COMPOSITION AND INTERACTION FOR BEHAVIOURAL SPECIFICATIONS

Simone Veglioni
Programming Research Group
Oxford University - U.K.
veglioni@comlab.ox.ac.uk

Francesco Parisi-Presicce
Dipart. di Scienze dell'Informazione
Università La Sapienza - I
parisi@dsi.uniroma1.it

Abstract: A (behavioural) algebraic formalization of composition and interaction for open distributed systems is proposed. Components (behavioural specifications) can be composed in a way that local behaviours and independence are preserved. This construction, called *Independent Composition*, verifies universal properties that favour distributivity of proofs. Interaction is modeled by *Interactive Composition*, a construction that adds new operations to the independent composition that affect both the components, so to allow them to communicate. As for many-sorted algebras, specifications can be put together via colimits, and the corresponding categories of models via limits.

INTRODUCTION

Modern systems are generally regarded as networks of components, thus requiring development methodologies supporting compositionality of specifications.

An algebraic semantic would be desirable for many reasons, including its formality, abstractness of specifications and effectiveness of proof techniques, but at least three ingredients are needed:

- an observational semantic, so to model states and components;
- non-determinism, as required by state abstraction;
- composition and interaction mechanisms, to model concurrency.

The long tradition of the *observable behaviour* of abstract data-types ([4; 1], to mention but a few) addresses the first point. On the other hand, non-determinism has been recently formalized in a behavioural setting with the hidden algebra framework [5; 6] and hidden abstract machines [10; 11]. This paper addresses the third point. The setting chosen is that of *hidden abstract machine* (HAM) but the principles proposed apply to behavioural specifications in general.

A HAM can be viewed as a black box having a hidden state, displays (i.e., attributes) giving details of the state in terms of data-values, and buttons (i.e., methods) providing

P. Ciancarini et al. (eds.), *Formal Methods for Open Object-Based Distributed Systems*

services, thus changing the state. We will show that, given two HAM (behavioural specifications) there is a third HAM showing the composition of the two behaviours, that is, their product: methods and attributes of P_1 will not interfere with those in P_2, and vice-versa.

Interaction between P_1 and P_2 can be expressed by means of a new method (or more than one), inserted in the independent composition, defined in terms of methods of P_1 and P_2. See [10; 11] for a pictorical representation.

Finally, a result on distribution is given. There is, in fact, a class of systems for which a global proof can be distributed along components. This is extremely important in distributed systems, where proofs can be very complex.

With respect to the paper in the previous FMOODS conference [11], here we give a more precise notion of observational semantics, and a formal categorical characterization of compositions.

PRELIMINARIES AND NOTATION

Hidden-sorted algebra [5; 6] belongs to the process of giving higher abstraction to (algebraic specification of) ADT. With respect to traditional behavioural approaches it introduces few important variations: first a common (visible) data-universe is fixed, then operations are restricted to be attributes or methods:

Definition 1 *[5; 6] Let D be a (V, Ψ) many sorted algebra, such that for each $d \in D_v$ with $v \in V$ there is some $\psi \in \Psi_{[],v}$ such that ψ is interpreted as d in D; for simplicity, we can assume that $D_v \subseteq \Psi_{[],v}$ for each $v \in V$. We call (V, Ψ, D) the* UNIVERSE OF DATA VALUES.

A HIDDEN SORTED SIGNATURE (OVER (V, Ψ, D)) *is a pair (H, Σ), where H is a set of hidden sorts, disjoint from V, and Σ is an $(H \cup V)$-sorted signature, such that:*

- *each $\sigma \in \Sigma_{w,s}$ with $w \in V^*$ and $s \in V$ lies in $\Psi_{w,s}$, and*
- *each $\sigma \in \Sigma_{w,s}$ has at most one element of w in H.*

If w contains a hidden sort then $\sigma \in \Sigma_{w,s}$ is called a METHOD *if $s \in H$, and an* ATTRIBUTE *if $s \in V$.*

