27], which is not immediately suitable for use in a specification language due to the presence of infinitary conjunction and the complex nature of its

modal operator). The syntax chosen here is largely in the spirit of [18,23] in that modalities are indexed by observer terms. The syntax of [19], inherited from [49], 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 [19], this choice does not affect expressivity at the level of state formulae. (The syntax of [19] 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 formulae.) 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, [19] is unique (along with CoCasl, of course) in admitting several non-observable sorts. Iterative modalities as in CoCasl are otherwise found only in CCSL.

The main novel feature of the modal logic introduced here is the generality of the datatypes admitted as observations, i.e. in the terminology of [36] the range of datatypes from which we generically extract modal operators. At this point, CCSL and the logic of [19] are incomparable in terms of generality: CCSL covers only absolutely free datatypes, while [19] admits, besides simple and parametrized observations, only (finite or infinite) power sets (under the heading of Kripke polynomial functors). Our notion of datatype includes both these cases, which are in fact even non-cancellative.

9 An institution for modal CoCasl

We now describe how modal formulae are incorporated into an extended institution for CoCasl. We have shown that the modal logic can be regarded as syntactical sugar over the remaining language. However, for some purposes it is necessary to retain the modal formulae 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 heterogeneous Casl [28,32], but most notably in order to integrate modal CoCasl into the institution-independent framework of Casl with regard to structured and architectural specifications.

We recall that defining an institution amounts to defining notions of signature, signature morphism, sentence, sentence translation, model, model reduction, and satisfaction of sentences in models. Satisfaction is subject to the *satisfaction condition* stating that a model M satisfies the translation of a sentence ϕ along a signature morphism σ iff the reduct of M along σ satisfies ϕ. See [14] for detailed definitions.

For definiteness, we record the following.

Definition 26 The *(plain)* CoCasl *institution* is identical to the Casl institution [34], except that it has two additional types of sentences, namely, *cogeneratedness constraints* and *cofreeness constraints* as explained in Sections 4 and 5.

From this institution, which does not record enough information on cotype definitions in order to define the required notion of modal formula, we distinguish the modal CoCasl institution, defined as follows.

Definition 27 An *extended* CoCasl *signature* consists of a Casl signature (cf. Sect. 1, see also [34] for details) and the following additional data:

- a transitive relation *sees* and a partial equivalence relation *sibling* on the set of sorts.
- A set $\mathcal{G}$ of *distinguished presentations*, where a *presentation* is a pair consisting of a sort generation constraint $(\mathcal{S}, F)$ and a finite set E of equations between terms of sorts in $\mathcal{S}$. The distinguished presentations are required to have pairwise disjoint sort sets; i.e. $((\mathcal{S}_1, F_1), E_1), ((\mathcal{S}_2, F_2), E_2) \in \mathcal{G}$ implies $\mathcal{S}_1 \cap \mathcal{S}_2 = \emptyset$. A sort S for which there exists $((\mathcal{S}, F), E) \in \mathcal{G}$, necessarily unique, such that $S \in \mathcal{S}$ is called a *derived datatype*, and in this case, $(\mathcal{S}, F)$ is called the *presentation of* S.

A sort is called a *cotype* if it is in the domain of *sibling* (in signatures generated by CoCasl specifications, the cotypes in this sense will indeed be the sorts coming from **cotype** declarations).

Signature morphisms σ are required to

- preserve the *sibling* and *sees* relations
- reflect derived datatypes; i.e. if $\sigma(S)$ is a derived datatype, then so is S
- be injective on derived datatypes
- preserve distinguished presentations; i.e. if $((\mathcal{S}, F), E)$ is a distinguished presentation, then so is $\sigma((\mathcal{S}, F), E) = ((\sigma[\mathcal{S}], \sigma[F]), \sigma[E])$ (since derived datatypes are mapped injectively, there is no need to annotate presentations with a signature morphism as in the case of sort generation constraints in general).

The set of *sentences* associated to an extended CoCasl signature Σ consists of the sentences associated to the underlying Casl signature in the CoCasl institution and, additionally, the *modal formulae* over Σ. The syntax of modal formulae is defined as in Sections 7 and 8, with flexible constants and modal operators determined as follows. We say that a modal operator has *type* $U \to S$ if it applies to modal formulae of type U, yielding modal formulae of type S. Each function symbol $f : R_1 \times \cdots \times R_n \times S \to W$, where S is a cotype that sees the R_i, gives rise to

- a parametrized flexible constant $f : R_1 \times \cdots \times R_n \to W$ if S *sees* W;
- simple modal operators $[f(r_1, \ldots, r_n)]$, $\langle f(r_1, \ldots, r_n)\rangle$, $[f(r_1, \ldots, r_n)*]$, and $\langle f(r_1, \ldots, r_n)*\rangle$ of type $W \to S$, parametrized over $r_i : R_i$, $i = 1, \ldots, n$, if W is a sibling of S;
- structured modal operators $[f(r_1, \ldots, r_n)]$ etc. of type $U \to S$ if W is a derived datatype with presentation $((\mathcal{S}, F), E)$ and U is a sibling of S that appears as an argument type of a constructor in F and is not contained in $\mathcal{S}$ (if there are several possible U, then there are several modal operators with corresponding different types).

Finally, modal operators can be combined e.g. in the form $[\{f_1(r_{11}, \ldots, r_{1n_1}), \ldots, f_l(r_{k1}, \ldots, r_{kn_k})\}]$, and parameters may be omitted as explained in Sections 7 and 8.

Given a modal formula ϕ, the translation of ϕ along a signature morphism σ is defined by recursion over the formula structure. Here, modal operators and flexible constants associated to a function symbol f are translated into the corresponding entities for $\sigma(f)$, which exist by preservation of the *sees* and *sibling* relations.

The notions of *model* and *model reduction* for extended CoCasl signatures is the same as in the plain CoCasl institution, up to the following additional condition on models: if W is a derived datatype with presentation $((\mathcal{S}, F), E)$, then the interpretation of W must be an initial algebra for this presentation, i.e. isomorphic to the corresponding sort in the term algebra of constructor terms w.r.t. the set F of constructors modulo the equations in E.

