

CATEGORICAL SEMANTICS OF PARALLEL PROGRAM DESIGN^(*)

JoséLuizFiadeiro

Department of Informatics
Faculty of Sciences, University of Lisbon
Campo Grande, 1700 Lisboa
PORTUGAL
llf@di.fc.ul.pt

TomMaibaum

Department of Computing
Imperial College of Science, Technology and Medicine
180 Queen's Gate, London SW7 2BZ
UNITED KINGDOM
tsem@doc.ic.ac.uk

Abstract— We formalise, using Category Theory, modularisation techniques for parallel and distributed systems based on the notion of superposition, showing that parallel program design obeys the "universal laws" formulated by J.Goguen for General Systems Theory, as well as other algebraic properties of modularity formulated for Specification Theory. The resulting categorical formalisation unifies the different notions of superposition that have been proposed in the literature and clarifies their algebraic properties with respect to modularisation. It also suggests ways of extending or revising existing languages in order to provide higher levels of reusability, modularity and incrementality in system design.

1 Introduction

The role of Category Theory in supporting the definition of 'scientific laws' of system modularisation and composition has been recognised since the early 70s when J.Goguen proposed the use of categorical techniques in *General Systems Theory* for unifying a variety of notions of system behaviour, including that of physical components, and their composition techniques [Goguen 71, 73, Goguen and Ginali 78]. Similar principles have been used to formalise process models for concurrent systems [Sassone et al 93] such as transition systems, synchronisation trees, event structures, etc. Based on similar categorical models, modularisation principles like those typical of object-oriented programming have been formalised [Ehrich et al 91, Costa et al 92, Goguen 92]. Through *institutions* [Goguen and Burstall 92], the theories of a logic have been shown to constitute a category whose morphisms correspond to property preserving translations between their languages; several modularisation techniques for specifications have been developed on the basis of this categorical formalisation [Sannella and Tarlecki 88, Veloso and Maibaum 95].

In this paper, we show that the unification of modularisation principles provided by Category Theory applies not only to mathematical models of program behaviour and their logical specifications, but also to parallel program design languages based on the notion of *superposition* [Bougé and Francez 88, Chandy and Misra 88, Kurki-Suonio and Järvinen 89, Francez and Forman 90b, Katz 93]. Superposition was proposed as a means of supporting a layered approach to systems design by which we are allowed to build on already developed components (drawing on the services they provide) by "augmenting" them (by, say, extending their state space and/or their actions/control activity) while

(*) This work was partially supported by the Esprit BRA 8319 (MODELAGE), the HCM Scientific Network CHRX-CT92-0054 (MEDICIS), and the PRAXIS XXI contract 2/2.1/MAT/46/94 (ESCOLA).

preserving their properties. In mathematics, preservation of structure is usually formalised in terms of (homo)morphisms between the objects concerned. This is why we decided to formalise superposition in terms of morphisms of programs.

Based on this formalisation, we show which algebraic properties of superposition justify the assertion that parallel program design languages such as IP (Interacting Processes) [Francez and Forman 90a] and UNITY [Chandy and Misra 88] can support a modular approach to program development, allowing software in general to be built from basic building blocks that can be extended and interconnected. We further show how the proposed categorical formalisation can contribute to an increased reusability of programs and incrementality in the design process.

Having these goals in mind, the remainder of the paper is structured as follows.

Section 2 defines the syntax and semantics of COMMUNITY – the language that we will use to illustrate the categorical formalisation of parallel program design. The differences between COMMUNITY and IP and UNITY were all motivated by categorical principles as explained in the other sections.

In section 3, we show how superposition in the sense of UNITY, i.e. as a transformation between programs, can be captured through the morphisms of a category of COMMUNITY programs. We show how different notions of superposition give rise to different categories and that the notion of spectative superposition satisfies an important property from the point of view of modularisation: it is model-expansive.

In section 4, we show how, through universal constructions in the category of COMMUNITY programs, we can formalise parallel composition of programs, thus capturing the sense in which IP defines superposition. Hence, it emerges from the categorical formalisation that both uses of the notion of superposition can be unified in a strong algebraic sense.

Finally, in section 5, we put this formalisation to work in addressing the configuration of complex systems. We argue that diagrams in the category of COMMUNITY programs capture configurations of complex systems, and show how COMMUNITY supports incremental design. The notion of superposing a regulator over a base program defined in [Francez and Forman 90b] is formalised in this setting, and so is the superposition of *observers* or *monitors* [Katz 93] over base programs. Based on the algebraic properties of spectative superposition, namely the fact that pushouts preserve spectative morphisms, we show how the two configuration techniques support modularity in the development process.

The paper relies only on elementary notions of Category Theory, all of which can be found in any textbook, e.g. [Barr and Wells 90].