A HIDDEN Σ-ALGEBRA *is a many-sorted Σ-algebra whose reduct to the visible signature Ψ is D. This means that all hidden algebras have the same visible part.*

A HIDDEN SORTED THEORY *(or* SPECIFICATION*) is a tuple (H, Σ, E), where (H, Σ) is a hidden sorted signature and E is a set of Σ-equations; we may abbreviate this to (Σ, E) if the context permits. In the course of this paper we always assume that the set of data-values is D.*

A (HIDDEN) SIGNATURE MAP $\phi : \Sigma \to \Sigma'$ *is a many sorted signature morphism $\phi : \Sigma \to \Sigma'$ mapping hidden sorts to hidden sorts, and being identity on Ψ.*

The hidden sorts are meant to be the sorts of the states, thus calling for an observational notion of equivalence: "equivalence under all experiments". This can be formalized by contexts [1]:

Definition 2 *Given a hidden sorted signature (H, Σ), a $(H \cup V)$-sorted set X of variable symbols, and a subsignature $\Delta \subseteq \Sigma$ (it plays the rôle of the observable interface of an abstract machine), then a* Δ-CONTEXT *is a visible sorted Δ-term having a single occurrence of a new variable symbol z. Let us write $c[t]$ for the result of substituting a term t for z in the context c. We let $T_\Delta[z]$ denote the set of Δ-contexts containing z.*

A Σ-algebra A OBSERVATIONALLY Δ-SATISFIES *a Σ-equation e of the form $(\forall X)\ t = t'$ if $t_1 = t'_1, ..., t_n = t'_n$, written $A \models^{\Delta}_{\Delta} e$, iff for every interpretation $\theta : X \to A$, such that $\theta_h : X_h \to \Delta r A_h$ (the subcarrier reachable from operations in Δ) for each hidden sort h, we have $\theta^*(c[t]) = \theta^*(c[t'])$ for all Δ-contexts c whenever, for $j = 1, ..., m$, $\theta^*(c_j[t_j]) = \theta^*(c_j[t'_j])$ for all Δ-contexts c_j.*

Given a hidden sorted theory $P = (\Sigma, E)$, then a CONCRETE P-MACHINE *A is a hidden Σ-algebra that observationally Σ-satisfies each equation in E.*

CM$_P$ *is the category of concrete P-machines and observational homomorphisms, where an* OBSERVATIONAL HOMOMORPHISM *$h : A \to B$ is a Σ-homomorphism from rA to rB, being the identity on visible carriers.*

P is CONSISTENT *iff* **CM**$_{(\Sigma,E)}$ *has at least one model.*

We say that P OBSERVATIONALLY Δ-SATISFIES *a Σ-equation e, written $P \models^{\Delta}_{\Delta} e$, if all concrete P-machines observationally Δ-satisfy e; that is, if $A \models^{\Sigma}_{\Sigma} P \implies A \models^{\Delta}_{\Delta} e$.*

Notice that a concrete machine is a hidden algebra whose reachable part behaviourally satisfies the theory, in the sense of [6].

Two concrete machines are observationally equivalent if they 'appear' the same, regardless of how they internally implement states:

Definition 3 *Given hidden signatures $\Delta \subseteq \Sigma$, then two hidden Σ-algebras A and B are* OBSERVATIONALLY Δ-EQUIVALENT, *written $A \equiv^{\Delta}_{\Delta} B$, iff $c[k]_A = c[k]_B$ (equality in the common data part D) for all Δ-contexts c and hidden Δ-constants k.*

In this paper we define a theory of composition for the particular class of hidden theories having a unique hidden sort, representing the state space, and a distinguished hidden constant, *init*, representing the initial state. Other hidden constants may only denote states reachable from *init*:

Definition 4 *A* (HIDDEN) ABSTRACT MACHINE — *(H)AM — is a hidden-sorted theory (Σ, E) with only one hidden sort h and a hidden constant $init :\to h$.*