The *satisfaction relation* for modal formulae is defined as described in Section 7. The semantics of structured modal operators is determined by the presentations of derived datatypes according to Definition 15. Explicitly, let ϕ be a modal formula of type U, and let $f : R_1 \times \cdots \times R_n \times S \to W$, where S sees the R_i, W is a derived datatype with presentation $((\mathcal{S}, F), E)$, and U is a sibling of S that appears as an argument type of a constructor in F and is not contained in $\mathcal{S}$. The presentation $((\mathcal{S}, F), E)$ induces an initial algebra functor $T : \mathbf{Set}^n \to \mathbf{Set}$, where n is the number of sorts outside $\mathcal{S}$. Since $U \notin \mathcal{S}$, we obtain a functor $\bar{T} : \mathbf{Set} \to \mathbf{Set}$ by fixing all arguments of the functor T except the one for the sort U to their interpretations in the present model; note in particular $\bar{T}U = W$. Then $[f(r_1, \ldots, r_n)]\phi$ holds in a state $s : S$ iff $\bar{T}[\![\phi]\!](f(r_1, \ldots, r_n, s)) = \bar{T}\top(f(r_1, \ldots, r_n, s))$, where $[\![\phi]\!] : U \to \mathit{Bool}$ is the interpretation of ϕ and the term $f(r_1, \ldots, r_n, s)$ is abused to denote its own interpretation.

Proposition and Definition 28 *These data define an institution, the* modal CoCasl institution.

PROOF. We have to establish the satisfaction condition for modal formulae. This is done by a straightforward induction on the formula structure; the step involving applications of structured modal operators goes through thanks to preservation of presentations of derived datatypes. □

The additional data for extended CoCasl signatures show up in the semantics of CoCasl constructs as follows. The *sees* and *sibling* relations are determined purely by the **cotype** declarations. W.r.t. these relations, a **cotype** declaration of cotypes $S_1, \ldots, S_n$ has the following effects.

- The *sees* relation is extended by relations S_i *sees* U for all sorts U in the local environment such that S_i has a selector with result type U or a parameter (i.e. argument other than S_i) of type U, *except* when U is either one of the S_j or when one of the S_j is a derived datatype and U appears as an argument type of a constructor in its presentation. The transitive closure of the resulting relation is the new *sees* relation.
- The cotypes $S_1, \ldots, S_n$ are declared to be *sibling*s. The partial equivalence generated by the resulting relation is the new *sibling* relation.

In particular, redeclaring a cotype may increase the number of sorts it *sees* as well as the number of its *sibling*s. The derived datatypes of an extended CoCasl signature, on the other hand, are determined by the **free** %**modal** blocks: by the format enforced for such a block, it determines a presentation, which is added to the set of distinguished presentations of the extended signature. The rules for **free** %**modal** blocks are such that presentations are then indeed disjoint. Note that the restriction that signature morphisms be injective on derived datatypes does impose an additional condition on renamings, which however does not seem to be an actual limitation in a practical sense.

The intuition behind this is that the local environment is regarded as observable for purposes of observing a given cotype; i.e. the *sees* relation gives rise to a *local* notion of observability. In particular, it is possible to instantiate observable parameter sorts in a parametrized specification such as the specification List [**sort** *Elem*] of lists of entries of type *Elem* with, $\langle f(r_1, \ldots, r_n)\rangle$, $[f(r_1, \ldots, r_n)*]$, and $\langle f(r_1, \ldots, r_n)*\rangle$ a non-observable argument sort to obtain e.g. lists of streams.

Remark 29 The above definitions do not prevent the user from causing a certain amount of havoc by abusive renaming or redeclaration of symbols. E.g. it is possible to declare a cotype S that *sees* a sort T in its local environment, and then redeclare T as a cotype that *sees* S and hence itself. Then an observer $f : T \to T$ gives rise both to a flexible constant f and to a modal operator $[f]$, despite the proviso in the semantics of cotypes which excludes siblings from the *sees* relation. While this would certainly be regarded as a specification error

— the sorts S and T would more appropriately be defined within a single cotype declaration — we have preferred delegating this and further problems to a forthcoming set of methodological guidelines rather than overburden the definition of the signature category with further formal restrictions.

Remark 30 The definition, and hence the implementation, of the modal CoCASL institution can be simplified for the sublanguage of CoCASL that uses only structured modal operators for non-cancellative datatypes (this restriction can be statically checked). In this sublanguage, it is, thanks to Theorem 18, unnecessary to record the equations defining a derived datatype — we need only its constructors, which then enable us to define the semantics of modal operators by means of Conditions (i) or (ii) of Theorem 18.

The following fact is of importance w.r.t. instantiations of parametrized specifications in the institution-independent framework of CASL structured specifications.

Proposition 31 *The category of extended* CoCASL *signatures has pushouts.*

PROOF. Let $\sigma : \Sigma_1 \to \Sigma_2$ and $\tau : \Sigma_1 \to \Sigma_3$ be morphisms of extended CoCASL signatures, and let

$$\begin{array}{ccc} \Sigma_1 & \xrightarrow{\sigma} & \Sigma_2 \\ {\scriptstyle\tau}\downarrow & & \downarrow{\scriptstyle\bar{\tau}} \\ \Sigma_3 & \xrightarrow[\bar{\sigma}]{} & \Sigma_4 \end{array}$$

be the pushout of the underlying CASL signatures (which we denote by Σ_1 etc. as well). The resulting CASL signature Σ_4 is made into an extended CoCASL signature by taking the *sees* and *sibling* relations to be the smallest transitive relation and partial equivalence relation, respectively, that make $\bar{\sigma}$ and $\bar{\tau}$ morphisms of extended CoCASL signatures; the presentations in Σ_4 are defined as the images of the presentations in Σ_2 and in Σ_3 under $\bar{\sigma}$ and $\bar{\tau}$, respectively. This defines Σ_4 as a pushout of extended CoCASL signatures. The only points that require actual verification are those that relate to the treatment of presentations and derived datatypes.

To begin, we have to check that $\bar{\sigma}$ and $\bar{\tau}$ are injective on derived datatypes and reflect derived datatypes. This is seen as follows: note that the sort component of the above pushout is a pushout in **Set**. Since σ and τ reflect derived datatypes, this pushout can be decomposed as the disjoint sum of two pushouts in **Set**, one for the derived datatypes and one for the other sorts. It

follows that $\bar{\tau}$ and $\bar{\sigma}$ reflect derived datatypes; moreover, since injective maps are stable under pushouts in **Set**, $\bar{\tau}$ and $\bar{\sigma}$ are injective on derived sorts.

