In: Recent Trends in Data Type Specification, Lecture Notes in Computer Science n. 534, 1991.

ENTITIES: AN INSTITUTION FOR DYNAMIC SYSTEMS

GIANNA REGGIO

DIPARTIMENTO DI MATEMATICA - UNIVERSITA' DI GENOVA
VIA L.B. ALBERTI 4, 16132 GENOVA, ITALY

INTRODUCTION

In this paper we introduce the entity framework (entity algebras and entity specifications) and show, also with the help of several examples, how they can be used for formally modelling and specifying dynamic systems.

Entity algebras are structures devised as formal models for both data types, processes and objects, thus allowing to give integrated abstract specifications of processes, data types and objects. They are a subclass of partial algebras with predicates, having the following features:

- some sorts correspond to dynamic elements, that we call *entities*; dynamics is represented by the possibility of performing labelled transitions;
- entities may have subcomponent entities together with usual (static) subcomponents; it is important to notice that the structure of an entity is not fixed, as the binary parallelism with interleaving of CCS or the record-like structure of usual objects, but is definable by giving appropriate operations and axioms;
- entities have *identities* in such a way that it is possible to retrieve entity subcomponents depending on their identities and to describe sharing of subcomponents.

Moreover the structure of each entity can be represented graphically in a way that makes *sharing* explicit.

Correspondingly to this model level, it is possible to define *entity specifications*; which are just the correspondent in the entity field of the usual specifications of abstract data types and can be used to formally specify dynamic systems. Using entity specifications we can abstractly describe how it is structured a dynamic system, give properties about its activity and the activity of its dynamic subcomponents; for example we can require for a certain entity that the order of the subcomponents is not relevant, that it must always terminate its activity, that its subcomponents interact in a synchronous way and so on.

However using this kind of entity specifications, called *specifications in the small*, we cannot formalize "very" abstract properties about the structure of classes of dynamic systems; for example we cannot specify the class of all "simple" dynamic systems (ie, that cannot be decomposed in several other entities interacting between them) and the class of all systems where no entity is shared between several others. To handle these cases it is possible to use a kind of ultra loose entity specifications, called *entity specifications in the large*. To enlighten the distinction between specifications in the small and in the large consider the well-known firing squad problem (see [DHJW]): for specifying one system which represents a possible solution of the problem we use an entity specification in the small; while for specifying the class of all systems representing solutions of the problem we use an entity specification in the large.

Work partially funded by COMPASS-Esprit-BRA-W.G. n 3264, by CNR-PF-Sistemi Informatici e Calcolo Parallelo and
by a grant ENEL/CRA (Milano Italy).

In this paper we consider only entity specifications in the small and give a brief hint of specifications in the large, while a full treatment of them can be found in [AR2]. We show that the specifications in the small constitute an institution; give conditions ensuring the existence of the initial model and show that has interesting properties about the dynamics and the structure (it is characterized by the minimum amount of activity and by the simpler structure).

Since entity specifications are particular algebraic specifications of abstract data types, we can extend to them various notions and results about classical specifications (eg, structured and hierarchical specifications, observational/behavioural semantics, implementation and so on); usually these extensions are adequate for coping with the particular features of entities. Here we only consider the notion of implementation (see [W]) and show that its extension to entity specifications is reasonable.

Because of their characteristics (dynamics, structure, identities) entity algebras and specifications could be used for modelling and specifying object systems; in this paper we just give an example, but do not handle this topic in a general way (for more about that see [Dragon]).

The basic algebraic institution we use in the paper is that of partial algebras with predicates; see [GM] for the institution of algebras with predicates and [BW] for the institution of partial algebras; the institution of partial algebras with predicates can be found in [AC]; note that the combination of the two institutions does not pose particular problems.

The problem of the development of an integrated algebraic framework for the specification of concurrent systems has been treated in previous papers about the SMoLCS methodology, see for example [AR], [AR1], [AGR]; the SMoLCS specifications of concurrent systems could be considered as particular entity specifications in the small. Recently it has been considered also the problem of using algebraic techniques for the formal modelling and specifications of classical object-oriented systems, see for example [BZ] and [FB]; a very recent paper of Meseguer [M] tries to offer a unifying and integrated framework for the specifications of dynamic system based on rewriting. Since the technical framework of all these approaches is quite different from the present one, it is not easy to make a comparison; some special effort should be made for investigating their relationship.

In sect. 1 and 2 we introduce entity algebras and entity specifications in the small and show their features; these sections include complete examples. Entity specifications in the large are shortly introduced in sect. 3. The proofs are omitted and will appear in [AR2].

1 ALGEBRAS FOR DYNAMIC ENTITIES

Entity algebras are particular partial algebras with predicates used for modelling dynamic concurrent and object-oriented systems; "particular" means that they always have certain sorts, operations and predicates, whose interpretations must respect some conditions. These special algebraic ingredients are used for:

- representing dynamic systems (for example, processes) and their activity;
- determining whether such systems are simple or structured, and, in the second case, allowing to describe their structure (eg, fix the number of dynamic subcomponents),
- determining the identity of such systems

and so on.

In the paper we abbreviate "signature with predicates" and "partial algebra with predicates" with "psignature" and "palgebra".

1.1 Entity signatures

Entity signatures are particular psignatures where some sorts correspond to the usual static values and some other correspond to dynamic elements (entities), their identities and their "cores". Each entity is completely determined by its identity and its core. The sorts corresponding to cores of entities, ie, whose elements are the cores of entities, are called *dynamic sorts*. In an entity signature for each dynamic sort there exist some special sorts, operations and predicates; precisely, given a dynamic sort s:

– a sort, ent(s), of entities (dynamic elements) with cores of type s (shortly entities of type s);
– a sort, ident(s), of identities of the elements of ent(s);
– an operation building the elements of ent(s)
  $\_ : \_ : \mathrm{ident}(s) \times s \to \mathrm{ent}(s)$
  which taken an identity and a core returns an entity;
– a sort, lab(s), of labels for the transitions of the elements of ent(s);
– a predicate describing the activity of the elements of ent(s) by means of labelled transitions
  $\_ \xrightarrow{\_} \_ : \mathrm{ent}(s) \times \mathrm{lab}(s) \times \mathrm{ent}(s)$.

Notice that here and in the following we use operations and predicates with mixfix syntax; the symbol "_" denotes the places of the arguments.

For giving an entity signature it is sufficient to give the basic sorts, distinguishing the ones which correspond to the cores of dynamic elements (dynamic sorts), while we can omit the special components; thus we have the following definition.

Def. 1.1

• An *entity signature* (or simply *signature*) is a 4-tuple $E\Sigma = (D, S, OP, PR)$ such that:
  – D and S are two disjoint sets
    (the basic sorts; those in D correspond to the cores of dynamic elements and are called *dynamic sorts*);
  – $(D^E \cup S, OP, PR)$ is a psignature, where
    $D^E = D \cup (\cup_{s \in D} \{ \mathrm{ent}(s), \mathrm{lab}(s), \mathrm{ident}(s) \})$
    (the sorts having form ent(s) are called *entity sorts*).

• $E\Sigma^E$ is the psignature $(D^E \cup S, OP^E, PR^E)$, where:
  – $OP^E = OP \cup \{ \_ : \_ : \mathrm{ident}(s) \times s \to \mathrm{ent}(s) \mid s \in D \}$,
  – $PR^E = PR \cup \{ \_ \xrightarrow{\_} \_ : \mathrm{ent}(s) \times \mathrm{lab}(s) \times \mathrm{ent}(s) \mid s \in D \}$. $\Box$

Notation: in the following $E\Sigma$ will be a generic entity signature (D, S, OP, PR) and $Dsorts(E\Sigma)$ will denote the set of the dynamic sorts of $E\Sigma$. Moreover we will write

sorts $s_1, \dots, s_n$
dsorts $d_1, \dots, d_m$
opns $Op_1, \dots, Op_k$
preds $Pr_1, \dots, Pr_h$

for denoting the entity signature $(\{ d_1, \dots, d_m\}, \{ s_1, \dots, s_n\}, \{ Op_1, \dots, Op_k\}, \{ Pr_1, \dots, Pr_h\})$.