States, $p(V_1, ..., V_n)$, where $p : v_1 ... v_n \to h$, must be init*-reachable: the equation $(\forall \{V_1{:}v_1, ..., V_n{:}v_n\})\ p(V_1, ..., V_n) = t$ must be in E, where t is a term not containing hidden constants other than* init. *Similarly for any hidden constant $c :\to h$.*

Example 1 *With OBJ3 [7] let us specify a coffee dispenser. It has attributes giving the number of coins inserted by the last client (*`money`*), the quantity of coffee available (*`level`*) and the total amount of money stored (*`amount`*); and methods to insert a coin (*`coin`*), to get a coffee (*`coffee`*), to fill up the coffee container (*`fill`*) and to collect all coins stored (*`collect`*). Only one coin is needed to get a coffee; and if*

there is no coffee available, or the client has already inserted a coin (without getting a coffee), a coin is not accepted. The procedure of inserting a coin - then getting a coffee cannot be interrupted, i.e., after having inserted a coin, no service can be provided except that of giving a coffee. In the initial state the machine is empty. See [10] for a transitional view of the behaviour.

```
obj COFFEEDIS is
  pr NAT .                          *** the data universe
  sort h .
  op init :  -> h .
  op money :  h -> Nat .            *** attributes
  op amount :  h -> Nat .
  op level :  h -> Nat .
  op coin :  h -> h .               *** methods
  op coffee :  h -> h .
  op collect :  h -> h .
  op fill :  h -> h .
  var H : h .  var N : Nat .
  eq money(init) = 0 .
  eq amount(init) = 0 .
  eq level(init) = 0 .
  cq coin(H) = H if level(H) == 0 or money(H) == 1 .
  cq money(coin(H)) = 1 if level(H) > 0 .
  eq amount(coin(H)) = amount(H) .
  eq level(coin(H)) = level(H) .
  cq coffee(H) = H if money(H) == 0 .
  eq money(coffee(H)) = 0 .
  cq amount(coffee(H)) = amount(H) + 1 if money(H) == 1 .
  cq level(coffee(H)) = level(H) - 1 if money(H) == 1 .
  cq collect(H) = H if money(H) == 1 .
  cq amount(collect(H)) = 0 if money(H) == 0 .
  eq money(collect(H)) = money(H) .
  eq level(collect(H)) = level(H) .
  cq fill(H) = H if money(H) == 1 .
  cq level(fill(H)) = 50 if money(H) == 0 .
  eq money(fill(H)) = money(H) .
  eq amount(fill(H)) = amount(H) .
endo
```

We may regard a method m 'not available' in a state s if the equation $(\forall \emptyset)\ m(s, \overline{d}) = s$ is observationally satisfied by the machine, for all appropriate tuples $\overline{d}$ of data-values.

Hidden abstract machines can be non-deterministic, in that the execution of a method can lead to different states (i.e., to an *underspecified state*). This gives rise to the view of nondeterminism formalized in [12] for process algebras: a nondeterministic system can be regarded as a class of deterministic ones, each one representing a solution, that is a possible world.

Definition 5 *An abstract machine $P = (\Sigma, E)$ is deterministic if it is consistent and, for every ground hidden Σ-term s (state), attribute a, appropriate tuple $\overline{d}$ of data-values, there is a $d \in D$ (it will be exactly one because of consistency) such that $P \models^{\Sigma}_{\Sigma} (\forall \emptyset)\ a(s, \overline{d}) = d$.*

INDEPENDENT COMPOSITION

So far, we have seen how to specify single machines. In this section we show how more machines can be composed in order to model distributed systems.

We first formalize a notion of composition in which components do not interact. This is important to regard two or more systems as a unique system, in the same way as two stack structures can be considered as a unique two-stack structure, or [11] two one-place buffers as a two-place buffer (a two-place buffer is an array of lenght two, not a buffer of capacity two).

A major result is that, as for many-sorted algebras [2], abstract machines (specifications) can be put together via colimits, and the corresponding categories of concrete machines (algebras) via limits.