It remains to prove disjointness of presentations in Σ_4. Let $(\mathcal{S}_1, F_1)$ and $(\mathcal{S}_2, F_2)$ be distinct presentations in Σ_4, and assume that there exists $S \in \mathcal{S}_1 \cap \mathcal{S}_2$. By injectivity of $\bar{\sigma}$ and $\bar{\tau}$ on derived datatypes, we can assume that $(\mathcal{S}_1, F_1)$ and $(\mathcal{S}_2, F_2)$ come from presentations in Σ_2 and in Σ_3, respectively; thus, S must come from a derived datatype in Σ_1. But then, by preservation and distinctness of presentations, both $(\mathcal{S}_1, F_1)$ and $(\mathcal{S}_2, F_2)$ must come from the presentation of s in Σ_1, in contradiction to their distinctness in Σ_4. □

Finally, the observation that modal formulae can be regarded as syntactical sugar now becomes the formal statement that the modal CoCasl institution can be encoded in the institution of structured specifications over the plain CoCasl institution via an institution comorphism [16]. We recall that a comorphism $I \to J$ between institutions I and J consists of a translation Φ of I-signatures into J-signatures, a translation of Σ-sentences into $\Phi\Sigma$-sentences, and a reduction of $\Phi\Sigma$-models to Σ-models, subject to various naturality conditions and a satisfaction condition. In the case of CoCasl, Φ takes an extended CoCasl signature to its underlying Casl (i.e. plain CoCasl) signature, model reduction does nothing, and sentence translation is the encoding of modal logic formulae by structured specifications described in Sections 7 and 8.

Of course, CoCasl specifications containing modal formulae need to be interpreted in the modal CoCasl institution, while for CoCasl specifications without modal formulae, it does not really matter which of the two institutions is chosen. As soon as CoCasl is fully integrated into the heterogeneous tool set [29,28], it will be possibly to move back and forth between the two institutions using the comorphism explained above and the (trivial) embedding of the plain CoCasl institution in the modal CoCasl institution.

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 [55]: free models exist for specifications with universally quantified Horn axioms. Part of a corresponding coalgebraic result has been obtained in [24]. In summary, the following is known.

(i) Cofree coalgebras exist for bounded functors Σ on **Set**, more generally for accessible functors on locally presentable categories [4,40]. 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 [23,24].

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 32 *Let* $\mathbf{C}$ *be a category equipped with a factorization system* $(\mathbf{E}, \mathcal{M})$ *for sinks [1], 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):* Cf. e.g. [21].

(ii): The closure condition implies that $\mathbf{B}$ is $\mathcal{M}$-coreflective in $\mathbf{CoAlg}(\Sigma)$ [1]. The coreflection of the final Σ-coalgebra is final in $\mathbf{B}$. □

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 [4] (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.

In the following result, we make use of the notion model-theoretic conservativity as employed in CASL (annotation %**cons**). A specification Sp_2 extending a specification Sp_1 (e.g. $Sp_2 = Sp_1$ **then** Sp_3) is *model-theoretically conservative* or, briefly, *model-expansive* over Sp_1 if every model M of Sp_1 can be *expanded* to a model M' of Sp_2, i.e. there exists an Sp_2-model M' such that the reduct of M' to the signature of Sp_1 is M. Consistency in the sense of model existence can be subsumed under this notion: a specification Sp is consistent iff it is a model-expansive extension of the empty specification (which has a unique model).

Theorem 33 *Let Sp be a specification of the form*

$$Sp_1 \textbf{ then cofree } Sp_2 \ .$$

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

- *declarations of (new)* non-observable *sorts;*
- *a* **free %modal** *block declaring a number of derived datatypes (cf. Section 8);*
- *a redeclaration (cf. Section 1) of the non-observable sorts as cotypes, with only observable sorts as parameters, and*
- *modal logic formulae for the non-observable sorts, using modal operators only for derived datatypes that are either non-cancellative or have only one non-observable type parameter,*

in the given order. Then Sp is model-expansive over Sp_1, provided that Sp_1 **then** *Sp_2 is model-expansive 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 32, it now suffices to show that subcategories determined by modal logic formulae are closed under componentwise jointly surjective sinks. Thus, let ϕ be a modal formula of type S, a non-observable sort. We show that

$$s \models \phi \quad \Longleftrightarrow \quad h(s) \models \phi$$

for each coalgebra homomorphism $h : A \to B$ in $\mathbf{B}$ and each s in the carrier A_S of S in A.

To this end, we decode diamond modalities by negation and box modalities, box modalities with omitted parameters by universal quantification over observable sorts, combined box modalities $[\{\dots\}]$ by conjunction, and iterated box modalities by infinitary conjunction. We then proceed by induction over the structure of ϕ. 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 boolean operators including infinitary conjunction and disjunction and for quantifications over observable sorts are trivial (recall that observable sorts are fixed under h due to the fact that attention is restricted to fibres).

The remaining case is $\bar{\phi} = [f(r_1, \dots, r_n)]\psi$, where ψ is a modal formula of type U, $f : R_1 \times \cdots \times R_n \times S \to W$ is an observer, $R_1, \dots, R_n$ are observable sorts, and W is a derived datatype with U as one of its type parameters (it suffices to treat the case of structured modal operators, since these semantically subsume simple modal operators; cf. Section 8), with associated datatype functor $T : \mathbf{Set}^m \to \mathbf{Set}$, where m is the number of non-observable type parameters of W (recall that all other sorts are fixed under h). Let p be the interpretation of ψ as a boolean-valued function on the carrier of U in the target B of h; by induction, the interpretation of ψ in the source A of h is $p \circ h_U$. Moreover, let $t_A : A_S \to A_U$ and $t_B : B_S \to B_U$ be the interpretations of f as a function between carrier sets in A and B, respectively, for fixed (observable) values of $r_1, \dots, r_n$. Since h is a Σ-coalgebra homomorphism and the datatype functor T is part of a polynomial decomposition of Σ, we have $t_B \circ h_S = T\bar{h} \circ t_A$, where $\bar{h}$ is the family of maps h_V with V non-observable. By the definition of the semantics of $[f(r_1, \dots, r_n)]$, we have to show

$$[\![t_A(s)]\!]\,(p \circ h_U) \iff [\![(T\bar{h})(t_A(s))]\!]\,p \qquad (*)$$

for each $s \in A_S$. If W has only one non-observable type parameter, i.e. $m = 1$, then $\bar{h} = (h_U)$, so that the claim follows from Proposition 17. Otherwise, T is, by assumption, non-cancellative. In this case, the claim follows from Theorem 18: let α be a constructor term representing $t_A(s)$ in the sense laid out in Section 8. By Condition (ii) of Theorem 18, the left hand side of $(*)$ says that $p \circ h(u) = \top$ for all $u \in A_U$ that appear in α, while the right hand side states that $p(u) = \top$ for all $u \in B_U$ that occur in the term obtained from α by substituting all variables v of non-observable sort V by $h_V(v)$; by non-cancellativity of T, the two statements are equivalent. □

Remark 34 The last proviso in the theorem is needed because the given modal formulae may be inconsistent in the sense that they are false for all states. In the full category of coalgebras over $\mathbf{Set}^n$, such formulae 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 CO-

CASL, carrier sets are explicitly required to be non-empty, so that the model class of, say, the modal formula *false* is indeed empty.

