CoCasl at Work — Modelling Process Algebra

Till Mossakowski [1,4] Markus Roggenbach [2,5] Lutz Schröder [3,6]

BISS, Department of Computer Science
Universität Bremen, Germany

Abstract

CoCasl [11], a recently developed coalgebraic extension of the algebraic specification language Casl [2], allows for modelling systems in terms of inductive datatypes as well as of co-inductive process types. Here, we demonstrate how to specify process algebras, namely CCS [10] and CSP [8,17], within such an algebraic-coalgebraic framework. It turns out that CoCasl can deal with the fundamental concepts of process algebra in a natural way: The type system of communications, the syntax of processes and their structural operational semantics fit well in the algebraic world of Casl, while the additional coalgebraic constructs of CoCasl cover the various process equivalences (bisimulation, weak bisimulation, observational congruence, and trace equivalence) and provide fully abstract semantic domains. CoCasl hence becomes a meta-framework for studying the semantics and proof theory of reactive systems.

Among the various frameworks for the description and modelling of reactive systems, process algebra plays a prominent role. It has proven to be suitable at the level of requirement specification, for formal refinement proofs as well as for writing design specifications. Almost all of the underlying concepts of process algebra can be found in the languages CCS [10] and CSP [8,17]: a type system on the communications; synchronous as well as asynchronous communication; operational semantics; and also various notions of process equivalence like strong and weak bisimulation, observation congruence, and trace equivalence. Thus, when proposing a new framework which aims at the specification of reactive systems in general, it is worthwile to study if these process algebras and their semantic concepts are covered.

[1] `till@informatik.uni-bremen.de`

[2] `roba@informatik.uni-bremen.de`

[3] `lschrode@informatik.uni-bremen.de`

[4] Research supported by the DFG project Multiple (KR 1191/5-2)

[5] Research supported by the DFG project COOFL (Qi 1/4-2)

[6] Research supported by the DFG project HasCASL (KR 1191/7-1)

This is a preliminary version. The final version will be published in
Electronic Notes in Theoretical Computer Science
URL: `www.elsevier.nl/locate/entcs`

CoCasl [11], a recently defined coalgebraic extension of the algebraic specification language Casl [2], allows for modelling systems in terms of inductive datatypes as well as of co-inductive process types. Here, the language Casl provides a many sorted first order logic with partial functions and subsorting, and also features sort generation constraints for generated and free datatypes. Furtheron, Casl allows for initial semantics in terms of a structured free construct. CoCasl extends this language by a basic co-datatype construct, cogeneratedness constraints, and structured cofree specifications; moreover, coalgebraic modal logic is introduced as syntactical sugar.

The idea behind the design of CoCasl is to obtain a fruitful synergy of algebraic and coalgebraic system modelling. Specifying the process algebras CCS and CSP provides an extensive case study in this language. Thus, besides beeing a proof of concept for CoCasl, these specifications demonstrate how to integrate different system aspects within one framework. Casl as the algebraic sub-language of CoCasl will deal with the syntactic aspects and the operational semantics of these process algebras, while the equivalence relations on processes like bisimulation, which have been shown to be of coalgebraic nature [3,19,16], will be modelled with the coalgebraic constructs provided by CoCasl. The full specifications are available at [12].

The paper is organized as follows: Section 1 gives an overview of CoCasl. Then the different aspects of the process algebras CCS and CSP are specified in parallel, separating between syntax and semantics: Section 2 describes process algebra syntax in terms of Casl free datatypes; Section 3 provides the structural operational semantics. Finally, various notions of process equivalences are specified in Section 4.

1 An overview of CoCasl

The specification language Casl (*Common Algebraic Specification Language*) has been designed by CoFI, the international *Common Framework Initiative for Algebraic Specification and Development* [7]. For the language definition cf. [2,4]; a full formal semantics is laid out in [1]. Here, we recall some basic language features as needed for understanding the specifications below.