Definition 6 *Given abstract machines $P_1 = (\Sigma_1, E_1)$, $P_2 = (\Sigma_2, E_2)$ and $P = (\Sigma, E)$, then hidden signature maps $\Sigma_1 \xrightarrow{\varphi_1} \Sigma \xleftarrow{\varphi_2} \Sigma_2$ are* INDEPENDENT *iff for all $i, j \in \{1, 2\}$ with $i \neq j$, we have:*

- $\varphi_1(init) = \varphi_2(init)$,
- $P \models^{\phi\Sigma_i}_{\phi\Sigma_i} \varphi_i(E_i)$ *(φ_i is a refinement),*
- $P \models^{\Sigma}_{\Sigma} (\forall \underline{X})(\forall \underline{Y})(\forall H)\ \varphi_i(a_i)(\varphi_j(m_j)(H, \underline{Y}), \underline{X}) = \varphi_i(a_i)(H, \underline{X})$,
- $P \models^{\Sigma}_{\Sigma} (\forall \underline{X})(\forall \underline{Y})(\forall H)\ \varphi_i(m_i)(\varphi_j(m_j)(H, \underline{Y}), \underline{X}) = \varphi_j(m_j)(\varphi_i(m_i)(H, \underline{X}), \underline{Y})$,

where H is an h-sorted variable, $\underline{X}, \underline{Y}$ are tuples of visible variables, and m_i and a_i represent respectively a generic attribute and a generic method of P_i. In this case, we will write $\varphi_1 \perp \varphi_2$, and call the last two forms of axioms INDEPENDENCE AXIOMS. *This can be extended to a family of maps.*

The insight behind this definition is that independent maps determine an abstract machine which shows a possible composition of behaviours. We have in fact that local behaviours are preserved ($P \models^{\varphi\Sigma_i}_{\varphi\Sigma_i} \varphi_i(E_i)$) and that methods of a machine do not affect attributes of other machines (independence axioms).

Independent composition is defined to be an initial abstract machine (i.e., minimal) among all machines showing a composition of behaviours:

Definition 7 *Given a collection $\mathcal{P} = \{P_i \mid i \in I\}$ of abstract machines, let $\mathbf{Ind}(\mathcal{P})$ be the full subcategory of co-cones φ over $\mathcal{P}$ such that $\varphi_i \perp \varphi_j$ for all $i \neq j \in I$. That is, $\mathbf{Ind}(\mathcal{P})$ is the category whose objects are families $\varphi = (\varphi_i : P_i \to P)_{i \in I}$ of independent signature maps, and whose morphisms $\rho : \varphi \to \varphi'$ are refinement $\rho : P \to P'$ such that $\varphi_i ; \rho = \varphi'_i$ for all $i \in I$.*

Let us call an initial object in $\mathbf{Ind}(\mathcal{P})$ an INDEPENDENT COMPOSITION *of P_i, and let us denote it by $||_{i \in I}\ P_i$ or, more concisely, by $||\mathcal{P}$.*

Any family of machines can be composed as the independent composition always exists (giving rise to a colimit construction):

Theorem 1 *Every collection* $\mathcal{P} = \{P_i = (\Sigma_i, E_i) \mid i \in I\}$ *of abstract machines has an independent composition.*

The category of models of an independent composition is the product of the categories of models of components:

Theorem 2 *Let us consider two abstract machines* $P_1 = (\Sigma_1, E_1)$ *and* $P_2 = (\Sigma_2, E_2)$, *and the independent composition* $P = P_1 \parallel P_2$. *Then, there are functions* $\pi_i : |CM_P| \to |CM_{P_i}|$ *and* $(\_\,,\_) : |CM_{P_1}| \times |CM_{P_2}| \to |CM_P|$ *which satisfy the following properties (*A, A_1 *and* A_2 *are respectively concrete* P, P_1 *and* P_2*-machines):*

$$\pi_i(A_1, A_2) \equiv^{\Sigma_i}_{\Sigma_i} A_i \qquad i = 1, 2$$

$$(\pi_1 A, \pi_2 A) \equiv^{\Sigma_i}_{\Sigma_i} A.$$

Functions π_i and $(\_\,,\_)$ can be extended to functors in the obvious way, giving rise to a limit construction. An important consequece of the theorem is that the independent composition of consistent machines is consistent. This is not verified, for example, by *Independent Sum* in [5].