Remark 35 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 datatypes that are either non-cancellative or have only one type parameter.

Remark 36 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 33:

Example 37 Restricting non-observable sorts by equational axioms, rather than modal formulae, 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 33 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 $\mathcal{P}$ 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 $\mathcal{P}$ — 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 [6]. Almost all of the underlying concepts of process algebra can be found in the languages CCS and CSP: 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. Typical system building operations include sequential composition, parallel composition, and nondeterministic choice. For details, refer to [25] (CCS) and [17,48] (CSP).

The description of the specifications is organized as follows. In Section 11.1, we define the syntax of both process algebras using (algebraic) datatypes. In Section 11.2, we specify the operational semantics, and in Section 11.3, we defined the various standard process equivalences. We lay out a denotational semantics in terms of the final non-deterministic automaton for the finitely branching (i.e. weakly guarded) fragment of CCS in Section 11.4. Section 11.5 presents an example of a buffer specification that uses modal logic. Finally, related approaches to process algebra specification are discussed in Section 11.6.

11.1 Elements of process algebra syntax

Process algebras observe reactive systems by means of communications. While CSP uses an unstructured set of communications, 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 communications. At the semantical level, this alphabet Σ is extended by an *invisible action* τ 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; the intuition behind this is that parallel execution of n and $\bar{n}$ represents an internal communication of the system, regarded as invisible to the outside. Names and co-names together form the set of *labels*. Adding to this set the *silent action* τ results in the set of *actions*.

spec ACTION =
  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*
end

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 $Fun[Label] ::= (eval : Label \to Label)$
then
sort $Relabelling = \{\ f : Fun[Label]\ .$
$\forall\ l{:}Label\ .\ eval(bar(l), f) = bar(eval(l, f))$
$\wedge\quad \forall\ l, k{:}Label\ .\ (eval(l, f) = eval(k, f) \Rightarrow k = l$
$\wedge\quad \forall\ l{:}Label\ .\ \exists\ k{:}Label\ .\ l = eval(k, f))\ \}$

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 $Relation[Sigma] ::= (holds :\ Sigma \times Sigma \to Bool)$

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 17 and 18. 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 17 the sorts *AgentVariable*, *AgentConstant*, *Act*, *Set*[*Label*] and *Relabelling*) and the constructor operations.

In [25], 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, we restrict the language to finite nondeterministic choice — this is expressive enough to retain full computational power (cf. [25], p. 135). Similarly, we limit the internal choice operator of CSP to the finite case.

free type

$$
\begin{array}{lll}
\mathit{AgentExpression} ::= & \textbf{sort}\ \mathit{AgentVariable} & \\
| & \textbf{sort}\ \mathit{AgentConstant} & \\
| & \mathit{0} & \%\% \text{ inactive agent} \\
| & \_\_\rightarrow\_\_(\mathit{Act};\ \mathit{AgentExpression}) & \%\% \text{ Prefix} \\
| & \_\_+\_\_(\mathit{AgentExpression};\ \mathit{AgentExpression}) & \%\% \text{ Sum} \\
| & \_\_||\_\_(\mathit{AgentExpression};\ \mathit{AgentExpression}) & \%\% \text{ Parall.} \\
| & \_\_-\_\_(\mathit{AgentExpression};\ \mathit{Set}[\mathit{Label}]) & \%\% \text{ Restriction} \\
| & \mathit{rel}(\mathit{AgentExpression};\ \mathit{Relabelling}) & \%\% \text{ Relabelling} \\
| & \mathit{fix}(\mathit{AgentVariable};\ \mathit{AgentExpression}) & \%\% \text{ Recursion}
\end{array}
$$

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

free type

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

Fig. 18. The CSP Syntax as a free type.

While CCS uses environments that bind agent constants to agent expressions, the version of CSP in [48], 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 construction of the process syntax as a free type, it is straightforward to define substitution as a recursive function, as carried out for the case of CCS in Figure 19.

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 be

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

Fig. 19. Inductive definition of substitution in CCS.

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. $\_\_ - \_\_ \to \_\_ : AgentExpression * Act * AgentExpression$, holds on a minimal subset. Figures 20 and 21 show (part of) the operational semantics of CCS and CSP, respectively. (*Omega* is a theoretical construct introduced in [48] in order to deal with termination in the operational semantics). 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 20 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 proceeds 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 semantics 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 terms of bisimulation.

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

Fig. 20. Part of the CCS Semantics.

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. 21. Semantics of CSP External and Internal Choice.

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 $Process ::= ::= semBisim(AgentExpression)$
pred $\_\_ - \_\_ \rightarrow \_\_ : Process \times Act \times Process$
$\forall\ E,\ E'{:}AgentExpression;\ a{:}Act$

• $E - a \to E' \Leftrightarrow semBisim(E)\ -\ a\ -> semBisim(E')$

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

pred $\_\_\sim\_\_ : AgentExpression \times AgentExpression$
$\forall$ E, F:*AgentExpression*
• $E \sim F \Leftrightarrow semBisim(E) = semBisim(F)$

The cogeneratedness constraint guarantees full abstractness via a coinduction axiom, which in this case amounts to stating that strong bisimulation is equality, cf. [52,25]. Since strong bisimulation is a congruence, it is consistent to shift the operations of the process syntax from the level of agent expressions to the level of processes. 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] (see also [50] for a general coalgebraic treatment of weak bisimulation in a similar spirit):

Theorem 38 (Weak vs. Strong Bisimulation) *Let $\mathcal{T}_i = (S_i, s_i, Act, \to_i)$ be transition systems over Act with state sets S_i, initial states $s_i \in S_i$ and transition relations $\to_i$, $i = 1, 2$. Then*

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

where $\approx$ denotes weak bisimulation [25], and $\sim$ stands for strong bisimulation.

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

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

PROOF. To prove '⇒', 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 $r \xrightarrow{\alpha}_w r'$ with $\alpha \neq \tau$ in a transition system $W(\mathcal{T})$ corresponds to a derivation $r(\xrightarrow{\tau})^* r_1 \xrightarrow{\alpha} r_2(\xrightarrow{\tau})^* r'$ 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 $WProcess\ WProcess ::= semWeakBisim(AgentExpression)$
free type $Act' ::=$ **sort** $Label \mid epsilon$
pred $\_\_-\_\_\rightarrow\_\_ : WProcess \times Act' \times WProcess$
$\forall$ *E*, *E'*:*AgentExpression*; *l*:*Label*
• $semWeakBisim(E) - l \rightarrow semWeakBisim(E') \Leftrightarrow$
$\exists$ *E1*, *E2*:*AgentExpression* . $E \longrightarrow E1 \wedge E1 - l \rightarrow E2 \wedge E2 \longrightarrow E'$
• $semWeakBisim(E) - epsilon \rightarrow semWeakBisim(E') \Leftrightarrow E \longrightarrow E'$

Having this available, we can apply Theorem 38, i.e. strong bisimulation is defined as equality on *WProcess* and transferred to the CCS *AgentExpression*:

cogenerated { **sort** *WProcess*
pred $\_\_-\_\_\rightarrow\_\_ : WProcess \times Act' \times WProcess$ }
pred $\_\_\approx\_\_ : AgentExpression \times AgentExpression$
• $\forall$ *E*, *F*:*AgentExpression* • $E \approx F \Leftrightarrow semWeakBisim(E) = semWeakBisim(F)$

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 [25], 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 account:

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

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 39 (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) bisimilar transition systems are trace equivalent and (ii) that the above described powerset construction yields a trace equivalent transition system.