1.2 Entity algebras

Given an entity signature $E\Sigma$, an $E\Sigma$-entity algebra is a particular $E\Sigma^E$-palgebra, but not all $E\Sigma^E$-palgebras are suitable as entity algebras; the interpretations of the special sorts, operations and predicates must respect some conditions. Consider indeed the entity signature ΣS

dsorts nat, stack
opns $0: \to \mathrm{nat}$
$\mathrm{Succ}: \mathrm{nat} \to \mathrm{nat}$
$\alpha, \beta, \dots : \mathrm{ident}(\mathrm{nat})$
$\mathrm{Empty}: \to \mathrm{stack}$
$\mathrm{Push}: \mathrm{ent}(\mathrm{nat}) \times \mathrm{stack} \to \mathrm{stack}$
$\mathrm{Pop}: \mathrm{ent}(\mathrm{stack}) \to \mathrm{ent}(\mathrm{stack})$
$\Gamma, \Psi, \dots : \mathrm{ident}(\mathrm{stack})$

and let AS be a ΣS^E-palgebra; for example, it may happen that in AS:

– $AS \models \alpha: 0 = \beta: 1$ and $AS \models \alpha \neq \beta$ (entities with different identities are identified);

– $AS \models \Gamma: \text{Empty} \xrightarrow{1} \Psi: \text{Empty}$ and $AS \models \Gamma \neq \Psi$
(an entity changes its identity performing a transition);

– there exists $v \in AS_{ent(nat)}$ and for all $id \in AS_{ident(nat)}$, $n \in AS_{nat}$ $v \neq id :^{AS} n$
(we cannot determine the identity and the core of an entity);

– $AS \models 0 \neq 1$ and the interpretation of Γ: Push(α: 0, Push(α: 1, Empty)) in AS is defined
(an entity has two different subentities with the same identity).

Obviously an $E\Sigma^E$-palgebra where one of the above properties holds cannot be considered an entity algebra.

To define formally which $E\Sigma^E$-palgebras are entity algebras, we need some technical definitions and for clarity we first illustrate them on an example.

Given an $E\Sigma^E$-palgebra and an element of entity sort, say e, we can find out the possible "views" of the dynamic structure of e by showing which are its dynamic subcomponents (ie, entity subcomponents) and how they are put together.

Consider a term-generated ΣS^E-palgebra BS, where sorts and operations are interpreted in the obvious way: $BS_{nat} = \mathbb{N}$, $BS_{ent(nat)} = \{ \alpha, \beta, \dots \} \times BS_{nat}$, $BS_{stack} = BS_{ent(nat)}{}^*$, $BS_{ent(stack)} = \{ \Gamma, \Psi, \dots \} \times BS_{stack}$ and so on.

Here and in the following $Symb^{ALG}$, the interpretation in an algebra ALG of Symb, either a predicate or an operation symbol, will be simply written Symb and analogously for ground terms, thus t^{ALG} will be written t.

The element

$e = \Gamma$: Push(α: 0, Push(β: 1, Empty))

represents an entity of type stack with identity Γ and core Push(α: 0, Push(β: 1, Empty)), which has two *subentities* of type nat, represented respectively by α: 0 and β: 1, organized in a stack. Thus e could be viewed as

$\Gamma : \lambda\, e_1, e_2.\ (\text{Push}(e_1, \text{Push}(e_2, \text{Empty})))(\alpha: 0, \beta: 1)$,

where the function

(*1) $\lambda\, e_1, e_2.\ \text{Push}(e_1, \text{Push}(e_2, \text{Empty})): BS_{ent(nat)} \times BS_{ent(nat)} \to BS_{stack}$

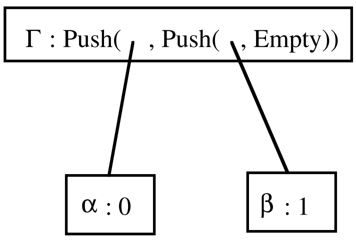
represents the way the entities α: 0, β: 1 are *composed* (ie, organized in a stack) to build up the compound entity e.

The entities α: 0, β: 1 are in some sense "simple" (ie, without subcomponents); indeed the zero-ary functions

(*2) $0, 1: \ \to BS_{nat}$

say that they are not built by composing other entities.

Graphically this way *of viewing the dynamic structure* of e can be represented by



where the term with holes "Push(, Push(, Empty))" stands for the function (*1). Notice that

$\boxed{\alpha : 0}$ $\boxed{\beta : 1}$

represent the *views of the dynamic structures* of the two entities α: 0 and β: 1 (they have no subcomponents).

The functions like (*1) and (*2), which describe how some entities are put together to get the core of compound entities, are called *entity composers*; while the structures pictorially represented above by graphs are called *entity views*; both notions are formally given in def. 1.2 and 1.3.

The entity composers such as (1*) and (2*) correspond to open terms on ΣS; however we are not restricted to term-generated algebras, since, as usual we can use a new signature obtained by adding zero-ary operations for all junk elements (recall that $v \in ALG_s$ is a *junk* element iff there does not exist a ground term t s.t. $v = t^{ALG}$).

$E\Sigma^J$ is the psignature

enrich $E\Sigma^E$ **by** $\{ Op_v\colon \to s \mid s \in Sorts(E\Sigma^E), v \in A_s \text{ junk } \}$;

and A^J the term-generated $E\Sigma^J$-palgebra obtained from A by interpreting each Op_v as v.

Def. 1.2 Let A be an $E\Sigma^E$-palgebra. The set $EC(E\Sigma, A)$ of the *entity composers* on $E\Sigma$ and A is the set of functions

$\{ \lambda\, e_1, \ldots, e_n.\ t^{A^J, [e_1/x_1, \ldots, e_n/x_n]} \mid t \in (T_{E\Sigma^J}(X))_s$
$Var(t) = \{ x_1\colon ent(s_1), \ldots, x_n\colon ent(s_n) \}$, for some $s, s_1, \ldots, s_n \in Dsorts(E\Sigma)$ and
$x_1, \ldots, x_n$ are the only subterms of t of entity sort $\}$. □

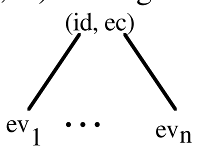
Notice that entity composers are given at a "semantic level" not at a "syntactical level", ie they are composition of operation interpretations of the algebra not terms.

Notation: given $ec \in EC(E\Sigma, A)$, if ec is a function from $A_{ent(s_1)} \times \ldots \times A_{ent(s_n)}$ into A_s, then we simply write $ec\colon ent(s_1) \times \ldots \times ent(s_n) \to s$.

Def. 1.3

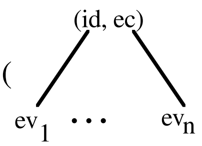
- The set $EV(E\Sigma, A)$ of the *entity views* of an $E\Sigma^E$-palgebra A is the subset of ordered trees with nodes labelled by couples consisting of an entity identity and an entity composer for entity of the same sort (such sort is the sort of the view) inductively defined as follows:

  for all $ec\colon ent(s_1) \times \ldots \times ent(s_n) \to s \in EC(E\Sigma, A)$ with $n \geq 0$, $id \in A_{ident(s)}$ and $ev_1, \ldots, ev_n \in EV(E\Sigma, A)$ having sorts respectively $s_1, \ldots, s_n$

  $\in EV(E\Sigma, A)$.

- With each entity view there is associated an element of entity sort, defined in the following way:

  V(

) $= id : ec(V(ev_1), \ldots, V(ev_n)) \in \cup_{s \in Dsorts(E\Sigma)} A_{ent(s)}$;

  if $V(ev) = e$, then we say that ev *is a view for* e. □

Notice that only the operations having sort s contribute to define the composers for entities of sort $ent(s)$, ie, the structure of the entities is determined by the operations whose result sort is the core sort; for example the operation Pop, having sort $ent(stack)$, never appears in a composer for entities of sort $ent(stack)$, ie, it does not contribute to the structure of the entities of type stack. Also note that in general an element of entity sort does not admit a unique view of its structure as it is shown in the following (see sect. 1.3).

We need also the following terminology.

Assume that A is an $E\Sigma^E$-palgebra, $e \in \cup_{s\in \mathrm{Dsorts}(E\Sigma)} A_{ent(s)}$ and $ev \in EV(E\Sigma, A)$.

- e is a *(proper) subentity* of ev iff there exists a (proper) subtree of ev which is a view for e.
- ev is *simple* iff it is a tree of depth 1.
- e *has identity* id or *is identified by* id iff $e = id\colon v$ for some v.
- ev is *sound* iff for all subentities of ev e' and e'', if e' and e'' have the same identity, then $e' = e''$.