A crucial problem of independent composition is that the set of equations obtained is in general infinite, because all ground instances must be taken. We cannot directly take the equations of components because the composition of two consistent machines would not be necessarily consistent [10].

The following theorem shows that this is not a problem, as a larg class of abstract machines admit finite compositions.

Definition 8 *Given a hidden signature* Σ, *the* ROOT *of a hidden* Σ*-term is recursively defined as follows:*

- $root(H) = H$, *for all hidden variables* H,
- $root(c) = init$, *for all hidden constants* c,
- $root(m(t, \overline{v})) = root(t)$, *for all methods* m, *hidden terms* t *and visible tuple* $\overline{v}$.

Theorem 3 *Given root preserving abstract machines* $P_1 = (\Sigma_1, E_1)$ *and* $P_2 = (\Sigma_2, E_2)$, *then the (finite) abstract machine* $P' = (\Sigma, E')$, *with* $E' = \bigcup_{i=1,2} \iota_i(E_i) \cup \bigcup_{i \neq j} (ABS_{ij} \cup COM_{ij})$ *is observationally equivalent to* $P_1 \parallel P_2$.

INTERACTIVE COMPOSITION

Our approach to interaction is not sophisticated and roughly consists in making the independent composition and then adding an interaction specification, that is, a specification of methods that affect states of both machines.

Supposing m_1 and m_2 are respectively methods of two abstract machines, handshaking communication can be realized with the following specification of *inter* (an added method):

$inter(H) = m_1(m_2(H))$ *if* "both m_1 and m_2 are available"
$inter(H) = H$ *if* "at least one is not available"

(Our treatment of 'non-available' methods does not allow to distinguish methods that idle and cannot synchronize - represented with an absense of arrow in process algebras - from methods that idle and can synchronize - represented with idling arrows in process algebras. However, they can be distinguished by writing a stronger condition, or by using 'availability' attributes, i.e., attributes that in any state say which method can synchronize.)

Once we have inserted an interaction method, which can be seen as an abstract communication line, we may 'program' it in order to model the desired kind of interaction. This means that we are able to model non-deterministic interactions too. Suppose, in fact, to model a lossy communication between a sender and a receiver, that is, a communication which can be successful (the element sent is received) or unsuccessful (the element sent is lost). In our framework this can be modeled directly at the communication level (e.g. without using channels) by defining the following interaction specification:

```
LOSSY is
  op lost :  h -> bool .        *** underspecified attribute
  op trans :  h -> h .
  var H : h .
  cq trans(H) = send(H) if lost(H) .              *** unsuccess
  cq trans(H) = receive(send(H)) if not lost(H) . *** success
```

Following this line, the alternating bit protocol can be specified without explicitly building channels [10].

The reader may notice that not every enrichment of an independent composition realizes interaction. For example, if we add to $P_1 \parallel P_2$ a set of attributes and methods, without imposing equations, then the two components will not interact. Roughly speaking, interaction can be captured if the methods we add are defined in terms of methods of the individual components. This means that they do not generate or unify states. Formally,

Definition 9 *Let us consider two abstract machines* $P_1 = (\Sigma_1, E_1)$ *and* $P_2 = (\Sigma_2, E_2)$, *and the independent composition* $P_1 \parallel P_2$. *Let* IS *denote a triple* (At, Me, Eq) *of attributes and methods not already in* $\Sigma_1 \parallel \Sigma_2$, *and equations in the signature* $\Sigma_1 \parallel \Sigma_2 \cup At \cup Me$. *Let* $P_1 \parallel P_2/IS$ *denote the abstract machine obtained by adding* IS *to* $P_1 \parallel P_2$. IS *is an* INTERACTION SPECIFICATION OF P_1 AND P_2 *if for every concrete* $(P_1 \parallel P_2/IS)$*-machine* A, *every reachable state in* A *is equivalent to a* $(\Sigma_1 \parallel \Sigma_2)$*-reachable state.*