Casl is separated into various levels, including a level of *basic specifications* and a level of *structured specifications*. Basic specifications essentially list signature items and axioms in an unstructured way, thus determining a category of first order models. Structured specifications serve to combine such basic specifications into larger specifications in a hierarchical and modular fashion. The structuring operations are independent of the underlying logic and hence apply to language extensions such as CoCasl in the same way as to Casl. At the level of basic specifications, one can declare sorts (keyword **sort**), operations (keyword **op**), and predicates (keyword **pred**) with given input and result sorts. Sorts may be declared to be in a *subsorting* relation; if s is a subsort of t, then terms of type s may be used wherever terms of type t are expected. Subsorts may also be *defined* in the form $s = \{x : t \bullet \phi\}$, with the effect that s consists of all elements of t that satisfy ϕ. Operations may be declared to be partial by using a modified function arrow $\rightarrow ?$. Using the symbols thus declared, one may then write axioms in first order logic. Moreover, one can specify datatypes (keyword **type**), given in terms of alternatives consisting of data constructors and, optionally, selectors, which may be declared to be **generated** or **free**. Generatedness amounts to an implicit higher order induction axiom and intuitively states that all elements of the datatypes are reachable by constructor terms ('no junk'); freeness additionally requires that all these constructor terms are distinct ('no confusion'). Thus, free datatypes are really free algebras for the signature given by the constructors.

At the level of structured specifications, one has features such as unions of specifications, parametrized named specifications, extensions of specifications (keyword **then**), and free specifications **free {** ... **}**. The latter are particularly important in this context; the semantics of a specification

$$SP_1 \textbf{ then free } \{SP_2\}$$

is given by those models of SP_1 **then** SP_2 that are free over their reduct to the signature of SP_1.

CoCasl extends Casl both at the level of basic specifications and at the level of structured specifications by features that support coalgebraic specification, so that the combined power of algebra and co-algebra becomes available. For details beyond the description provided below, including examples for the usage of all CoCasl features, see [11]. Figure 1 provides an overview of the dualities between Casl and CoCasl concepts and constructs.

To begin, the datatype constructs are complemented in CoCasl by support for cogenerated and final datatypes. *Cotypes* (keyword **cotype**) dualize datatypes; they are given by declaring selectors and, optionally, constructors. In addition to the material generated by a **type** definition, such cotype definitions give rise to axioms that guarantee that the models of the cotype are coalgebras for the corresponding polynomial set-functor (with alternatives corresponding to sums). Cotypes may be declared to be **cogenerated** or **cofree**. These constraints produce implicit axioms which guarantee that models of the cotype are fully abstract coalgebras (i.e. subcoalgebras of the absolutely final coalgebra) or (absolutely) final coalgebras, respectively. Cogeneratedness amounts to an implicit coinduction axiom, while cofreeness (finality) additionally requires realization of all possible behaviours; both types of axioms require higher order quantifiers (so that the constraints are more than just syntactical sugar).

Both generatedness and cogeneratedness constraints may enclose any number of signature items (i.e. declarations of sorts, operations, or predicates), so that one may write

cogenerated { ... **}**

where the grouping brackets may contain any basic specification without ax-

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

Fig. 1. Summary of dualities between Casl and CoCasl.

ioms. As in the case of simple datatype or cotype definitions, these constraints give rise to implicit induction or coinduction axioms, respectively.

At the level of structured specifications, CoCasl provides a **cofree {**. . .**}** construct, dual to Casl's structured **free**. The models of a specification

$$SP_1 \textbf{ then cofree } \{SP_2\}$$

are the *fibre-final* models of SP_1 **then** SP_2, i.e. those models that are final objects in their fibre w.r.t. the reduction to the signature of SP_1. Here, the *fibre* of a model M consists of all models that have the same reduct to SP_1 as M. Thus, the semantics of **cofree** is not fully dual to the semantics of **free**; cf. [11] for a discussion of this point and an example that illustrates why the more liberal semantics of **cofree** is necessary (for the latter point, see also Section 2.2 below).

In order to specify properties of coalgebras, CoCasl provides syntactic sugar for a modal logic in the style of [9]. In [11], a rather general existence theorem for final models is provided which guarantees the consistency of a large range of **cofree** specifications.

2 Elements of process algebra syntax

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

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

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

2.1 Type system of communications: Casl *subsorting*

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

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

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

CCS processes communicate 'names'. Each name n has a 'co-name' $\bar{n}$, where the function $\bar{\_}$ (bar) is involutive. Names and co-names together form the set of 'labels'. Adding to this set the 'silent action' τ results in the set of 'actions'. Again, Casl captures this simple type system:

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

Note that we have the subsort relations *Name* < *Label* < *Act*. The operation *bar* is introduced twice: as constructor from *Name* into *Label* and as function on *Label*. The involution property on 'names' is obtained by implicit overloading axioms.