Def. 1.4 An $E\Sigma$-*entity algebra* (or simply $E\Sigma$-*algebra)* is an $E\Sigma^E$-palgebra EA such that for all $s \in \mathrm{Dsorts}(E\Sigma)$

- $EA_{ent(s)} \subseteq EA_{ident(s)} \times EA_s{}^{(*)}$ and $(\_ : \_ : ident(s) \times s \to ent(s))^{EA} = \lambda\, id, x. <id, x>$;
- if $EA \models e \xrightarrow{1} e'$, then e and e' have the same identity;
- all elements of $EA_{ent(s)}$ have at least a view and only sound views.

The class of all $E\Sigma$-algebras is denoted by $eAlg_{E\Sigma}$. □

Notice that the last property requires that usually the interpretations in EA of some operations (eg, the entity builder operations _ : _) are partial functions.

Fact 1.5 shows that def. 1.4 is reasonable.

Fact 1.5 Let EA be an $E\Sigma$-algebra and $e \in EA_{ent(s)}$ for some s sort of $E\Sigma$,

i) e has one and only one identity;

ii) if e' and e" are two distinct subentities of a view for e, then e' and e" have different identities;

iii) if e has identity id, then a view for e cannot have a proper subentity identified by id. □

Instead the following properties show that our definition of entity algebras is not too restrictive and allows to formally describe several interesting situations.

For example, it is possible to give an $E\Sigma$-algebra EA and $e, e' \in EA_{ent(s)}$ for some s sort of $E\Sigma$ s.t.:

- $EA \models e \xrightarrow{1} e'$ and e, e' have views with a different number of subentities (dynamic creation and termination of subentities);
- e has several different views also with different subentities (different ways to put together (different) groups of entities are semantically equivalent);
- there exists a view for e where two distinct subentities have the same subentity (sharing of subentities).

1.3 An Example: Distributed Concurrent Calculi

In this section we define a particular algebra DC (and some variations) and use them to enlighten the most relevant features of entity algebras.

DC represents a simple distributed concurrent calculus. In DC we have sequential processes which evolve in an interleaving way and interact between them by handshaking communication; Nil, _ . _ and _ +_ are the combinators for expressing the sequential processes and _ || _ is the parallel combinator.

Let $D\Sigma$ be the following entity signature:

(*) More precisely $EA_{ent(s)}$ is isomorphic to a subset of $EA_{ident(s)} \times EA_s$.

dsorts proc, prog – processes and programs

opns Tau, Alpha, Beta, … $\to$ lab(proc) – process actions

$\overline{\_}$: lab(proc) $\to$ lab(proc) – complementary operation on actions

Nil: $\to$ proc

_ ▪ _ : lab(proc) $\times$ proc $\to$ proc

_ + _ : proc $\times$ proc $\to$ proc

$\alpha, \beta, \ldots : \to$ ident(proc)

_: ent(proc) $\to$ prog

_ || _ : prog $\times$ prog $\to$ prog

_ : lab(proc) $\to$ lab(prog)

$\Gamma, \Psi, \ldots : \to$ ident(prog)

DC is a term-generated DΣ-algebra such that:

- its carriers are subsets of the quotient of the ground terms on DΣ modulo the congruence generated by the equations $\overline{\overline{l}} = l$ for $l \neq$ Tau, $\overline{\text{Tau}}$ = Tau and those corresponding to the fact that + and || are commutative, associative and that Nil, id: Nil for all id are their identities;
- its operations are defined in the obvious way;
- the transition relation predicates ($\longrightarrow$ for processes and $\Longrightarrow$ for programs) are defined by the following inductive rules, where p: proc, ep: ent(proc), id: ident(proc), a: lab(proc), pg: prog, pid: ident(prog).

$$\frac{}{\text{id: a} \cdot \text{p} \xrightarrow{a} \text{id: p}} \qquad \frac{\text{id: p}_1 \xrightarrow{a} \text{id: p}_1'}{\text{id: p}_1 + \text{p}_2 \xrightarrow{a} \text{id: p}_1'}$$

$$\frac{\text{ep} \xrightarrow{a} \text{ep}'}{\text{ep} \xRightarrow{a} \text{ep}'} \qquad \frac{\text{pid: pg}_1 \xRightarrow{a} \text{pg}_1'}{\text{pid: pg}_1 \parallel \text{pg}_2 \xRightarrow{a} \text{pid: pg}_1' \parallel \text{pg}_2}$$

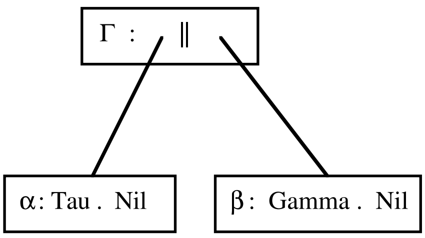
$$\frac{\text{pid: pg}_1 \xRightarrow{a} \text{pg}_1' \quad \text{pid: pg}_2 \xRightarrow{\overline{a}} \text{pg}_2'}{\text{pid: pg}_1 \parallel \text{pg}_2 \xRightarrow{\text{Tau}} \text{pid: pg}_1' \parallel \text{pg}_2'} \qquad a \neq \text{Tau}$$

Different ways of composing some entities together may be equivalent

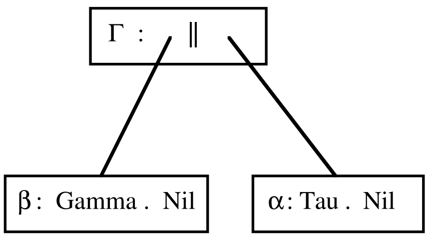
epg = Γ: (α: Tau . Nil || β: Gamma . Nil) $\in$ $DC_{ent(prog)}$

is an example of an entity whose structure may be seen in two different ways; indeed epg is also equal to Γ: (β: Gamma . Nil || α: Tau . Nil);

the two views of epg are graphically represented by



and



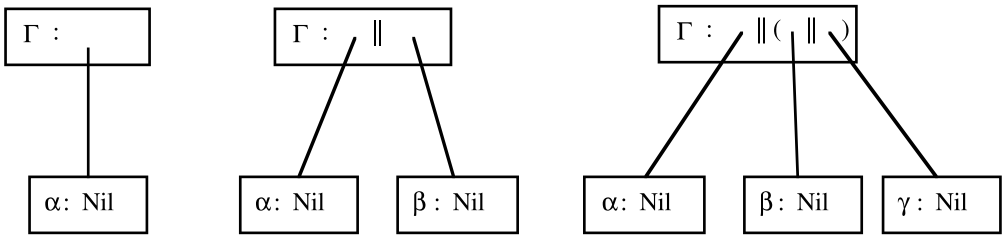
That means that in DC programs the order of the processes put in parallel is not relevant.

Compositions of different groups of entities may be equivalent

It is possible that different views of an entity can differ also for the number of dynamic subentities, as it is shown by the entity epg' = Γ: α: Nil; indeed epg' is also equal to

Γ: α: Nil || β: Nil , Γ: α: Nil || β: Nil || γ: Nil and to any number of Nil processes put in parallel (recall that processes having form id: Nil are identities for the operation ||).

Various views of the structure of epg' are graphically represented by:



Thus in the DC programs the processes which cannot perform any action (id: Nil) do not matter.

Not all operations contribute to the entity composers

Here we consider a calculus DC_1 differing from DC only for having an operation for extracting from a program a process with a given identity,

Get: ent(prog) $\times$ ident(proc) $\rightarrow$ ent(proc),

whose interpretation is given by

Get(pid: id: p || ep_1 … || ep_n, id) = id: p

Get(pid: id_1: p_1 || … || id_n: p_n, id, p') undefined if id $\neq$ id_i for i = 1, …, n.

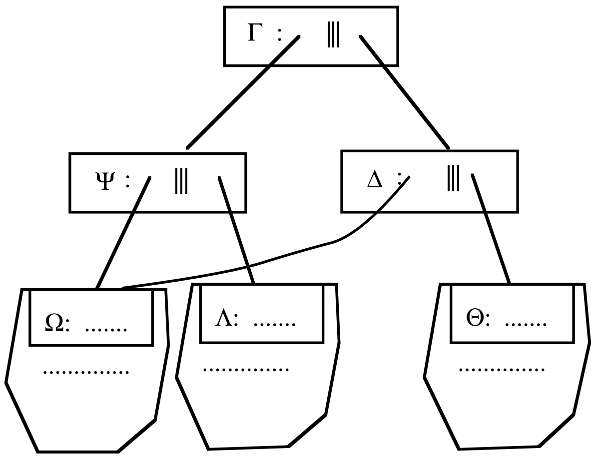
The set of entity composers on DC_1 is the same of those on DC.