In this case, $P_1 \parallel P_2/IS$, *is an* INTERACTIVE COMPOSITION OF P_1 AND P_2 ALONG IS. *Again, this can be extended to a family of machines.*

Checking whether an enrichment of an independent composition is interactive might not be easy. The following is a safe schema to build interactive compositions: for every method $m : h\overline{v} \rightarrow h$ in IS we write a number of equations eq_i $(i = 1, .., n)$

eq_i : $\quad m(H, \overline{V}) = t_i(H, \overline{V})$ if $C_i(H, \overline{V})$

where t_i are $(\Sigma 1 \parallel \Sigma_2)$-contexts and C_i are (conjunction of) visible contexts in the signature of the interactive composition. The conditions must be exclusive (otherwise

states might be unified): $C_i \wedge C_j = false$ or all $i \neq j$; and total (otherwise new states might be generated): $\vee_{i=1,..,n} c_i = true$.

An important class of interaction specifications, on which we will prove a distributivity property, is the following:

Definition 10 *An interaction specification IS of $P_1 = (\Sigma_1, E_1)$ and $P_2 = (\Sigma_2, E_2)$ is* BASIC *if it consists of only methods $m : h \rightarrow h$ and equations of this form:*

$m(H) = t(H)$ if $c_1(H), ..., c_n(H)$

where m is an IS method, t a $(\Sigma 1 \parallel \Sigma_2)$-context, and c_i a visible Σ_1 or Σ_2-context.

The most suitable verification technique for hidden theories seems to be coinduction with term rewriting [6]. It is notorious that term rewriting poses many NP problems, therefore it is extremely important to perform a verification task on theories having a small number of equations. But this number may become greater and greater as compositions are realized. Therefore, it seems particularly important to be able to distribute the verification of a global property among components:

Theorem 4 (Distribution) *Given two abstract machines $P_1 = (\Sigma_1, E_1)$ and $P_2 = (\Sigma_2, E_2)$, the independent composition $P = P_1 \parallel P_2$, a subsignature $\Delta \subseteq \Sigma_1 \parallel \Sigma_2$ and an equation e whose hidden terms do not involve attributes or visible functions (they only have constants as visible subterms), then, $P \models^{\Delta}_{\Delta} e$ iff $P_i \models^{\Delta_i}_{\Delta_i} \pi_i(e)$ for $i = (1, 2)$, where Δ_i consists of attributes and methods of Σ_i whose translation is included in $\Sigma_1 \parallel \Sigma_2$, and where:*

$\pi_i(init) = init,$

$\pi_i(m(t, \overline{d})) = m(\pi_i(t), \overline{d})$ *if* $m \in \Sigma_i$,

$\pi_i(m(t, \overline{d})) = \pi_i(t)$ *if* $m \notin \Sigma_i$.

In general, an equation can be distributed if every visible term involved can be reduced to a data-value. This is the case, e. g., of ground equations in a deterministic composition and an interactive composition with a basic interaction specification:

Corollary 1 *Given a basic interactive deterministic composition $P_1 \parallel P_2/IS$, then every ground equation can be distributed.*

A distributed proof for a sender-receiver abstract machine is given in [11].