2 A parallel program design language

The language that we chose to illustrate the categorical formalisation of parallel program design, COMMUNITY, is in the style of UNITY [Chandy and Misra 88] and combines elements from IP [Francez and Forman 90a] for a richer model of system interconnection and superposition.

2.1 The language

A COMMUNITY program P has the following structure:

$$\begin{aligned}
 P \equiv & \textit{data } \Sigma \\
 & \textit{read } R \\
 & \textit{var } V \\
 & \textit{init } I \\
 & \textit{do } \prod_{g \in \Gamma} g: [B(g) \rightarrow \prod_{a \in D(g)} a := F(g,a)]
 \end{aligned}$$

where

- Σ represents the data types that the program uses; if we intend to use COMMUNITY to actually develop programs in a given environment, then Σ represents the data types available in that environment and, hence, is fixed for every program (and is thus omitted); however, to support more abstract levels of program design, it may be helpful to work with specifications of these data types, in which case Σ can be given through a signature (S, Ω) in the usual algebraic sense [Ehrig and Mahr 85], i.e. S is a set (of sort symbols) and Ω is an $S^* \times S$ -indexed family (of function symbols), together with a set of (first-order) axioms over (S, Ω) defining the properties of the operations;
- R is the set of external attributes, i.e. the attributes that the program needs to read from its environment (*open* attributes in the sense of IP);
- V is the set of local attributes (the program "variables");
- We denote by A the union (assumed disjoint) of R and V – the set of attributes of the program; attributes are typed – every attribute $a \in A$ has an associated sort s ; A_s will denote the set of attributes of sort s ; the distinction between the two classes of attributes is necessary to formalise superposition, namely forms of program interconnection that result from superposing regulators over base programs – a regulator can read the attributes of the base program but cannot update them.
- Γ is the set of *action names*; each action name has an associated command (multiple assignment) that it performs atomically, and can act as a *rendez-vous* point for program synchronisation;
- I is a condition on the attributes – the initialisation condition;
- for every action $g \in \Gamma$, $B(g)$ is a condition on the attributes – the *guard* of the action;
- for every action $g \in \Gamma$, $D(g) \subseteq V$ is the set of attributes that action g can change; we also denote by $D(a)$, where $a \in V$, the set of actions that can change a ;
- for every action $g \in \Gamma$ and local attribute $a \in D(g)$, $F(g,a)$ is an expression that has the same type as a .

Formally,

Definition 2.1: A *program signature* is a quadruple (Σ, V, R, Γ) where

- Σ is a data signature in the algebraic sense [Ehrig and Mahr 85];
- V and R are S -indexed families of sets.
- Γ is a 2^V -indexed family of sets.

All these sets of symbols are assumed to be finite and mutually disjoint. ■

For simplicity, we shall assume that the data types are fixed and omit the *data* clause from programs. We shall also use the notation (A, Γ) , where $A = V \oplus R$, or $(A = V \oplus R, \Gamma)$, for program signatures.

Attributes are used as atoms in the definition of terms:

Definition 2.2: Given a signature $\theta=(A,\Gamma)$, the language of *terms* is defined as follows: for every sort $s \in S$,

$$t_s ::= a \mid c \mid f(t_{1s_1}, \dots, t_{ns_n})$$

for $a \in A_s$, $c \in \Omega_{\langle \rangle, s}$, and $f \in \Omega_{\langle s_1, \dots, s_n \rangle, s}$;

The language of *propositions* is defined as follows:

$$\phi ::= (t_{1s} =_s t_{2s}) \mid (\phi_1 \supset \phi_2) \mid (\phi_1 \wedge \phi_2) \mid (\neg \phi) \quad \blacksquare$$

For simplicity, every boolean term b will be used as an abbreviation of the proposition ($b = \text{true}$). Terms and propositions are used to define programs.

Definition 2.3: Given a signature $(A=V \oplus R, \Gamma)$, and a subset $V' \subseteq V$, a V' -command F maps every attribute $a \in V'_s$ to a term $F(a)$ of sort s . \blacksquare

Commands model multiple assignments. The term $F(a)$ denotes the value that is assigned to a . If V' is empty (which is the case, for instance, of some communication channels), the only available command is the empty one: *skip*.

Definition 2.4: A *program* is a pair (θ, Δ) where θ is a signature (A, Γ) and Δ , the *body* of the program, is a triple (I, F, B) where

- I is a θ -proposition (constraining the initial values of the attributes);
- F assigns to every action $g \in \Gamma$ a $D(g)$ -command;
- B assigns to every action $g \in \Gamma$ a θ -proposition (its guard). \blacksquare

It is easy to recognise in this definition the basic features of parallel programs, namely guarded simultaneous assignments: each action g defines the guarded command

$$[B(g) \rightarrow \parallel_{a \in D(g)} a := F(g, a)]$$

There are, however, some distinguishing features of COMMUNITY that should be discussed: the typing and the naming of actions.