Sharing of subentities

Here we consider a calculus DC_2 differing from DC only for having a multilevel parallelism instead of a flat one; we just take a new signature $D\Sigma_2$ obtained from $D\Sigma$ by replacing the operation _ || _ with
_ ||| _: ent(prog) $\times$ ent(prog) $\rightarrow$ prog,
and give a $D\Sigma_2$-algebra DC_2 in the same way of DC. In this case an entity of sort prog has either one subentity of sort proc or two subentities of the same sort prog.

epg'' = Γ: [Ψ: (Ω: pg_1 ||| Λ: pg_2) ||| Δ: (Ω: pg_1 ||| Θ: pg_3)],

is an entity where the subentity represented by Ω: pg_1 is shared between the subentities identified by Ψ and Δ; its structure is graphically represented by



Entities may terminate and new entities may be created

Here we consider a calculus DC_3 differing from DC only for allowing the termination and the creation of processes. We just take a new signature $D\Sigma_3$ obtained from $D\Sigma$ by adding the operations

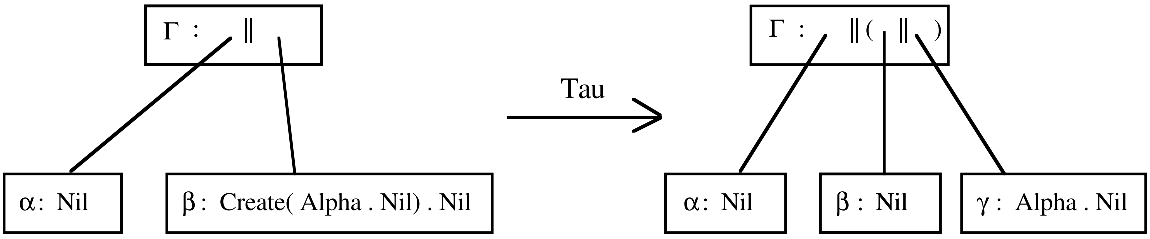
Terminate: $\rightarrow$ lab(proc)   and   Create: proc $\rightarrow$ lab(proc),

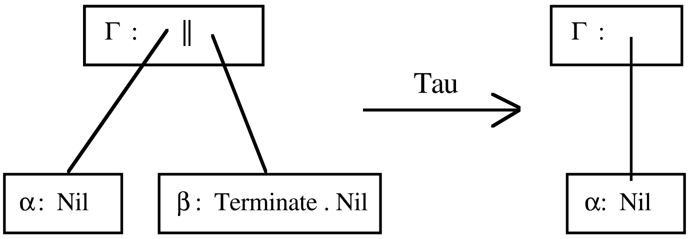
and DC_3 is defined as DC; the transitions due to the new actions are given by

$$\frac{ep \xRightarrow{\text{Terminate}} ep'}{pid\colon ep \parallel pg \xRightarrow{\text{Tau}} pid\colon pg} \qquad \frac{ep \xRightarrow{\text{Create}(p)} ep'}{pid\colon ep \parallel pg \xRightarrow{\text{Tau}} pid\colon ep' \parallel id\colon p \parallel pg}$$

for all id not used in pg

Graphically an example of a creation and of a termination of a process of DC_3 are shown by:





1.4 Entity homomorphisms

Since entity algebras are particular palgebras, we take as homomorphisms between entity algebras the total strengthening homomorphisms between palgebras (see [AC]) and will show that they have good properties. Recall that a total strengthening homomorphism between two palgebras $h\colon A \rightarrow B$ is a family of total functions $h_s\colon A_s \rightarrow B_s$ s.t.

- for all Op and $a_1, \dots, a_n \in A$
  if $Op^A(a_1, \dots, a_n)$ is defined, then $Op^B(h_{s_1}(a_1), \dots, h_{s_n}(a_n))$ is defined and
  $h_s(Op^A(a_1, \dots, a_n)) = Op^B(h_{s_1}(a_1), \dots, h_{s_n}(a_n))$;
- for all Pr and all $a_1, \dots, a_n \in A$
  if $Pr^A(a_1, \dots, a_n)$ holds, then $Pr^B(h_{s_1}(a_1), \dots, h_{s_n}(a_n))$ holds.

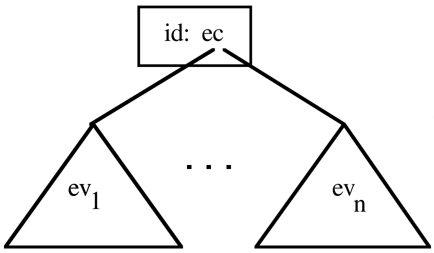
Def. 1.6 Let EA and EA' be two $E\Sigma$-algebras; a total strengthening homomorphism between $E\Sigma^E$-palgebras

$h\colon EA \rightarrow EA'$

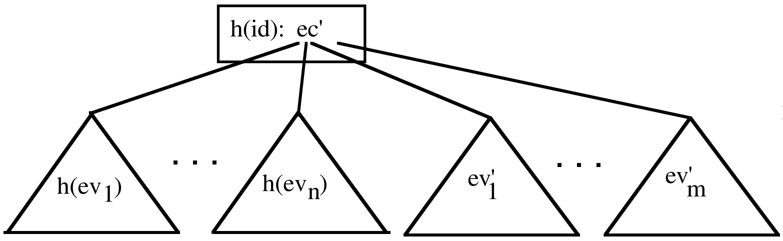
is an *entity homomorphism* from EA into EA' (still written $h\colon EA \rightarrow EA'$). ☐

It is easy to see that $eAlg_{E\Sigma}$ with the entity homomorphisms forms a category (still denoted by $eAlg_{E\Sigma}$).

Entity homomorphisms in some sense preserve the structure of the entities but can enrich it as prop. 1.7 shows; if a view of the structure of e is graphically represented by



then



represents a view of the structure of h(e).

Thus, for example,

- if e has only simple views, then h(e) could have a non-simple view;
- if h(e) has only simple views, then e has only simple views.

Prop. 1.7 Assume $h: EA \to EA'$ and e, e' elements of entity sort in EA,
if e' is a subentity of some view for e, then h(e') is a subentity of some view for h(e)
(the vice versa does not hold). □

Using entity homomorphisms we can speak of initial elements in a class of entity algebras and the following proposition shows their properties.

Prop. 1.8 Let *C* be a class of EΣ-algebras and $I \in C$ be initial in *C*; then

1) $I \models t = t'$ iff ($EA \models t = t'$ for all $EA \in C$)
   where "=" stands for strong equality;

2) for all predicates Pr of EΣ
   $I \models Pr(t_1, \ldots, t_n)$ iff ($EA \models Pr(t_1, \ldots, t_n)$ for all $EA \in C$), thus in particular
   - $I \models e \xrightarrow{l} e'$ iff ($EA \models e \xrightarrow{l} e'$ for all $EA \in C$)
     in I each entity has in some sense the minimum amount of activity,
   - $I \models D(t)$ iff ($EA \models D(t)$ for all $EA \in C$)
     I is minimally defined;

3) for all terms of entity sort et, et'
   et is a subentity of et' in I iff (et is a subentity of et' in EA for all $EA \in C$). □

Notice that 1) implies that two entity views ev and ev' are equivalent in I, ie $I \models V(ev) = V(ev')$, iff (they are equivalent in all $EA \in C$).

2 ENTITY SPECIFICATIONS IN THE SMALL

Correspondingly to the model level introduced in the previous section we can define *entity specifications* in the same way as it is done for the usual specifications of abstract data types: a specification is a couple (EΣ, Ax), where Ax is a set of formulas on EΣ and the validity is the usual validity in palgebras; and we have that this framework constitutes an institution (see [GB]).

To such specifications could be given either an initial or a loose semantics; in the first case we give conditions for its existence, while in the latter we show how the usual notion of implementation for abstract data type specifications can be extended to the entity specifications.

2.1 Entity specifications in the small

We define the *institution of entities in the small*

$\boldsymbol{e}$ = (**ESign**, eSen, eAlg, $\overset{e}{\models}$)

where:

- **ESign** is the category of the entity signatures (see the appendix); its objects are the entity signatures and its morphisms are the subclass of the morphism of psignatures respecting the particular features related to entities (eg, dynamic sorts are sent into dynamic sorts, special operations and predicates are sent into the corresponding special operations and predicates and so on);
- eSen is the sentence functor (see the appendix); sentences in ***e*** are first order formulas(*) on $E\Sigma^E$;
- eAlg is the algebra functor (see the appendix); $eAlg(E\Sigma)$ is the category $eAlg_{E\Sigma}$;
- $\overset{e}{\models}$ is the satisfaction relation defined by

  $EA \overset{e}{\models}_{E\Sigma} \vartheta$ iff $EA \overset{P}{\models}_{E\Sigma^E} \vartheta$,

  where $\overset{P}{\models}_{\Sigma}$ is the satisfaction relation in the institution of palgebras with existential equality, ie,
  $A \overset{P}{\models}_{\Sigma} t = t'$ iff t^A and t'^A are both defined and equal, thus the formula $t = t$ requires that the interpretation of t must be defined; in the following we will write $D(t)$ instead of $t = t$.