Example 2 *Here we sketch how the dispenser can be* **decomposed** *into a money menager* MM *and a coffee manager* CM*, interacting one another. The complete example, with a correctness proof can be found in [10].*

```
obj MM is
  pr DATA .   sort h .    op init :  -> h .
  op moneyMM : h -> [0,1] .
  op amount :  h -> Nat .
  op coinMM : h -> h .
  op coffeeMM : h -> h .
```

```
  op collect :  h -> h .
  var H : h .  var N : Nat .
  eq moneyMM(init) = 0 .
  eq amount(init) = 0 .
  eq moneyMM(coinMM(H)) = 1 .
  eq amount(coinMM(H)) = amount(H) .
  cq coffeeMM(H) = H if moneyMM(H) == 0 .
  eq moneyMM(coffeeMM(H)) = 0 .
  cq amount(coffeeMM(H)) = amount(H) + 1 if moneyMM(H) == 1 .
  cq collect(H) = H if moneyMM(H) == 1 .
  eq moneyMM(collect(H)) = moneyMM(H) .
  cq amount(collect(H)) = 0 if moneyMM(H) == 0 .
endo

obj CM is
  pr DATA .   sort h .    op init :  -> h .
  op moneyCM : h -> [0,1] .
  op level :  h -> Nat .
  op coinCM : h -> h .
  op coffeeCM : h -> h .
  op fill :  h -> h .
  var H : h .  var N : Nat .
  eq moneyCM(init) = 0 .
  eq level(init) = 0 .
  cq coinCM(H) = H if level(H) == 0 or moneyCM(H) == 1 .
  cq moneyCM(coinCM(H)) = 1 if level(H) > 0 .
  eq level(coinCM(H)) = level(H) .
  cq coffeeCM(H) = H if moneyCM(H) == 0 .
  eq moneyCM(coffeeCM(H)) = 0 .
  cq level(coffeeCM(H)) = level(H) - 1 if moneyCM(H) == 1 .
  cq fill(H) = H if moneyCM(H) == 1 .
  eq moneyCM(fill(H)) = moneyCM(H) .
  cq level(fill(H)) = 50 if moneyCM(H) == 0 .
endo
```

The reader may notice the parameterization/modularization expressed in this example: the coffee manager is concerned with methods and attributes that more directly have to do with coffee. However, in order to specify them, it needs a method `coinCM` and an attribute `moneyCM`, which are only partially specified in `CM`, and can be thought of as parameters. By means of an *interactive composition*, these parameters will be actualized by `MM`, which, on the other hand, implements `coinMM`, `collect`, and `amount`, and, in order to do so, requires `coffeeMM` (implemented in `CM`):

```
obj COFFEEDIS-IMPL is
  pr CM||MM .
  op COIN COFFEE : h -> h .
  var H : h .
  cq COIN(H) = coinMM(coinCM(H)) if
        moneyMM(H) == 0 and moneyCM(H) == 0 and level(H) > 0 .
```

```
    cq COIN(H) = H if moneyMM(H) == 1 or moneyCM(H) == 1 or
          levelCM(H) == 0 .
    cq COFFEE(H) = coffeeMM(coffee(CM(H)) if moneyMM(H) == 1 and
          moneyCM(H) == 1 .
    cq COFFEE(H) = H if moneyMM(H) == 0 or moneyCM(H) == 0 .
  endo
```

If we take the view that a method that does not change the state is "not available", then this interactive composition realizes handshaking communication.

Distribution is guarantedd, as `COFFEEDIS-IMPL` is basic and deterministic.

OTHER APPROACHES TO COMPOSITION

The first construction aiming to composing hidden theories is *Independent Sum* by Goguen and Diaconescu [5]. This construction has profoundly inspired our independent composition, where one difference is that it is defined in terms of independent *horizontal* signature morphisms, morphisms which concern encapsulation of classes and are not appropriate to composition. A direct extension of independent composition to general hidden signature maps leads to a composition $P = P_1 \parallel P_2$ which differs from ours because equations of P_1 and P_2 are taken as they are, i.e., without reducing them to ground instances. We have already seen the need to have ground instances in order to avoid inconsistency problems. Here, this problem is also linked to the treatment of constants: suppose P_1 has an attribute $a_1 : h \rightarrow Nat$, a method $m_1 : h \rightarrow h$ and an equation $a_1(m_1(H)) = a_1(H) + 1$. Suppose, then, that P_2 has a hidden constant k_2, a method $m_2 : h \rightarrow h$ and an equation $m_2(H) = k_2$. Then, the term $m_1(m_2(k_2))$ in the independent sum is equivalent to both k_2 and $m_1(k_2)$, because of commutativity axioms, thus contradicting the axiom of P_1 and raising inconsistency.