"$\Rightarrow$": Let $R \subseteq 2^{S_1} \times 2^{S_2}$ 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 $\_\_ - \_\_ \rightarrow \_\_ : TraceProcess \times Sigma \times TraceProcess$
$\forall\ X,\ Y{:}NonEmptyPS[Process];\ a{:}Sigma$
• $tr(X) - a \rightarrow tr(Y) \Leftrightarrow$
$(\forall\ Q{:}Process\ .\ eps(Q, Y) = True \Leftrightarrow$
$\exists\ P{:}Process\ .\ eps(P, X) = True \wedge P - a \rightarrow Q)$

According to Theorem 39, 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.

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

Note that trace equivalence is not only of coalgebraic nature. The extraction of CSP process traces, as described in [48], 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 A Coalgebraic Denotational Semantics for CCS

There are several possibilities to apply coalgebras and coinduction to the formalization of processes. In the previous sections, we have defined the syntax and operational semantics of the process calculi CSP and CCS in an *inductive* way, by a free type. Only for the definition of the bisimulation relation as a greatest relation we have used a *coalgebraic* construct, namely a cogeneration constraint. In this section, we will follow another path, which offers a more *direct coalgebraic* view at processes. We look for the appropriate system of observers, for the specific class of processes and later define, by coinduction, suitable operations on the final coalgebra. This approach makes clear that the semantics does not depend on these operations, since coinductive definitions of operations on the final coalgebra do not change the semantics of the process type, like inductive definitions of operations on free types do not change the semantics of the corresponding data type. In the sequel, we will illustrate this approach by defining operations on the final nondeterministic automaton that are inspired by the operations of CCS processes.

Remark 40 The exposition below is restricted to the finitely branching case, with all limitations this entails. The case with unlimited branching is more complicated, since the power set functor does not admit a final coalgebra; how-

ever, an analogous treatment is still possible using a loose semantical domain that admits the interpretation of all CCS expressions.

Recall that handshake synchronization of parallel processes is modeled by the structure of the set of actions as given by specification ACTION from Sect. 11.1.1: one has an operation *bar* on the sort *Act* of actions; two actions x, y are called complementary if $bar(x) = y$ or equivalently $bar(y) = x$.

First we define the process type of non–deterministic automata, which is represented by the final non–deterministic automaton, specified similarly as in Figure 13.

spec SET [**sort** *Elem*] =
  free {**type** $Set[Elem] ::= \{\}$
    $\mid \{\_\_\}(Elem)$
    $\mid \_\_\cup\_\_(Set[Elem]; Set[Elem])$
      op $\_\_\cup\_\_ : Set[Elem] \times Set[Elem] \to Set[Elem]$,
        assoc, *comm*, *idem*, *unit* $\{\}$
  }
end

spec EXTSET [**sort** *Elem*] =
  SET [**sort** *Elem*]
then
  pred $\_\_eps\_\_ : Elem \times Set[Elem]$
  op $\_\_intersection\_\_ : Set[Elem] \times Set[Elem] \to Set[Elem]$
  $\forall$ *a*, *b*: *Elem*; *s1*, *s2*: $Set[Elem]$
  • $\neg$ *a eps* $\{\}$
  • *a eps* { *b* } $\Leftrightarrow$ $a = b$
  • *a eps* (*s1* $\cup$ *s2*) $\Leftrightarrow$ *a eps s1* $\vee$ *a eps s2*
  • *a eps* (*s1 intersection s2*) $\Leftrightarrow$ *a eps s1* $\wedge$ *a eps s2*
end

spec FINALNONDETERMINISTICAUTOMATON [ACTION] =
  cofree {SET [**sort** *State*]
  then
    cotype $State ::= (next : Act \to Set[State])$
  }
end

Now we can easily define a specific process

spec ZERO =
  FINALNONDETERMINISTICAUTOMATON [ACTION]

then

op $zero : State$
var a: Act
• $next(a, zero) = \{\}$

end

This introduces the name *zero* for the process which cannot perform any action.

The next operation is the *action prefixing*:

spec ActionPrefixing =
FinalNonDeterministicAutomaton [Action]

then

op $\_\_\rightarrow\_\_ : Act \times State \rightarrow State$
$\forall\ x, y: Act; s: State$
• $next(x, y \rightarrow s) = \{\ s\ \}\ when\ x = y\ else\ \{\}$

end

Using rules which define the operational semantics in process algebras like CCS, the behavior of a process constructed by action prefixing would be defined by the axiom $a.s \stackrel{a}{\rightarrow} s$, as done e.g. in Section 11.2.

For the definition of binary operations on the final non–deterministic automaton, we need a data type of binary relations, equipped in particular with an image function for binary operations.

spec BinRel [**sort** S] =
Set [**sort** S] **and** Pair [**sort** S] [**sort** S] **and** Set [**sort** $Pair[S,S]$]

then

ops $\_\_*\_\_ : S \times Set[S] \rightarrow Set[Pair[S,S]]$;
$\_\_*\_\_ : Set[S] \times Set[S] \rightarrow Set[Pair[S,S]]$;
$\_\_*\_\_ : Set[S] \times S \rightarrow Set[Pair[S,S]]$
$\forall\ a: S; b: S; U, X: Set[S]; Y, Z: Set[S]$
• $a * \{\} = \{\}$
• $a * \{\ b\ \} = \{\ pair(a, b)\ \}$
• $a * (Y \cup Z) = a * Y \cup a * Z$
• $\{\} * Y = \{\}$
• $\{\ a\ \} * Y = a * Y$
• $(U \cup X) * Y = U * Y \cup X * Y$
• $X * b = X * \{\ b\ \}$

end

spec BinRelFun [**sort** S; **op** $\_\_+\_\_ : S \times S \rightarrow S$] =
BinRel [**sort** S]

then

op $power[\_\_+\_\_] : Set[Pair[S,S]] \to Set[S]$
$\forall\ s1, s2 : S;\ set1, set2 : Set[Pair[S,S]]$
• $power[\_\_+\_\_](\{\}) = \{\}$
• $power[\_\_+\_\_](\{\ pair(s1, s2)\ \}) = \{\ s1 + s2\ \}$
• $power[\_\_+\_\_](set1 \cup set2) = power[\_\_+\_\_](set1) \cup power[\_\_+\_\_](set2)$
end

The *summation* of two processes represents nondeterministic choice:

spec SUMMATION =
ZERO
and
BINRELFUN [**sort** *State*; **op** $\_\_+\_\_ : State \times State \to State$]
then
$\forall\ a : Act;\ s1, s2 : State$
• $next(a, s1 + s2) =$
$power[\_\_+\_\_](zero * next(a, s2) \cup next(a, s1) * zero)$
end

Using this coinductive definition, we can prove properties of the defined operation. We will prove that the constant automaton *zero* is a unit for summation, and that summation is commutative.