Theorem 2.1 ***e*** = (**ESign**, eSen, eAlg, $\overset{e}{\models}$) is an institution. □

Thus we can define an *entity specification (in the small)* as a couple

$ESP = (E\Sigma, Ax)$

where $E\Sigma$ is an entity signature and $Ax \subseteq eSen(E\Sigma)$; the class of the *models* of ESP, denoted by eMod(ESP), is

$\{ EA \mid EA$ is an $E\Sigma^E$-palgebra and $EA \overset{e}{\models} \vartheta$ for all $\vartheta \in Ax \}$.

To investigate the properties of entity specifications (existence of the initial model, of a sound and complete deductive system and so on), we first note that the class of the models of an entity specification $ESP = (E\Sigma, Ax)$ is the class of the models of a (partial with predicates) specification $ESP^P = (E\Sigma^E, Ax \cup eAxioms(E\Sigma))$, where

$eAxioms(E\Sigma) = \cup_{s \in Dsorts(E\Sigma)}$

(a) $\{ \exists\, id, x \,.\, e = id{:}\, x,$

(b) $id{:}\, x = id'{:}\, x' \supset id = id' \wedge x = x',$

(c) $id{:}\, x \xrightarrow{l} id'{:}\, x \supset id = id' \mid id, id'{:}\, ident(s), l{:}\, lab(s), x, x'{:}\, s, e{:}\, ent(s) \} \cup$

(d) $\{ D(tid{:}\, tv) \wedge tid' = tid'' \supset tv' = tv'' \mid$
$tid{:}\, tv \in T_{E\Sigma}(X)_{ent(s)}, tid'{:}\, tv', tid''{:}\, tv''$ are subterms of $tid{:}\, tv \}$.

Lemma 2.2 $eMod(ESP) = PMod(ESP^P)$,
where PMod denotes the class of models of a (partial with predicates) specification. □

Partial specifications with predicates whose axioms are (existential) positive conditional formulas have particular nice properties. Recall that a positive conditional formula has form

$\wedge_{1 \le i \le n} \alpha_i \supset \alpha,$

where α and α_i for $i = 1, \dots, n$ are atoms and atoms are formulas of the form either $Pr(t_1, \dots, t_n)$ with

---

(*) The set of the *first order formulas* on a predicate signature Σ and X (a Sorts(Σ)-sorted family of variables), indicated by $F_\Sigma(X)$, is inductively defined as follows:

- $t = t' \in F_\Sigma(X)$ for all $t, t' \in (T_\Sigma(X))_s$, $s \in Sorts(\Sigma)$;
- $Pr(t_1, \dots, t_n) \in F_\Sigma(X)$ for all $Pr{:}\, s_1 \times \dots \times s_n \in Preds(\Sigma)$, $t_i \in (T_\Sigma(X))_{s_i}$ $i = 1, \dots, n$;
- if $\vartheta, \vartheta' \in F_\Sigma(X)$ and $x \in X$, then $\neg \vartheta$, $\vartheta \supset \vartheta'$, $\exists\, x \,.\, \vartheta \in F_\Sigma(X)$;
- if for all $i = 1, \dots, n$ $\vartheta_i \in F_\Sigma(X)$, then $\wedge_{i = 1, \dots, n} \vartheta_i \in F_\Sigma(X)$.

Morover, $t \neq t'$, $\forall\, x \,.\, \vartheta, \vee_{i \in I} \vartheta_i$, $\exists\!\!\!\exists\, x \,.\, \vartheta$, $\vartheta \equiv \vartheta'$ are defined as abbreviations as usual.

Pr predicate or $t = t'$. In particular for such specifications there exist the initial model and a deductive system which is sound a complete w.r.t. the class of the models. If the signature of the specification Σ is such that $(T_\Sigma)_s \neq \emptyset$ for all sorts s, this system can be obtained, as in [C], by considering a system which is sound and complete for the total case and modifying it as follows: suppress reflexivity of equality; allow substitution of t for x only when t is defined (rule SUB below); add rules to assert that operations and predicates are strict (rules STRICT below).

Such system is given by the following rules, where $D(t)$ stands for $t = t$:

$$\frac{t = t'}{t' = t} \qquad \frac{t = t' \quad t' = t''}{t = t''}$$

$$\frac{D(Op(t_1, ..., t_n)) \qquad t_i = t_i' \ (i = 1, ..., n)}{Op(t_1, ..., t_n) = Op(t_1', ..., t_n')}$$

$$\frac{Pr(t_1, ..., t_n) \qquad t_i = t_i' \ (i = 1, ..., n)}{Pr(t_1', ..., t_n')}$$

$$\frac{(\wedge_{i = 1, \ldots, n}\ \vartheta_i) \supset \vartheta \qquad \vartheta_i \ (i = 1, ..., n)}{\vartheta}$$

$$\text{(STRICT)} \qquad \frac{D(Op(t_1, ..., t_n))}{D(t_i)} \qquad \frac{Pr(t_1, ..., t_n)}{D(t_i)}$$

$$\text{(SUB)} \qquad \frac{\vartheta[x] \quad D(t)}{\vartheta[t]} .$$

In the following we write $SP \vdash \vartheta$, if ϑ can be proved in the above system starting from the axioms of SP.

Theorem 2.3 Let $ESP = (E\Sigma, Ax)$ be an entity specification such that

0) for all sorts s $(T_{E\Sigma})_s \neq \emptyset$,

1) Ax is a set of positive conditional formulas,

2) for all $te \in (T_{E\Sigma})_{ent(s)}$ s.t. $ESP \vdash D(te)$
there exist $tid \in (T_{E\Sigma})_{ident(s)}$, $tv \in (T_{E\Sigma})_s$ s.t. $ESP \vdash te = tid{:}\ tv$,

then:

i) $ESP^P \vdash \alpha$ iff $(EA \models \alpha$ for all $EA \in eMod(ESP))$ for all atoms α;

ii) there exists the initial element of eMod(ESP), denoted by I_{ESP};

iii) $I_{ESP} \models \alpha$ iff $ESP^P \vdash \alpha$ for all atoms α. □

2.2 Some Examples

2.2.1 A specification of a class of objects

We give a specification describing an object-oriented version of the data type consisting of queues of integers; precisely we give a specification of a class queue whose objects are queues of objects of class int (just integers).

Let INT be the specification of a class of objects corresponding to integers with the entity sort ent(int) and we assume that each entity of sort ent(int) is simple.

The specification SPQ corresponding to the class queue is

enrich INT **by**
dsorts queue
opns
Empty: $\to$ queue (total)
Get: queue $\to$ ident(int)
Remove: ent(queue) $\to$ ent(queue)
Put: ent(int) $\times$ queue $\to$ queue
Γ, Ψ, … : $\to$ ident(queue) (total)
* _ . Empty: ident(queue) $\to$ lab(queue) (total)
* _ = _ . Get: ident(int) $\times$ ident(queue) $\to$ lab(queue) (total)
* _ . Remove: ident(queue) $\to$ lab(queue) (total)
* _ . Put(_): ident(queue) $\times$ ent(int) $\to$ lab(queue) (total)
* Queue : lab(int) $\to$ lab(queue)
-- defines the labels corresponding to sending a method to a subcomponent of class int

preds
_ Diff _ : ident(queue) $\times$ ident(queue)
Unused: queue $\times$ ident(int)
-- checks whether a queue has not an integer component with a certain identity

axioms
0 Γ Diff Ψ …

1 Get(Put(id: i, Empty)) = id
2 D(Get(q)) $\supset$ Get(Put(ei, q)) = Get(q)

3 Remove(id: Put(ei, Empty)) = id: Empty
4 Remove(id: q) = id: q' $\supset$ Remove(id: Put(ei, q)) = id: Put(ei, q')

5 Unused(id, Empty)
6 id Diff id' $\wedge$ Unused(id, q) $\supset$ Unused(id, Put(id': i, q))

7 Unused(id, q) $\supset$ D(Put(id: i, q))

-- sending a method to an element of class queue

8 id: q $\xrightarrow{\text{id . Empty}}$ id: Empty

9 D(Put(ei, q)) $\supset$ id: q $\xrightarrow{\text{id . Put(ei)}}$ id: Put(ei, q)

10 D(Remove(id: q)) $\supset$ id: q $\xrightarrow{\text{id . Remove}}$ Remove(id: q)

11 Get(q) = id' $\supset$ id: q $\xrightarrow{\text{id' = id . Get}}$ id: q

-- sending a method to a subcomponent of class int

12 ei $\xrightarrow{l}$ ei' $\supset$ id: Put(ei, q) $\xrightarrow{\text{Queue(l)}}$ id: Put(ei', q)