2.2 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 of booleans, and given a previously declared type of labels *Label*,

$$\begin{array}{lll}\mathbf{cofree}\ \{ & \mathbf{sort} & \mathit{Set}[\mathit{Label}] \\ & \mathbf{op} & \_\_\mathit{isIn}\_\_ : \mathit{Label} \times \mathit{Set}[\mathit{Label}] \to \mathit{Boolean}\ \}\end{array}$$

specifies $\mathit{Set}[\mathit{Label}]$ as the powerset of the set of labels. Note here the importance of the fact that the semantics of **cofree** is given in terms of *fibre*-cofree models: finality of a model M can only be expected w.r.t models that interpret the sort *Label* in the same way as M. Concerning CoCasl syntax, note that $\mathit{Set}[\mathit{Label}]$ 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. $Fun[\mathit{Label}]$ or $\mathit{Relation}[Sigma]$.

Similarly, the function spaces needed for relabelling are provided by means of a structured cofree (by exploiting their couniversal property). Since only bijections that commute with the 'bar' operation are admissible as CCS relabellings, the actual type of relabellings is defined as a subtype:

$$\begin{array}{llll} & \mathbf{cofree}\ \{ & \mathbf{sort} & \mathit{Fun}[\mathit{Label}] \\ & & \mathbf{op} & \mathit{eval} : \mathit{Fun}[\mathit{Label}] \times \mathit{Label} \to \mathit{Label}\ \} \\ \mathbf{then} & & & \\ & \mathbf{sort} & \multicolumn{2}{l}{\mathit{Relabelling} = \{\ f : \mathit{Fun}[\mathit{Label}]\ .} \\ & & & \forall\ l{:}\mathit{Label}\ .\ \ \mathit{eval}(f,\ \mathit{bar}(l)) = \mathit{bar}(\mathit{eval}(f,\ l)) \\ & & \wedge & \forall\ l,\ k{:}\mathit{Label}\ .\ \ (\mathit{eval}(f,\ l) = \mathit{eval}(f,\ k) \Rightarrow k = l \\ & & \wedge & \forall\ l{:}\mathit{Label}\ .\ \ \exists\ k{:}\mathit{Label}\ .\ \ l = \mathit{eval}(f,\ k))\ \}\end{array}$$

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

Sets of communications are also needed for the hiding and generalized parallel operators of CSP. Furtheron, the relational renamings of CSP require a type of binary relations on the communication alphabet Σ :

$$\begin{array}{lll}\mathbf{cofree}\ \{ & \mathbf{sort} & \mathit{Relation}[\mathit{Sigma}] \\ & \mathbf{op} & \mathit{holds} : \mathit{Relation}[\mathit{Sigma}] \times \mathit{Sigma} \times \mathit{Sigma} \to \mathit{Boolean}\ \}\end{array}$$

2.3 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 2 and 3. 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 2 the sorts $AgentVariable$, $AgentConstant$, *Act*, $\mathit{Set}[\mathit{Label}]$ and *Relabelling*) and the constructor operations.

In [10], Milner introduces CCS as a *class* of agent expressions. The crucial point is that the summation operator (non-deterministic choice) involves

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

Fig. 2. The CCS Syntax as a free type.

```
free type
Process ::= Skip
          | Stop
          | Omega
          | sort ProcessVar
          | __→__(Sigma; Process)                          %% Prefix
          | __seq__(Process; Process)                      %% Sequential Composition
          | __[]__(Process; Process)                       %% External Choice
          | __ |~|__(Process; Process)                     %% Internal Choice
          | __−__(Process; Set[Sigma])                     %% Hiding
          | __[[__]](Process; Relation[Sigma])             %% Relational Renaming
          | __[__]__(Process; Set[Sigma]; Process)         %% Generalized Parallel
          | mu(ProcessVar; Process)                        %% Recursion
```

Fig. 3. The CSP Syntax as a free type.

arbitrary index sets. This is beyond the scope of Casl and CoCasl, as the specified models interpret sorts by carrier *sets*. Therefore, and also in order to capture bisimulation via a final object in a suitably chosen category, we restrict the language to finite nondeterminism — this is expressive enough to retain full computational power (cf. [10], p. 135).