Let be $h_0 = \lambda s.(s + zero)$. The equality $s = s + zero$ is proved coinductively if we can show that h_0 is a homomorphism; i.e. we have to prove $power[h_0](next(a, s)) = next(a, s + zero)$ [1]. In order to improve readability, the proof is conducted in standard notation ($\mathcal{P}(+)$ in place of $power[\_\_ + \_\_]$ etc.):

$$
\begin{aligned}
& next(a, s + zero) \\
= \ & \mathcal{P}(+)((zero \times next(a, zero)) \cup (next(a, s) \times zero)) \\
= \ & \mathcal{P}(+)((zero \times \{\}) \cup (next(a, s) \times zero))) \\
= \ & \mathcal{P}(+)((\{\} \cup (next(a, s) \times zero))) \\
= \ & \mathcal{P}(+)(next(a, s) \times zero) \\
= \ & \{s' + zero \mid s' \in next(a, s)\ \} \\
= \ & \mathcal{P}(h_0)(next(a, s))
\end{aligned}
$$

In the same way one can prove the equality $s = zero + s$.

[1] Note that *power* is applied to a unary function here; adding this case is straightforward.

An immediate consequence of this two equations is

$$next(s_1 + s_2) = next(s_1) \cup next(s_2)$$

This equality together with the commutativity of set union implies the commutativity of summation of nondeterministic automata.

In case the set of actions is finite, one can define the parallel composition of processes with handshake synchronization represented by the silent action *tau* (cf. Section 11.1.1).

spec FinAct =
  ExtSet [**sort** *Label*]
then
  op *actions* : *Set*[*Label*]
  ∀ *l*: *Label* • *l eps actions*
end

spec Composition =
  FinAct
and
  FinalNonDeterministicAutomaton [Action]
and
  BinRelFun [**sort** *State*; **op** __||__ : *State* × *State* → *State*]
and
  ExtSet [**sort** *State*] **and** ExtSet [**sort** *Act*]
then
  ops __||__ : *State* × *State* → *State*;
    h : *State* × *State* × *Set*[*Label*] → *Set*[*State*]
  vars *l*: *Label*; *s1*, *s2*: *State*; *set1*, *set2*: *Set*[*Label*]
  • *h*(*s1*, *s2*, {}) = {}
  • *h*(*s1*, *s2*, { *l* }) = *next*(*l*, *s1*) *intersection* *next*(*bar*(*l*), *s2*)
  • *h*(*s1*, *s2*, *set1* ∪ *set2*) = *h*(*s1*, *s2*, *set1*) ∪ *h*(*s1*, *s2*, *set2*)
  • *next*(*l*, *s1* || *s2*) = *power*[__||__](*next*(*l*, *s1*) ∗ *s2* ∪ *s1* ∗ *next*(*l*, *s2*))
  • *next*(*tau*, *s1* || *s2*) =
    power[__||__]((*next*(*tau*, *s1*) ∗ *s2* ∪ *s1* ∗ *next*(*tau*, *s2*)) ∪
      h(*s1*, *s2*, *actions*))
end

Hiding and *relabelling* of actions is defined in a straightforward manner:

spec Hiding =
  FinalNonDeterministicAutomaton [Action]
and
  ActionRelabelling

then

op $\_\_-\_\_ : State \times Set[Label] \to State$

$\forall\ l$: *Label*; s: *State*; L: $Set[Label]$

• $next(l, s - L) = \{\}$ *when* l *isIn* L = *True else* $next(l, s)$

• $next(tau, s - L) = next(tau, s)$

end

spec RELABELLING =

FINALNONDETERMINISTICAUTOMATON [ACTION]

and

EXTSET [**sort** *Act*] **and** ACTIONRELABELLING

then

op $rel : State \times Relabelling \to State$

$\forall\ l$: *Label*; s: *State*; f: *Relabelling*

• $next(l, rel(s, f)) = next(eval(f, l), s)$

• $next(tau, rel(s, f)) = next(tau, s)$

end

Using the specifications introduced so far, we now give a denotational semantics to the CCS language introduced in Sect. 11.1.3. The only feature not realized so far is fixpoint equations. Indeed, it is not entirely clear how to treat these directly at a semantic level (one would need some analogue of cpos, ensuring the existence of least fixed points). Hence, we introduce fixed points only during the definition of the denotational semantics, and restrict them to weakly guarded expressions (i.e. expressions in which each variable is directly or indirectly prefixed by an action). It is well-known that for this restricted case, fixed points exist and are unique [26] and even finitely branching. The latter ensures that the fixed points can be expressed in our framework based on finite sets.

spec CCS_COALGEBRAIC_SEMANTICS =

FINACT **and** ZERO **and** ACTIONPREFIXING **and** SUMMATION **and** COMPOSITION **and** HIDING **and** RELABELLING **and** CCS

and

MAP [**sort** *AgentVariable*] [**sort** *State*]

with $Map[AgentVariable,State] \mapsto Env$

then

preds *prefixedVars*, *isWeaklyGuarded* : *AgentExpression*

op $\{\{\_\_\}\}\_\_ : AgentExpression \times Env \to ? State$

$\forall\ a$: *Act*; *env*: *Env*; C: *AgentConstant*;