13 id: q $\xrightarrow{\text{Queue(l)}}$ id: q' $\supset$ id: Put(ei, q) $\xrightarrow{\text{Queue(l)}}$ id: Put(ei, q')

In this specification we have that:

- the sort ent(queue) corresponds to the class queue;
- the entities of sort ent(queue) represent the objects of the class queue;
- the operations having either as result and/or as argument the sorts queue and ent(queue) (Empty, Put, Remove and Get) correspond to the methods of the class queue (Get, whose result sort is different from queue and ent(queue), corresponds to a method returning some result);
- the activity of the entities of sort ent(queue) is the result of the application of some methods, thus their transitions are labelled by message sendings (method applications in object-oriented jargon) defined by the operations marked with *.

Notice how in this specification the structure and the methods of the class queue have been abstractly specified. Since Empty and Put are the only operations of sort queue we have that each object of class queue has n ($n \geq 0$) subcomponent objects of class int arranged in a row. Axiom 7 requires that no object of class int may appear in two different places in the same queue. Axioms 1, …, 4 say that the methods Get and Remove respectively returns an identification of and removes the last element of the row.

Axioms 8, …, 11 describe the dynamic effects of the method applications; while axioms 12 and 13 describe the application of a method of the class int to some subcomponent.

2.2.2 Non-lazy processes with handshaking communications

Here we give the specification NLP (for Non-Lazy Process) of the concurrent systems with exactly the following properties:

- the active components of the systems are processes of "the same kind" and there are no "passive components";
- each component process is identified by a unique name;
- the component processes exchange messages (not further specified) between them in a handshaking way and that is the only kind of interaction between the processes; moreover each process sending a message states the name of receivers, analogously a process receiving a message states the name of the senders;
- there are no mutual exclusion requirements on the process actions (ie, whatever number of processes can perform internal actions and whatever number of disjoint couples of processes can exchange messages between them simultaneously);
- no component process can be lazy;
- processes can neither be created nor terminate.

Note that here we are abstractly specifying handshaking and nonlaziness, not just giving a particular system having such properties.

PROC =
sorts message
dsorts process
opns Tau: $\rightarrow$ lab(process)
SEND, REC: ident(process) $\times$ ident(process) $\times$ message $\rightarrow$ lab(process)
axioms
$ep \xrightarrow{l} ep' \supset l = Tau \vee (\exists pi, pi', m \centerdot l = SEND(pi, pi', m) \vee l = REC(pi, pi', m))$

The sort message is static, its element are usually values; while process is a dynamic sort, its elements are the cores of the dynamic elements corresponding to the process components of the systems.

NLP =
enrich PROC **by**
dsorts system
opns $\emptyset$: $\rightarrow$ system
_ : ent(process) $\rightarrow$ system
_ | _ : system $\times$ system $\rightarrow$ system (comm., assoc.)
Tau : $\rightarrow$ lab(system)
preds _ $\Longrightarrow$ _ : system $\times$ system
-- auxiliary predicate used for defining the properties of the transitions of the entities of sort system
axioms
1) $D(id: s) \supset (\exists ep \centerdot s = ep) \vee (\exists s_1, s_2 \centerdot D(id: s_1) \wedge D(id: s_2) \wedge s = s_1 \mid s_2)$
2) $ep \xrightarrow{Tau} ep' \supset ep \Longrightarrow ep'$
3) $ep_1 \xrightarrow{SEND(pi, pi', m)} ep_1' \wedge ep_2 \xrightarrow{REC(pi', pi, m)} ep_2' \supset ep_1 \mid ep_2 \Longrightarrow ep_1' \mid ep_2'$
4) $s_1 \Longrightarrow s_1' \wedge s_2 \Longrightarrow s_2' \supset s_1 \mid s_2 \Longrightarrow s_1' \mid s_2'$
5) $si: s \xrightarrow{l} si: s' \supset$
5.1) $l = Tau \wedge$
5.2) $\exists s_1, s_2, s_1' \centerdot (s = s_1 \mid s_2 \wedge s_1 \Longrightarrow s_1' \wedge s' = s_1' \mid s_2 \wedge$
5.3) $\not\exists s_3, s_3', s_4 \centerdot s_2 = s_3 \mid s_4 \wedge s_3 \Longrightarrow s_3')$

Axiom 1) requires that the only subcomponents of the system states are processes specified by PROC arranged in parallel by the "|" operator; that means, for example, that a model of this specification cannot have either additional processes or buffers to handle the communications between the component processes.

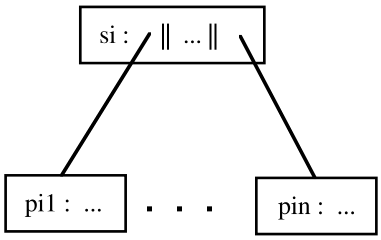
Axioms 2), 3) and 4) define the auxiliary predicate $\Longrightarrow$, which describes partial moves of groups of processes (combinations of internal actions and of handshaking communications).

Axiom 5) requires that:

- the whole system is closed (5.1);
- the only transitions of the system are caused by a combination of process internal actions and process communications (5.2), thus processes can interact between them only by exchanging messages in a handshaking way;
- no process can idle (5.3);

but note that these axioms do not specify what is the policy to be followed when some conflict arises; thus this specification admits models where the conflicts are solved by allowing nondeterministically all the possible choices, by using priorities associated with the processes and so on. Moreover axiom 5) also requires that a process can neither be created nor terminate.

The structure of the entities of sort system can be graphically represented by:



2.3 Implementation of Entity Specifications

Here we try to extend to the case of entity specifications the well-known general notion of implementation for algebraic specifications of Wirsing (see [W]); we have chosen this notion since it has been proved well adequate in the case of usual specifications.

For Wirsing a specification SP *is implemented by another specification* SP_1 *via* f, where f is a function from specifications into specifications built by composing specification operations (as enrich, rename, export, hide and so on) iff

$Mod(f(SP_1)) \subseteq Mod(SP)$.

The function f describes how the elements of the sorts, the operations and the predicates of SP can be obtained from those of SP_1; while requiring the inclusion of the classes of models instead of the coincidence allows SP_1 to be a "refinement" of SP, ie, "things" not specified in SP are made more precise in SP_1.

This notion is very general but it includes as particular cases (ie, when f has a particular form) the various notions presented in the literature as implementation by rename-forget-restrict-identify or implementation by behavioural abstraction (see [W]).

Def. 2.4 Let $ESP = (E\Sigma, Ax)$ and $ESP_1 = (E\Sigma_1, Ax_1)$ be two entity specifications and f a function from specifications with signature $E\Sigma$ into specifications with signature $E\Sigma_1$ built by composing specification operations.

ESP *is implemented by* ESP_1 *via* f (written $ESP \leadsto^{f} ESP_1$) iff $EMod(f(ESP_1)) \subseteq EMod(ESP)$. □

As examples we give two implementations of the entity specification NLP defined in sect. 2.2.2.

First we give a specification SYSTEM describing a particular concurrent calculus similar to Milner's SCCS having all the properties required by the specification NLP. In SYSTEM the elements of the static sort message and of the dynamic sort process (the cores of the system components) are completely defined (specifications CHAN and BEH); moreover the proper axioms of SYSTEM precisely define the activity of the systems.