While CCS uses environments that bind agent constants to agent expressions, the version of CSP in [17], which we specify here, is restricted to a core language without environments. The full language including e.g. the various CSP parallel operators can be recaptured as a definitional extension.

Thanks to the free type construct of the process syntax it is straightforward to introduce substitution operators, as carried out for the case of CCS in Figure 4.

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

Fig. 4. Inductive definition of substitution

3 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*, resp. The transitions are defined to be the smallest relation satisfying a certain set of inference rules. The corresponding Casl construct is a structured free, which has the effect that the introduced predicate, e.g. $\mathit{pred}\_\_ - \_\_ \rightarrow \_\_ : \mathit{AgentExpression} * \mathit{Act} * \mathit{AgentExpression}$, holds on a minimal subset. Figures 5 and 6 show (part of) the operational semantics of CCS and CSP, respectively. Both only use positive Horn clauses, hence the specifications are consistent (note that due to the definition of *Act* as free type, axioms with premise $\neg a = \mathit{tau}$ can be replaced by two axioms with equational premise). Figure 5 includes the CCS inference rule for recursion, which makes use of the substitution operator described above. CSP models recursion in the same way. Note how the rules for external choice in CSP are formulated along the type system of CSP communications on the semantical level. It is interesting to observe the difference between CCS and CSP in the modelling of nondeterminism. While CCS directly procedes with an action, the CSP semantics uses an invisible action τ. This inference rule among other, similar ones, is the reason why it is necessary to carefully extract the traces of observable actions from the specified transition system. The advantage of the — at first sight complicated — transition system for CSP is that it can also be taken as the basis for working out the denotations of processes in the failures and failures/divergences semantics of CSP.

4 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** $\_\_ - \_\_ \rightarrow \_\_ : \mathit{AgentExpression} \times \mathit{Act} \times \mathit{AgentExpression}$

%% (Act):

$\forall\ a{:}\mathit{Act};\ E{:}\mathit{AgentExpression}$

$\bullet\ (a \rightarrow E) - a \rightarrow E$

%% (Sum1):

$\forall\ E,\ E',\ F{:}\mathit{AgentExpression};\ a{:}\mathit{Act}$

$\bullet\ E - a \rightarrow E' \Rightarrow (E + F) - a \rightarrow E'$

. . .

%% (Rec):

$\forall\ X{:}\mathit{AgentVariable};\ E,\ E'{:}\mathit{AgentExpression};\ a{:}\mathit{Act}$

$\bullet\ E\{\mathit{fix}(X, E)/X\} - a \rightarrow E' \Rightarrow \mathit{fix}(X, E) - a \rightarrow E'$ }

Fig. 5. Part of the CCS Semantics.

free { **pred** $\_\_ - \_\_ \rightarrow \_\_ : \mathit{Process} \times \mathit{ExtSigma} \times \mathit{Process}$

. . .

%% External Choice:

$\forall\ P,\ P',\ Q{:}\mathit{Process}$

$\bullet\ P - \mathit{tau} \rightarrow P' \Rightarrow (P[\,]\,Q) - \mathit{tau} \rightarrow (P'[\,]\,Q)$

$\forall\ P,\ Q,\ Q'{:}\mathit{Process}$

$\bullet\ Q - \mathit{tau} \rightarrow Q' \Rightarrow (P[\,]\,Q) - \mathit{tau} \rightarrow (P[\,]Q')$

$\forall\ a{:}\mathit{ExtSigma};\ P,\ P',\ Q{:}\mathit{Process}$

$\bullet\ \neg\ a = \mathit{tau} \Rightarrow P - a \rightarrow P' \Rightarrow (P[\,]\,Q) - a \rightarrow P'$

$\forall\ a{:}\mathit{ExtSigma};\ P,\ Q,\ Q'{:}\mathit{Process}$

$\bullet\ \neg\ a = \mathit{tau} \Rightarrow Q - a \rightarrow Q' \Rightarrow (P[\,]\,Q) - a \rightarrow Q'$

%% Internal Choice:

$\forall\ P,\ Q{:}\mathit{Process}$

$\bullet\ (P|\tilde{}|Q) - \mathit{tau} \rightarrow P$

$\forall\ P,\ Q{:}\mathit{Process}$

$\bullet\ (P|\tilde{}|Q) - \mathit{tau} \rightarrow Q$

. . . }

Fig. 6. Semantis of CSP External and Internal Choice.