Each domain $D(g)$ consists of the attributes to which action g can make assignments. We shall also work with the dual notion, i.e. we define for every attribute $a \in V$ the set of the actions that can assign to a – $D(a) = \{g \in \Gamma \mid a \in D(g)\}$. There is a difference between the fact that an attribute a is not in the domain of action g and the fact that g performs the assignment $a := a$. The difference between these two situations is important from the point of view of concurrency within programs and will be further discussed and illustrated later on. But, anticipating the definition of the semantics of programs, the idea is that actions are allowed to occur concurrently (i.e. as part of the same event), e.g. actions that come from two program components that were put together in parallel. Hence, an action presents only a partial view of the transformation that is performed by a (global) event, namely it is concerned with only a subset of the attributes of the program. The assignment of specific domains to actions is, thus, a means of controlling the interference between different program components.

The separation between action *names* (i.e. the set Γ) and the guarded commands they execute (as given by F and B) is important for the definition of superposition and also to support interaction in the sense of IP. For the reader who is familiar with IP, we may state that action names act as *interaction names*, i.e. they establish synchronisation ("rendez-vous") points for processes. However, COMMUNITY differs from IP in that every action is a potential point of interaction. Indeed, interaction names in COMMUNITY are not global as in IP: interaction is established outside the programs, at "system configuration time", by identifying action names belonging to different component programs. Program interconnection will be discussed in the next section.

An example of a program is the following:

```

Pr ≡ read x:int
      var a:int; d:bool
      init d=false ∧ a=0
      do t : [-d ∧ x=a → d := true] [] r : [-d ∧ x≠a → a := x]

```

Intuitively, this program is capable of successively reading (action r) the value of the external attribute x , stopping (action t) whenever it consecutively reads the same value or the first value it reads is 0.

2.2 Itsemantics

In order to define the intended semantic structures for a program, we need a model for the abstract data type specification. As usual, such a model is given by a Σ -algebra \mathcal{U} , i.e. a set $s_{\mathcal{U}}$ is assigned to each sort symbol $s \in S$, and a (total) function $f_{\mathcal{U}}: s_1_{\mathcal{U}} \times \dots \times s_n_{\mathcal{U}} \rightarrow s_{\mathcal{U}}$ to each function symbol $f \in \Omega_{\langle s_1, \dots, s_n \rangle, s}$.

The semantic interpretation of programs is given in terms of transition systems:

Definition 2.5: A *transition system* $(\mathcal{W}, w_0, \mathcal{E}, \rightarrow)$ consists of

- a non-empty set \mathcal{W} (of states, or possible worlds);
- $w_0 \in \mathcal{W}$ (the initial state);
- a non-empty set \mathcal{E} (of events);
- a \mathcal{E} -indexed set of partial functions \rightarrow on \mathcal{W} (state transition performed by each event).

A θ -*interpretation structure* for a signature $\theta = (A = V \oplus R, \Gamma)$ is a triple $(\mathcal{T}, \mathcal{A}, \mathcal{G})$ where:

- \mathcal{T} is a transition system $(\mathcal{W}, w_0, \mathcal{E}, \rightarrow)$;
- \mathcal{A} is an S -indexed family of maps $\mathcal{A}_s: A_s \rightarrow (\mathcal{W} \rightarrow s_{\mathcal{U}})$;
- $\mathcal{G}: \Gamma \rightarrow 2^{\mathcal{E}}$. ■

That is to say, \mathcal{A} interprets attribute symbols as functions that return the value that each attribute takes in each state, and \mathcal{G} interprets the action symbols as sets of events – the set of the events during which the action occurs.

Notice that more than one action can take place during an event. Hence the execution model of COMMUNITY is more general than the one used for IP and UNITY. This feature is important in order to account for the independent behaviour of different components in non-strict interleaving execution models. It also accounts for the synchronisation of actions, i.e. for characterising action symbols as interaction names in the sense of IP. We shall often use \mathcal{G} to denote its dual $\mathcal{E} \rightarrow 2^\Gamma$, i.e. $\mathcal{G}(e)$ will denote the set of actions that occur during event e .

On the other hand, it is possible for no action to take place during an event. Such events correspond to environment steps, i.e. to steps performed by the other components in the system. Indeed, interpretation structures are intended to capture the behaviour of a program in the context of a system of which it is a component (*open semantics*). Hence, worlds are not identified with program states, i.e. with the values of the program attributes (V). The inclusion of such environment steps is essential for a *compositional semantics* of program configuration and interconnection, as put forward in [Barringer and Kuiper 84] in the context of the temporal specification of concurrent programs.