Concurrent Connection [6] is a construction similar to independent composition when hidden theories are abstract machines and only the initial state is shared. When the shared part is empty the concurrent connection of P_1 and P_2 is consistent if P_1 and P_2 are, but determinism is not preserved, thus suggesting a different treatment of constants. Finiteness is not guaranteed, and when the shared part is not empty, neither consistency is guaranteed. Furthermore, methods with different functionality cannot be shared, and neither methods with same functionality but different behaviour, thus being impossible even to compose two buffers of capacity one to form a buffer of capacity two. This construction seems to work properly mainly when the connection refers to copies of a same specification (memory cells, for example) and same methods and attributes are shared, because forcing a sharing of parts with different behaviour gives rise to inconsistency. In this direction, further work is necessary to find sufficient conditions for the shared part to have the same behaviour in all components, and for the composition to be finite. After this, it should be possible to define a sort of interactive composition with sharing capturing both sharing and interaction.

Recently, Fiadeiro and Maibaum [3] use the same approach of concurrent connection, but on a different institution, therefore, we expect it to inherit the same difficulties, if applied to a behavioural equational setting.

Finally, we remark that the idea of representing interaction as a restriction of the product of behaviours has been first followed by Hoare [8] and Milne [9].

CONCLUDING REMARKS

We have seen how composition and interaction of components in a distributed system can be formalized in an algebraic way. This allows us to use a unique and *homogeneous* methodology to express both static and dynamic aspects of computer systems. In particular, we have defined two constructions: *Independent Composition*, used to regard two independent components as a unique entity, and *Interactive Composition*, to compose components that communicate.

We have shown that these constructions verify very desirable universal properties, including the distribution of proofs along components.

References

[1] Michel Bidoit, Rolf Hennicker, and Martin Wirsing. Behavioural and abstractor specifications. *Science of Computer Programming*, 1995.

[2] Hartmut Ehrig and Bernd Mahr. *Fundamentals of Algebraic Specification 1: Equations and Initial Semantics*. Springer-Verlag, 1985.

[3] José L. Fiadeiro and Tom Maibaum. Categorical semantics of parallel program design. *Science of Computer Programming*, 28:111–138, 1997.

[4] A. Giarratana, F. Gimona, and Ugo Montanari. Observability concepts in abstract data specifications. In *Proceedings, Mathematical Foundations of Computer Science '93*. Springer, 1976. Lecture Notes in Computer Science, Volume 45.

[5] Joseph A. Goguen and Răzvan Diaconescu. Towards an algebraic semantics for the object paradigm. In Hartmut Ehrig and Fernando Orejas, editors, *Recent Trends in Data Type Specification*. Springer-Verlag Lecture Notes in Computer Science 785, 1994.

[6] Joseph A. Goguen and Grant Malcolm. A hidden agenda. Technical Report CS97-538, UCSD, 1997. To appear in Theoretical Computer Science.

[7] Joseph A. Goguen, Timothy Winkler, José Meseguer, Kokichi Futatsugi, and Jean-Pierre Jouannaud. Introducing OBJ. In Joseph A. Goguen and Grant Malcolm, editors, *Software Engineering with OBJ: Algebraic Specification in Practice*. Cambridge University Press, 1996.

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

[9] George J. Milne. Circal and the representation of communication, concurrency and time. *ACM Transaction on Programming Languages and Systems (TOPLAS)*, 7(2):270–298, 1985.

[10] Simone Veglioni. *Integrating Static and Dynamic aspects in the specification of Open Object-based Distributed Systems*. PhD thesis, Programming Research Group, Oxford University, 1997. Available on `ftp://ftp.univaq.it/pub/users/veglioni/thesis.ps`.

[11] Simone Veglioni. Objects as Abstract Machines. In *Proceedings, FMOODS'97*. Chapman & Hall, 1997.

[12] Simone Veglioni and Rocco de Nicola. Possible worlds for process algebras. In *Proceedings, CONCUR'98*, Lecture Notes in Computer Science.