CHAN = **sorts** chan
opns Alpha, Beta, … : $\to$ chan (total)

BEH =
enrich CHAN **by**
dsorts beh
opns SEND, REC: ident(beh) $\times$ ident(beh) $\times$ chan $\to$ lab(beh)
Tau: $\to$ lab(beh)
Nil: $\to$ beh
_ ?_ . _, _ !_ . _: chan $\times$ ident(beh) $\times$ beh $\to$ beh
_+_: beh $\times$ beh $\to$ beh (comm., assoc.)
δ _: beh $\to$ beh

axioms
$bi'\text{: } ch \,!\, bi \,.\, b \xrightarrow{SEND(bi',\, bi,\, ch)} bi'\text{: } b$

$bi'\text{: } ch \,?\, bi \,.\, b \xrightarrow{REC(bi',\, bi,\, ch)} bi'\text{: } b$

$bi\text{: } b_1 \xrightarrow{l} bi\text{: } b_1' \supset bi\text{: } b_1 + b_2 \xrightarrow{l} bi\text{: } b_1'$

$bi\text{: } \delta\, b \xrightarrow{Tau} bi\text{: } \delta\, b$

$bi\text{: } b \xrightarrow{l} bi\text{: } b' \supset bi\text{: } \delta\, b \xrightarrow{l} bi\text{: } b'$

SYSTEM =
enrich BEH **by**
dsorts system
opns
$\varnothing$: $\to$ system
_ : ent(beh) $\to$ system
_ | _ : system $\times$ system $\to$ system (comm., assoc.)
Tau : $\to$ lab(system)

axioms
$eb \xrightarrow{Tau} eb' \supset si\text{: } eb \xRightarrow{Tau} si\text{: } eb'$

$eb_1 \xrightarrow{SEND(ch,\, bi,\, bi')} eb_1' \wedge eb_2 \xrightarrow{REC(ch,\, bi',\, bi)} eb_2' \supset si\text{: } eb_1 \mid eb_2 \xRightarrow{Tau} si\text{: } eb_1' \mid eb_2'$

$si\text{: } eb_1 \xRightarrow{Tau} si\text{: } eb_1' \wedge si\text{: } eb_2 \xRightarrow{Tau} si\text{: } eb_2' \supset si\text{: } eb_1 \mid eb_2 \xRightarrow{Tau} si\text{: } eb_1' \mid eb_2'$

SYSTEM is an implementation of NLP, indeed changing the sort names chan, beh into message, process, hiding the operations Alpha, Beta, …, Nil, ?, ! , +, δ and defining the auxiliary predicate $\Longrightarrow$ we get a new specification, whose models are included into the models of NLP. Formally NLP $\sim\sim\sim>^{f}$ SYSTEM, where

f = λ X . **hide** { Alpha, Beta, …, Nil, !, ?, +, δ } **in**
enrich X[process/beh, message/chan] **by**
preds _ $\Longrightarrow$_: system $\times$ system
axioms
$si\text{: } es \xRightarrow{l} si\text{: } es' \supset es \Longrightarrow es'$

The notion of implementation of def. 2.4 allows also that simple entities are implemented by structured entities having several dynamic subcomponents, as it is shown by specifications as NLP2, given below, where the entities of type process are implemented by groups of other entities interacting between them.

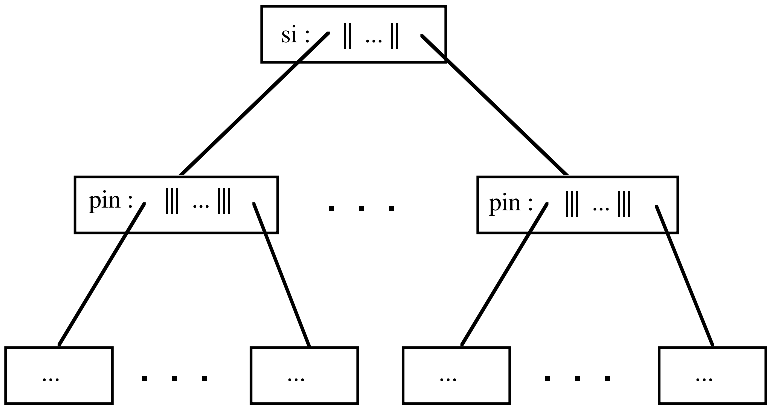
NLP2 = **enrich** NLP **by**
dsorts agent
opns _: ent(agent) $\to$ process

_ |||_: ent(agent) × process → process

axioms …

In this case NLP $\stackrel{f}{\leadsto}$ NLP2, where f is just the hiding of the dynamic sort agent and the operation |||.

The structure of the entities of sort system in NLP2 is graphically represented by:



3 FURTHER DEVELOPMENTS: ENTITY SPECIFICATIONS IN THE LARGE

Entity specifications in the small allow us to do a lot of things: for example we can formally represent object-systems where subcomponents are shared, specify concurrent systems by giving very abstract properties about their activity and so long; but they do not allow us to write down specifications whose axioms only require some very abstract properties about the structure of the entities but do not completely describe such structure; ie, we do not want to fix which are the operations for composing entities together and give their properties (as we have done in NLP), but just require some properties of the resulting structure.

This point is very important for keeping the level of the specifications very abstract.

Consider the following paradigmatic situations: we want to give an entity specification requiring one of the following conditions:

i) each entity of sort s is simple, ie, it has not entity subcomponents (eg, each process of type s is sequential, each object of type s has no object attributes);

ii) each entity of sort s has exactly two simple subentities (eg, each process of type s consists of the parallel composition of two sequential processes);

iii) each entity of sort s has either a subentity satisfying P or a subentity satisfying Q (P and Q predicates) but not both (eg, each process of type s has either a subcomponent which is a printer or a modem but not both);

but we do not want to completely describe the entities of sort s.

Situations of this kind can arise when we want to give only very abstract properties of some dynamic system without fully defining its structure. Think, for example, of giving the requirements for a net of personal computers just saying how many people must be able to work simultaneously and how many resources they may have, without specifying anything about the use of servers, bridges and so on; or also of specifying the firing squad problem (see [DHJW]).

To solve this problem we can define a new kind of entity specifications, called *entity specifications in the large*, whose axioms allow to express properties on the dynamic structure of the entities without fully describing such structure.

The idea is to introduce some special predicates "_ Are-Sub _" for checking which are the subcomponents of the entities; these predicates, given a set of entities es and an entity e of a certain sort, return true iff es is the set of all subentities (proper and not) of e w.r.t. some view.

Using these predicates the properties i), ii) and iii) may be formalized by the following formulas.

i) es Are-Sub e $\supset$ es = { e }

ii) es Are-Sub e $\supset$ $\exists\, e_1, e_2$. $e_1 \neq e_2 \neq e$ $\wedge$ es = { e_1, e_2, e }

iii) es Are-Sub e $\supset$ $\exists$ e' . e' $\in$ es $\wedge$ e $\neq$ e' $\wedge$ (P(e') $\vee$ Q(e')) $\wedge \neg$ (P(e') $\wedge$ Q(e'))

The axioms of these new entity specifications are the first order formulas on a new signature $E\Sigma^{ST}$, obtained by enriching $E\Sigma$. The validity of one of these new formulas, say ϑ, in an $E\Sigma$-entity algebra EA is defined as the validity of ϑ in a particular $E\Sigma^{ST}$-palgebra, EA^{ST}, which is an extension of EA:

$$EA \stackrel{E}{\models}_{E\Sigma} \vartheta \text{ iff } EA^{ST} \stackrel{P}{\models}_{E\Sigma^{ST}} \vartheta.$$

Also for entity specifications in the large it is possible give a corresponding institution

$$\boldsymbol{E} = (\mathbf{ESign}, \text{ESen}, \text{EAlg}, \stackrel{E}{\models})$$

where **ESign** is defined in the appendix, $\text{ESen}(E\Sigma) = \text{eSen}(E\Sigma^{ST})$ and

$$\text{EAlg}(E\Sigma) = \{ EA \mid EA \text{ is an } E\Sigma'\text{-algebra for some } E\Sigma' \text{ s.t. } E\Sigma \subseteq E\Sigma' \}$$

Here for lack of room we cannot further justify the definition of EAlg and prove that ***E*** is truly an institution (see [AR2] for a full treatment).