4.1 Strong Bisimulation

Modelling strong bisimulation is straightforward. We built 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 $\mathit{Process} ::= [[\_\_]](\mathit{AgentExpression})$

pred $\_\_ - \_\_ \rightarrow \_\_ : \mathit{Process} \times \mathit{Act} \times \mathit{Process}$

$\forall$ *E*, *E'*:*AgentExpression*; *a*:*Act*
$\bullet$ $E - a \rightarrow E' \Leftrightarrow [[E]] - a \rightarrow [[E']]$

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

pred $\_\_\sim\_\_$: *AgentExpression* × *AgentExpression*
$\forall$ *E*, *F*:*AgentExpression*
$\bullet$ $E \sim F \Leftrightarrow [[\ E\ ]] = [[\ F\ ]]$

The cogeneratedness constraint guarantees full abstractness via a coinduction axiom, which in this case amounts to stating that strong bisimulation is equality, cf. [19,10]. Note that the existence of strong bisimulation is guaranteed by the results of [10]; hence, the above specification is consistent. Moreover, since strong bisimulation even is a congruence, it is also consistent to shift the operations of the process syntax from the level of agent expressions to the level of processes. Also note that there are other abstraction principles on processes like weak bisimulation discussed below that are not a congruence.

An alternative approach to characterize bisimulation on CCS is modal logic, c.f. [10]. Our choice of specifying the operational semantics as a free type keeps us from using the built-in modal logic of CoCasl. But one can easily define such a logic over a free type of formulae. Its satisfaction relation between processes and formulae can then be introduced by an inductive definition over the structure of formulae. Due to the restriction to finite nondeterminism in our specification of CCS, the modal logic also needs only finite conjunctions in order to provide an equivalent characterization of bisimulation.

4.2 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 [6]:

Theorem 4.1 (Weak and Strong Bisimulation)
Let $\mathcal{T}_i = (S_i, s_i, Act, \rightarrow_i)$ be transition systems over Act with state set S_i, initial state $s_i \in S_i$ and transition relation $\rightarrow_i$, $i = 1, 2$. Then

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

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

The operator W maps a transition system $\mathcal{T} = (S, s, Act, \rightarrow)$ to a transition system $W(\mathcal{T}) = (S, s, Act^*, \rightarrow_w)$ with $r \overset{\hat{\alpha}}{\rightarrow}_w r' :\iff r \overset{\hat{\alpha}}{\Rightarrow} r'$, where $\hat{}: Act \rightarrow Act^*$ with

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

Proof (Sketch) 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 ‘⇐’ holds because the transition systems $W(\mathcal{T}_i)$ have essentially $\overset{\hat{\alpha}}{\Rightarrow}_i$ as their transition relations. □

Thus, in order to model weak bisimulation, it is necessary to specify the operator W, i.e. the transition relation $\overset{\hat{\alpha}}{\Rightarrow}$, in CoCasl. The specification below shows how to iterate *tau*-transitions to combine this with transitions on visible actions.

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

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

$\forall\ E, E'{:}\mathit{AgentExpression};\ l{:}\mathit{Label}$
$\bullet\ [[[E]]] - l \rightarrow [[[E']]] \Leftrightarrow$
$\quad \exists\ E1, E2{:}\mathit{AgentExpression}\ .\ E \longrightarrow E1 \wedge E1 - l \rightarrow E2 \wedge E2 \longrightarrow E'$
$\bullet\ [[[E]]] - \mathit{epsilon} \rightarrow [[[E']]] \Leftrightarrow E \longrightarrow E'$

Having this available, we proceed in the same way as with strong bisimulation, i.e. we build a new transition system with a ‘copy’ of the type *AgentExpression* as a type of nodes formed via the constructor $[[[\_\_]]]$, and $\overset{\hat{\alpha}}{\Rightarrow}$ as transition relation, obtain strong bisimulation on this transition system using the cogenerated construct, and, finally, carry this relation over to the sort *AgentExpression* via a predicate $\_\_ \approx \_\_$. 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.

4.3 Observation Congruence

Having the notion of weak bisimulation available, we can express Milner’s definition of observation congruence in [10], p.153, directly in CoCasl. The crucial point of this definition is that it involves a new transition relation $\_\_ == \_\_ \Longrightarrow \_\_$, which also takes the *tau* action into accout:

pred $\_\_ == \_\_ \Longrightarrow \_\_ : \mathit{AgentExpression} \times \mathit{Act} \times \mathit{AgentExpression}$

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

4.4 Trace Equivalence on CSP

The extraction of CSP process traces, as described by [17], can be directly formulated in CoCasl. As a first step, one defines a new transition relation, which extracts the sequences of communications from the transition system obtained by the CSP operational semantics.