E, *E1*, *E2*: *AgentExpression*; X: *AgentVariable*; *set*: $Set[Label]$;

f: *Relabelling*

• $prefixedVars(C)$

• $\neg\ prefixedVars(X)$

- $prefixedVars(0)$
- $prefixedVars(a \rightarrow E)$
- $prefixedVars(E1 + E2) \Leftrightarrow$
  $prefixedVars(E1) \wedge prefixedVars(E2)$
- $prefixedVars(E1 \mid\mid E2) \Leftrightarrow$
  $prefixedVars(E1) \wedge prefixedVars(E2)$
- $prefixedVars(E1 - set) \Leftrightarrow prefixedVars(E1)$
- $prefixedVars(rel(E, f)) \Leftrightarrow prefixedVars(E)$
- $prefixedVars(fix(X, E)) \Leftrightarrow prefixedVars(E)$
- $isWeaklyGuarded(C)$
- $isWeaklyGuarded(X)$
- $isWeaklyGuarded(0)$
- $isWeaklyGuarded(a \rightarrow E) \Leftrightarrow isWeaklyGuarded(E)$
- $isWeaklyGuarded(E1 + E2) \Leftrightarrow$
  $isWeaklyGuarded(E1) \wedge isWeaklyGuarded(E2)$
- $isWeaklyGuarded(E1 \mid\mid E2) \Leftrightarrow$
  $isWeaklyGuarded(E1) \wedge isWeaklyGuarded(E2)$
- $isWeaklyGuarded(E1 - set) \Leftrightarrow isWeaklyGuarded(E1)$
- $isWeaklyGuarded(rel(E, f)) \Leftrightarrow isWeaklyGuarded(E)$
- $isWeaklyGuarded(fix(X, E)) \Leftrightarrow$
  $isWeaklyGuarded(E) \wedge prefixedVars(E)$
- $\{\{ X \}\}\ env = lookup(X, env)$
- $\{\{ C \}\}\ env = \{\{ definitionOf(C) \}\}\ env$
- $\{\{ 0 \}\}\ env = zero$
- $\{\{ a \rightarrow E \}\}\ env = a \rightarrow \{\{ E \}\}\ env$
- $\{\{ E1 + E2 \}\}\ env = \{\{ E1 \}\}\ env + \{\{ E2 \}\}\ env$
- $\{\{ E1 \mid\mid E2 \}\}\ env = \{\{ E1 \}\}\ env \mid\mid \{\{ E2 \}\}\ env$
- $\{\{ E - set \}\}\ env = \{\{ E \}\}\ env - set$
- $\{\{ rel(E, f) \}\}\ env = rel(\{\{ E \}\}\ env, f)$
- $isWeaklyGuarded(E) \Rightarrow$
  $\{\{ fix(X, E) \}\}\ env =$
  $\{\{ E \}\}\ env\ [\ \{\{ fix(X, E) \}\}\ env\ /\ X\ ]$
- $\neg\ isWeaklyGuarded(E) \Rightarrow \neg\ def\ \{\{ fix(X, E) \}\}\ env$

then %implies

$\forall\ a$: Act; env: Env; E, $E1$, $E2$: $AgentExpression$;
X: $AgentVariable$

- $isWeaklyGuarded(E) \Rightarrow def\ \{\{ fix(X, E) \}\}\ env$
- $isWeaklyGuarded(E1) \wedge isWeaklyGuarded(E2) \Rightarrow$
  $(E1 - a \rightarrow E2 \Leftrightarrow \{\{ E2 \}\}\ env\ eps\ next(a, \{\{ E1 \}\}\ env))$

end

We finally express as implied statements that fixed points are unique and that the denotational semantics coincides with the operational one.

In order to illustrate the use of modal logic formulae in connection with the formalization of CCS in CoCasl, consider the following specification (adapted from [25]) of two very simple two-element and three-element buffers (specified by the agent constants B and C, respectively). The buffer can input or output symbols. For simplicity, the symbols are just bits here.

spec BUFFER =
CCS
then
free type $Elem ::= 0 \mid 1$
free type $Name ::= input(Elem) \mid output(Elem)$
free type
$AgentConstant ::= B \mid B\_0 \mid B\_1 \mid C \mid C\_0 \mid C\_1 \mid C\_00 \mid C\_01 \mid C\_10 \mid C\_11$
- $definitionOf(B) = (input(0) \rightarrow B\_0) + (input(1) \rightarrow B\_1)$
- $definitionOf(B\_0) = bar(output(0)) \rightarrow B$
- $definitionOf(B\_1) = bar(output(1)) \rightarrow B$
- $definitionOf(C) = (input(0) \rightarrow C\_0) + (input(1) \rightarrow C\_1)$
- $definitionOf(C\_0) = (input(0) \rightarrow C\_00) + (input(1) \rightarrow C\_10) + (bar(output(0)) \rightarrow C)$
- $definitionOf(C\_1) = (input(0) \rightarrow C\_01) + (input(1) \rightarrow C\_11) + (bar(output(1)) \rightarrow C)$
- $definitionOf(C\_00) = bar(output(0)) \rightarrow C\_0$
- $definitionOf(C\_01) = bar(output(1)) \rightarrow C\_0$
- $definitionOf(C\_10) = bar(output(0)) \rightarrow C\_1$
- $definitionOf(C\_11) = bar(output(1)) \rightarrow C\_1$

end

A desirable property of both buffers is that every input symbol is eventually output. This means more precisely that if a process performs $input(a)$, it must in a finite number of steps reach a state where it can perform $bar(output(a))$. This is an invariant required to hold at any point of time during the execution of the process. Formally

$$\forall\ a\text{: } Elem \bullet [next(input(a))] \langle next* \rangle \langle next(bar(output(a))) \rangle\ true$$

The two buffers B and C are distinguished by their capacity. We can express

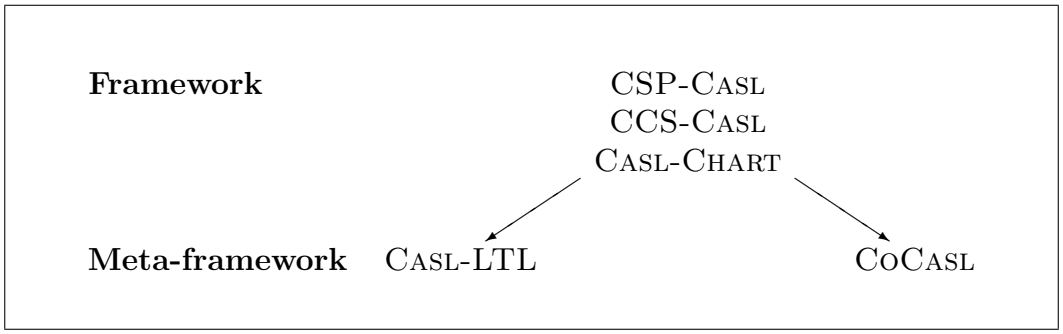

Fig. 22. Relationship between COCASL and other reactive CASL extensions

that a buffer has capacity of at least two as follows: from any state, we can reach — in a finite number of steps — a state where it is possible to perform two arbitrary *input* operations consecutively. Formally:

$$\forall\ a,\ b\colon \mathit{Elem} \bullet \langle \mathit{next}* \rangle\ \langle \mathit{next}(\mathit{input}(a)) \rangle\ \langle \mathit{next}(\mathit{input}(b)) \rangle\ \ \mathit{true}$$

Capacity of at least n is expressed similarly, and capacity smaller or equal to n is expressed by the negation of capacity of at least $n-1$. The conjunction of both specifies capacity exactly n.