APPENDIX

The Category of Entity Signatures

Def. A.1 Given an entity signature $E\Sigma$, $s \in \text{Sorts}(E\Sigma^E)$ is *entity-reaching* iff for some dynamic sort s' there exists a term t of sort ent(s') having a subterm of sort s. $\square$

Def. A.2 Given two entity signatures $E\Sigma$ and $E\Sigma'$, an *entity signature morphism* ϕ is a psignature morphism (see [AC]) $\phi: E\Sigma^E \to E\Sigma'^E$ such that:

a1) $\phi(\text{Dsorts}(E\Sigma)) = \text{Dsorts}(E\Sigma')$;

a2) for all $s \in \text{Dsorts}(E\Sigma)$

$\phi(\text{ent}(s)) = \text{ent}(\phi(s))$,
$\phi(\text{lab}(s)) = \text{lab}(\phi(s))$,
$\phi(\text{ident}(s)) = \text{ident}(\phi(s))$,
$\phi(\_ : \_ : \text{ident}(s) \times s \to \text{ent}(s)) = \_ : \_ : \text{ident}(\phi(s)) \times \phi(s) \to \text{ent}(\phi(s))$,
$\phi(\_ \xrightarrow{\_} \_ : \text{ent}(s) \times \text{lab}(s) \times \text{ent}(s)) = \_ \xrightarrow{\_} \_ : \text{ent}(\phi(s)) \times \text{lab}(\phi(s)) \times \text{ent}(\phi(s))$;

a3) for all $s' \in \text{Dsorts}(E\Sigma')$

$\phi^{-1}(\{ \text{ent}(s') \})$(*) $= \{ \text{ent}(s) \mid s \in \phi^{-1}(\{ s' \}) \}$,
$\phi^{-1}(\{ \text{lab}(s') \}) = \{ \text{lab}(s) \mid s \in \phi^{-1}(\{ s' \}) \}$,
$\phi^{-1}(\{ \text{ident}(s') \}) = \{ \text{ident}(s) \mid s \in \phi^{-1}(\{ s' \}) \}$,
$\phi^{-1}(\{ \_ : \_ : \text{ident}(s') \times s' \to \text{ent}(s') \}) = \{ \_ : \_ : \text{ident}(s) \times s \to \text{ent}(s) \mid s \in \phi^{-1}(\{ s' \}) \}$,
$\phi^{-1}(\{ \_ \xrightarrow{\_} \_ : \text{ent}(s') \times \text{lab}(s') \times \text{ent}(s') \}) =$
$\{ \_ \xrightarrow{\_} \_ : \text{ent}(s) \times \text{lab}(s) \times \text{ent}(s) \mid s \in \phi^{-1}(\{ s' \}) \}$;

a4) $\phi^{-1}(\{ Op \}) \neq \emptyset$
for all $Op: s_1 \times \ldots \times s_n \to s \in \text{Opns}(E\Sigma^E)$ with s entity-reaching. $\square$

(*) If $f: A \to B$ is a function, then f^{-1} indicates the function from $P(B)$ into $\mathbb{P}(A)$ defined by $f^{-1}(\mathsf{B}) = \{ a \mid f(a) \in \mathsf{B} \}$.

Conditions a2 and a3 require that ϕ sends the special elements of $E\Sigma$ into the corresponding special elements of $E\Sigma'$; while a1 (a4) requires that all dynamic sorts (dynamic operations, ie those which build the entity composers) of $E\Sigma'$ are image w.r.t. ϕ of some dynamic sort (dynamic operation) of $E\Sigma$.

If there exists an entity signature morphism $\phi: E\Sigma_1 \to E\Sigma_2$, then in some sense $E\Sigma_1$ and $E\Sigma_2$ may be used to describe dynamic systems with a similar structure.

Fact A.3 Entity signatures and entity signature morphisms form a category denoted by **ESign**. □

The Institution of Entity Algebras in the Small ***e***

$\boldsymbol{e} = (\mathbf{ESign}, \text{eSen}, \text{eAlg}, \overset{e}{\models})$

We recall that $\boldsymbol{P} = (\mathbf{PSign}, \text{PSen}, \text{PAlg}, \overset{P}{\models})$ is the institution of palgebras with first order formulas (see [AC]).

- **The signature category**

  ESign is given above.

- **The sentence functor**

  eSen: **ESign** $\to$ **Set**

is the functor defined by:
– on objects: $\text{eSen}(E\Sigma) = \text{PSen}(E\Sigma^E)$;
– on morphisms: $\text{eSen}(\phi: E\Sigma \to E\Sigma') = \text{PSen}(\phi: E\Sigma^E \to E\Sigma'^E)$;
it is trivial to see that eSen is a functor.

- **The algebras functor**

  eAlg: **ESign** $\to$ $\mathbf{Cat}^{\mathbf{OP}}$

is the functor defined by:

- on objects: $\text{eAlg}(E\Sigma) = \text{eAlg}_{E\Sigma}$ ($\text{eAlg}_{E\Sigma}$ is a category, see fact 1.10);
- on morphisms: $\text{eAlg}(\phi: E\Sigma \to E\Sigma')$ is the restriction of $\text{PAlg}(\phi: E\Sigma^E \to E\Sigma'^E)$ to $\text{eAlg}_{E\Sigma'}$.

Fact A.4 ensures that $\text{PAlg}(\phi)(EA)$ is an $E\Sigma$-algebra; moreover, since entity homomorphisms (in the small) are just phomomorphisms and PAlg is a functor, it is easy to see that eAlg is a functor.

Fact A.4 $\text{PAlg}(\phi)(EA)$ is an $E\Sigma$-algebra. □

- **The satisfaction relation**

For all entity signatures $E\Sigma$, $\overset{e}{\models}_{E\Sigma} \subseteq |\text{eAlg}(E\Sigma)| \times \text{eSen}(E\Sigma)$ is defined by

$EA \overset{e}{\models}_{E\Sigma} \vartheta$ iff $EA \overset{P}{\models}_{E\Sigma^E} \vartheta$.

Acknowledgement. This paper grew out of some common work with Prof. E. Astesiano on further developments of the SMoLCS approach. I wish to thank him for constant inspiration and encouragement.

REFERENCES

[AC] Astesiano E.; Cerioli M. "Commuting between Institutions via Simulation", Internal Report Dipartimento di Matematica Università di Genova n. 5, June 1990.

[AGR] Astesiano, E.; Giovini, A.; Reggio, G. "Data in a concurrent environment", in (Vogt, F. ed.) *Proc Concurrency '88 (International Conference on Concurrency, Hamburg, FRG October 1988)*, Berlin, Springer Verlag, 1988 (Lecture Notes in Computer Science n.335), pp. 140-159.

[AR] Astesiano, E.; Reggio, G. "An Outline of the SMoLCS Methodology", (Venturini Zilli, M. ed.) *Mathematical Models for the Semantics of Parallelism, Proc. Advanced School on Mathematical Models of Parallelism*, Berlin, Springer Verlag, 1987 (Lecture Notes in Computer Science n. 280), pp. 81-113.

[AR1] Astesiano, E.; Reggio, G. "SMoLCS-Driven Concurrent Calculi", *Proc. TAPSOFT'87*, vol.1, Berlin, Springer Verlag, 1987 (Lecture Notes in Computer Science n. 249), pp. 169–201.

[AR2] Astesiano, E.; Reggio G. *Entity Institutions: Frameworks for Dynamic Systems*, in preparation.

[BW] Broy, M.; Wirsing, M. "Partial abstract data types", *Acta Informatica* 18 (1982), 47-64.

[BZ] Breu, R.; Zucca, E. "An algebraic compositional semantics of an object-oriented notation with concurrency", in (Veni Madhavan, C. E. ed.) *Foundations of Software Technology and Theoretical Computer Science (Proc. of the Ninth conference, Bangalore, India)*, 1989, Berlin, Springer Verlag, (Lecture Notes in Computer Science n. 405).

[C] Cerioli M. "A sound and equationally-complete deduction system for partial conditional (higher order) types" Proc. 3rd Italian Conf. on Theoretical Comp. Sci. Mantova 1989, World Scientific Pub. pp 164 - 175.

[DHJW] Denvir B.T.; Harwood W. T.; Jackson M.I.; Wray M.J. *Proc. of the Workshop on The Analysis of Concurrent Systems, Cambridge, 1983*, Berlin, Springer Verlag, 1985 (Lecture Notes in Computer Science n. 207).

[Dragon] Astesiano E.; Breu R. Hennicker R.: Reggio G.; Wirsing M.; Zucca E. *A Theory of Reuse and its Applications to Object Oriented Environments*, Deliverable of the DRAGON project, June 1990.

[GB] Goguen J.A.; Burstall R. M. "Introducing Institutions", *Logics of Programming*, Berlin, Springer Verlag, 1983 (Lecture Notes in Computer Science n. 164).

[GM] Goguen, J.A.; Meseguer, J. "Models and Equality for Logical Programming", *Proc. TAPSOFT'87*, vol.2, Berlin, Springer Verlag, 1987 (Lecture Notes in Computer Science n. 249), pp. 1-22.

[M] Meseguer J. *Rewriting as a Unified Model of Concurrency*, draft, 1990.

[FB] Fiadeiro J.; Maibaum T. "Describing, Structuring and Implementing Objects", Draft, presented at the REX School/Workshop on Foundations of Object-Oriented Languages, May, 1990.

[W] Wirsing M. "Algebraic Specifications", *Handbook of Theoretical Computer Science, Vol. B*, North-Holland, 1990.