List [**sort** *ExtSigma* **fit sort** *Elem* $\mapsto$ *ExtSigma*]
then
List [**sort** *Process* **fit sort** *Elem* $\mapsto$ *Process*]
then
pred $\_\_ - \_\_ \longrightarrow \_\_ : Process \times List[ExtSigma] \times Process$
$\forall\ P,\ Q{:}Process;\ s{:}List[ExtSigma]$
$\bullet\ P - s \longrightarrow Q \Leftrightarrow \exists\ PL{:}List[Process]\ .$
$\qquad\qquad (\#\ s\ ) + 1 = \#\ PL$
$\qquad \wedge \quad first(PL) = P$
$\qquad \wedge \quad last(PL) = Q$
$\qquad \wedge \quad (\forall\ i{:}Nat\ .\ \ 0 < i \wedge i < (\#\ PL\ ) \Rightarrow$
$\qquad\qquad\qquad (PL!i) - (s!i) \rightarrow (PL!(i+1))\ )$

Here, a type of lists is imported by referencing a parametrized specification List[**sort** *Elem*]. The parameter **sort** *Elem* is explicitly instantiated with the sorts *ExtSigma* and *Process*, respectively. The instantiated specifications then define sorts *List*[*ExtSigma*] and *List*[*Process*], respectively, making use of an automatic renaming mechanism provided by Casl structured specifications.

CSP processes are trace equivalent iff they have the same sets of traces, i.e. they have they can take the same steps in terms of sequences of communications, where the invisible action *tau* has been removed.

pred $\_\_==\_\_ : Process \times Process$
$\forall\ P,\ Q{:}Process$
$\bullet\ P == Q \Leftrightarrow \forall\ s{:}List[ExtSigma]$
$\quad .\ \exists\ P'{:}Process\ .\ \ P == s \Longrightarrow P' \Leftrightarrow \exists\ Q'{:}Process\ .\ \ Q == s \Longrightarrow Q'$

This modelling is so intuitive that we refrained from translating trace equivalence into strong bisimulation as suggested by the following theorem,

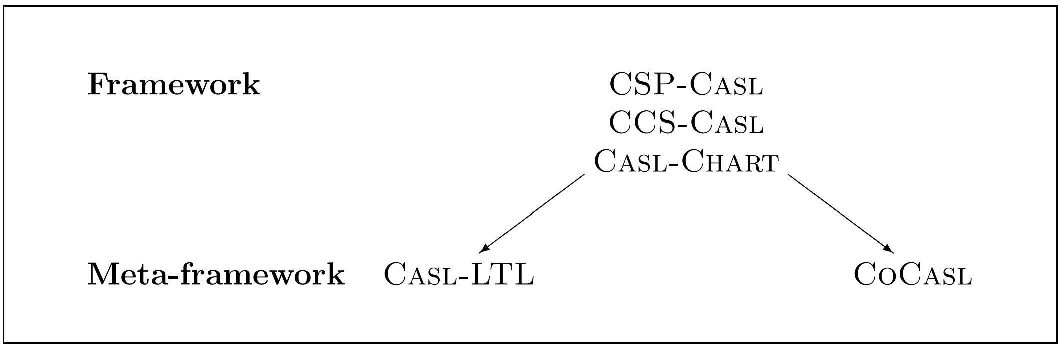

Fig. 7. Relationship between CoCasl and other reactive Casl extensions

again reformulating a result by [6]:

Theorem 4.2 (Trace Equivalence and Strong Bisimulation) *Let $\mathcal{T}_i = (S_i, s_i, \Sigma, \rightarrow_i)$ be transition systems over Σ with state set S_i, initial state $s_i \in S_i$ and transition relation $\rightarrow_i$, $i = 1, 2$. Then*

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

where $=_t$ denotes trace equivalence, and $\sim$ stands for strong bisimulation.