11.6 Related Approaches in Modelling Process Algebra

We have presented a general scheme for specifying models of concurrency: a clear distinction 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 22 for a small selection. Our definition of COCASL differs from CASL extensions like CSP-CASL [47], CCS-CASL [53,54] or CASL-CHART [42]. 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 [41] 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 [41] 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 datatypes such as finite sets or lists of states. We have given a sufficient criterion for the existence of cofree models for specifications using initial datatypes and modal formulae. Moreover, we have described an institution for modal CoCASL that incorporates a local notion of observability; this institution can be encoded in structured specifications in plain CoCASL.

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. Moreover, we have given a coalgebraic denotational semantics to the finitely branching fragment of CCS, and we have expressed the relation between the operational semantics and the denotational semantics 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 [51] has fewer datatypes available, while hidden algebra such as in BOBJ [46] and reachable-observable algebra such as in COL [7] do not support cofree types. If, for example, streams are not specified as the final (=cofree) model, then there are stream models which do not contain all corecursively definable functions (like the flipping of streams), so that corecursive definitions 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 [39] support infinite data structures 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 data structure expressible in these languages are present in the models of a specification. The Haskell semantics for lazy data structures (at least for the

non-left-→-recursive case) indeed comprises *all* infinite trees, i.e. is captured exactly by a behaviour algebra.

The institution of *Constructor-based Observational Logic (COL)* [7] combines reachability induced by constructors with observational equality induced by observers. CoCasl does not directly support observational equality or bisimilarity, but full abstractness ('bisimilarity is equality') 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 [7] 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 [51], developed in close cooperation with the LOOP project [57], is based on the observation of [43] 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* [56] (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; cofree models usually fail to exist in the presence of binary observers.

Future work will concentrate in particular on the development of tools for CoCasl. The *heterogeneous tool set* [29,28] already provides a parser and static analysis for Casl and CoCasl basic and structured specifications. Concerning proof support, it is planned to extend the coding of Casl into Isabelle/HOL [30] 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 [37] will provide useful tactics for the encoding of CoCasl into Isabelle/HOL. All the specifications shown in this paper are available under

`http://www.cofi.info/Libraries`.

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. Erwin R. Catesbeiana deserves thanks for help with logical issues revolving around falsum elimination. We also thank Hendrik Tews for useful discussions on the relation between CCSL and CoCasl.

References

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

[2] Jiří Adámek and Václav Koubek, *On the greatest fixed point of a set functor*, Theoretical Computer Science **150** (1995), no. 1, 57–75.

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

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

[5] H. Baumeister, M. Cerioli, A. Haxthausen, T. Mossakowski, P. D. Mosses, D. Sannella, and A. Tarlecki, Casl *semantics*, Part III of [34].

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

[7] 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.

[8] M. Bidoit and P. D. Mosses, Casl *user manual*, LNCS, vol. 2900, Springer, 2004.

[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 Language Design Group, CASL *summary*, Part I of [34], edited by B. Krieg-Brückner and P.D. Mosses.

[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] J. Goguen and G. Rosu, *Institution morphisms*, Formal aspects of computing **13** (2002), 274–307.

[17] C.A.R. Hoare, *Communicating sequential processes*, Prentice Hall, 1985.

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

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

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

[21] A. Kurz, *Logics for coalgebras and applications to computer science*, Ph.D. thesis, Universität München, 2000.

[22] ———, *Coalgebras and modal logic*, Course Notes for the European Summer School on Logic, Language and Information (CD-ROM), University of Helsinki, 2001.

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

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

[25] R. Milner, *Communication and concurrency*, Prentice Hall, 1989.

[26] R. Milner, *Operational and algebraic semantics of concurrent processes*, Handbook of Theoretical Computer Science (J. van Leewen, ed.), vol. B: Formal Models and Semantics, MIT Press, 1990, pp. 1201–1242.

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

[28] T. Mossakowski, *The heterogeneous tool set*, Available at `www.tzi.de/cofi/hets`, University of Bremen.

[29] ———, *Implementing logics: from genericity to heterogeneity*, Tech. report, University of Bremen.

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

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

[32] _____, *Foundations of heterogeneous specification*, Recent Trends in Algebraic Development Techniques, 16th International Workshop, WADT 2002, Frauenchiemsee, Germany, 2002, Revised Selected Papers (M. Wirsing, D. Pattinson, and R. Hennicker, eds.), 2755, 2003, pp. 359–375.

[33] 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, Elsevier, 2003.

[34] P. D. Mosses (ed.), CASL *reference manual*, LNCS, vol. 2960, Springer, 2004.

[35] D. Pattinson, *Expressivity results in the modal logic of coalgebras*, Ph.D. thesis, University of Munich, 2001.

[36] _____, *Semantical principles in the modal logic of coalgebras*, Symposium on Theoretical Aspects of Computer Science, LNCS, vol. 2010, Springer, 2001, pp. 514–526.

[37] _____, *Expressive logics for coalgebras via terminal sequence induction*, Tech. report, LMU München, 2002.

[38] D. Pavlovic and V. Pratt, *On coalgebra of real numbers*, Coalgebraic Methods in Computer Science, Electron. Notes Theoret. Comput. Sci., vol. 19, Elsevier, 1999.

[39] S. Peyton-Jones (ed.), *Haskell 98 language and libraries — the revised report*, Cambridge, 2003, also: J. Funct. Programming **13** (2003).

[40] J. Power and H. Watanabe, *An axiomatics for categories of coalgebras*, Coalgebraic Methods in Computer Science, Electron. Notes Theoret. Comput. Sci,, vol. 11, Elsevier, 2000.

[41] 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.

[42] 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.

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

[44] _____, *A uniform model theory for the specification of data and process types*, Recent Developments in Algebraic Development Techniques, 14th International Workshop (WADT 99), LNCS, vol. 1827, Springer, 2000, pp. 348–365.

[45] 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, vol. 2755, Springer, 2003, pp. 376–392.

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

[47] M. Roggenbach, CSP-Casl — *a new integration of process algebra and algebraic specification*, AMAST Workshop on Algebraic Methods in Language Processing (AMiLP 2003), TWLT, vol. 21, University of Twente, 2003.

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

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

[50] J. Rothe, *Behavioural equivalences for coalgebras*, Ph.D. thesis, University of Dresden, 2003.

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

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

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

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

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

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

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