The operator T is the usual powerset construction, i.e. it maps a transition system $\mathcal{T} = (S, s, \Sigma, \rightarrow)$ to a transition system $P(\mathcal{T}) = (RS(S), s, \Sigma, \rightarrow_P)$, where

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

and $RS(S)$ is the least subset of $2^S \backslash \emptyset$ such that

(i) *$\{s\} \in RS(S)$ and*

(ii) *$X \in RS(S), X \xrightarrow{\alpha}_P Y$ implies $Y \in RS(X)$.*

Conclusion and related work

We have presented CoCasl specifications for the process algebras CCS and CSP including various notions of equivalences, namely strong bisimulation, weak bisimulation, observation congruence, and trace equivalence. Interestingly enough, CoCasl also provides easy constructions for higher order types like power sets, relation types, and function spaces. 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.

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

There are various proposals of reactive Casl extensions – see Figure 7 for a small selection. Our definition of CoCasl differs from Casl extensions like

CSP-Casl [15], CCS-Casl [20,21] or Casl-Chart [14] 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, proof support for CoCasl [11] can be used to prove meta-properties about these frameworks.

Casl-LTL [13] 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 [13] 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 within Casl-LTL.

Future work on CoCasl as a meta-framework will include the specification of further models of concurreny and their various notions of behavioural equivalence (see e.g. [22,5]). Moreover, we aim at direct specifications of reactive systems, in particular in the spirit of recently introduced co-algebraic paradigms for object-oriented modelling [18].

Acknowledgements

The authors wish to thank Christoph Lüth for vigorous comments, Horst Reichel for fruitful collaboration on CoCasl, Erwin R. Catesbeiana for advice on the role of consistency, and the participants of an informal workshop in Munich on December 8 and 9, 2002, for insightful discussions about observational approaches.

References

[1] Casl – *The* CoFI *Algebraic Specification Language – Semantics* (1999), Note S-9 (Documents/CASL/Semantics, version 0.96), in [7].

[2] Casl – *The* CoFI *Algebraic Specification Language – Summary, version 1.0.1* (2001), Documents/CASL/Summary, in [7].

[3] Aczel, P. and N. Mendler, *A final coalgebra theorem*, in: *Category Theory and Computer Science*, LNCS **389** (1989), pp. 357–365.

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

[5] Broy, M., *Algebraic specification of reactive systems*, in: *Algebraic Methodology and Software Technology*, LNCS **1101** (1996), pp. 487–503.

[6] Cheng, A. and M. Nielsen, *Open maps (at) work*, Technical Report RS-95-23, BRICS (1995).

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

[8] Hoare, C. A. R., "Communicating Sequential Processes," Prentice Hall, 1985.

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

[10] Milner, R., "Communication and Concurrency," Prentice Hall, 1989.

[11] Mossakowski, T., H. Reichel, M. Roggenbach and L. Schröder, *Algebraic-coalgebraic specification in* CoCasl, presented at WADT 02, submitted for publication.

[12] Mossakowski, T., M. Roggenbach and L. Schröder, *Specifications of CCS and CSP in* CoCasl, available under `http://www.pst.informatik.uni-muenchen.de/∼baumeist/CoFI/case.html`.

[13] Reggio, G., E. Astesiano and C. Choppy, Casl-LTL *— a* Casl *extension for dynamic reactive systems — summary*, Technical Report DISI-TR-99-34, Università di Genova (2000).

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

[15] Roggenbach, M., CSP-Casl *— a new integration of process algebra and algebraic specification*, presented at WADT 02, submitted for publication.

[16] Roggenbach, M. and M. Majster-Cederbaum, *Towards a unified view of bisimulation: a comparative study*, Theoret. Comput. Sci. **238** (2000), pp. 81–130.

[17] Roscoe, A., "The theory and practice of concurrency," Prentice Hall, 1998.

[18] Rothe, J., H. Tews and B. Jacobs, *The coalgebraic class specification language CCSL*, J. Universal Comput. Sci. **7** (2001), pp. 175–193.

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

[20] Salaün, G., M. Allemand and C. Attiogbé, *A formalism combining CCS and* Casl, Technical Report 00.14, University of Nantes (2001).

[21] Salaün, G., M. Allemand and C. Attiogbé, *Specification of an access control system with a formalism combining CCS and* Casl, in: *Parallel and Distributed Processing*, IEEE, 2002, pp. 211–219.

[22] Sassone, V., M. Nielsen and G. Winskel, *Models for concurrency*, Theoret. Comput. Sci. **170** (1996), pp. 297–348.