Because environment steps are taken into account, state encapsulation techniques, like those typical of object-oriented design, can be formalised through particular classes of interpretation structures.

Definition 2.6: A θ -interpretation structure $(\mathcal{T}, \mathcal{A}, \mathcal{G})$ for a signature $\theta = (A=V \oplus R, \Gamma)$ is called a *locus* iff, for every $a \in V$ and $w, w' \in \mathcal{W}$, if $w \xrightarrow{e} w'$ and $e \notin \mathcal{G}(g)$ for any $g \in D(a)$, then $\mathcal{A}(a)(w') = \mathcal{A}(a)(w)$. ■

That is, a *locus* is an interpretation structure in which the values of the program variables remain unchanged during events in which no action occurs that contains them in their domain. That is, given an attribute a and action g such that $g \in D(a)$, an occurrence of g will not change a unless it occurs during an event in which an action $h \in D(a)$ also occurs.

Given that worlds are global, transitions between worlds also occur at the level of the system and may imply the participation of more than one program. Hence, for instance, the fact that R-attributes can only be read cannot be modelled through the following constraint: if $w \xrightarrow{e} w'$ with $e \in \mathcal{G}(g)$ for some $g \in \Gamma$ then, for every $a \in R$, $\mathcal{A}(a)(w) = \mathcal{A}(a)(w')$. Indeed, it may happen that the transition e is a rendez-vous (synchronisation) point that involves the execution of an action of the component that contains a as a program variable. The restriction of non-assignment to R-attributes is only enforced in the definition of programs. This aspect can only be fully appreciated in the next section when program interconnection is discussed. For this reason, the semantics of action symbols as interconnection names will also be discussed in the next section.

Definition 2.7: Given a signature $\theta = (A, \Gamma)$ and a θ -interpretation structure $S = (\mathcal{T}, \mathcal{A}, \mathcal{G})$, the semantics of terms (for every sort s , term t of sort s and $w \in \mathcal{W}$, $\llbracket t \rrbracket^{S(w)} \in s_{\mathcal{U}}$ is the value taken by t in the world w) and propositions is defined as follows:

- if $a \in A_s$, $\llbracket a \rrbracket^{S(w)} = \mathcal{A}(a)(w)$
- if $c \in \Omega_{\diamond, s}$, $\llbracket c \rrbracket^{S(w)} = c_{\mathcal{U}}$
- if $f \in \Omega_{<s_1, \dots, s_n>, s}$, $\llbracket f(t_1, \dots, t_n) \rrbracket^{S(w)} = f_{\mathcal{U}}(\llbracket t_1 \rrbracket^{S(w)}, \dots, \llbracket t_n \rrbracket^{S(w)})$
- $(S, w) \models (t_1 =_s t_2)$ iff $\llbracket t_1 \rrbracket^{S(w)} = \llbracket t_2 \rrbracket^{S(w)}$
- $(S, w) \models (\phi_1 \supset \phi_2)$ iff $(S, w) \models \phi_1$ implies $(S, w) \models \phi_2$
- $(S, w) \models (\neg \phi)$ iff $(S, w) \not\models \phi$ ■

Definition 2.8: A θ -proposition ϕ is *true* in a θ -interpretation structure S , written $S \models \phi$, iff $(S, w) \models \phi$ at every state w . A proposition ϕ is *valid*, written $\models \phi$, iff it is true in every interpretation structure. ■

We can now define when an interpretation structure is a model of a program.

Definition 2.9: Given a program (θ, Δ) where $\theta = (A=V \oplus R, \Gamma)$ and $\Delta = (I, F, B)$, a *model* of (θ, Δ) is an interpretation structure $S = (\mathcal{T}, \mathcal{A}, \mathcal{G})$ for θ , such that:

- $(S, w_0) \models I$
- for every $g \in \Gamma$, $a \in D(g)$, $e \in \mathcal{G}(g)$ and $w, w' \in \mathcal{W}$ st $w \xrightarrow{e} w'$, $\mathcal{A}(a)(w') = \llbracket F(g, a) \rrbracket^{S(w)}$.
- for every $w \in \mathcal{W}$ and $g \in \Gamma$, if $e \in \mathcal{G}(g)$ and $w \xrightarrow{e} w'$ for some $w' \in \mathcal{W}$ then $(S, w) \models B(g)$.

A model is said to be a *locus* if it is a locus as an interpretation structure.

A model S is said to be *polite* iff for every $w \in \mathcal{W}$ and $g \in \Gamma$, $(S, w) \models B(g)$ implies that there is $e \in \mathcal{G}(g)$ and $w' \in \mathcal{W}$ such that $w \xrightarrow{e} w'$. ■

That is to say, a *model* of a program is an interpretation structure for its signature that enforces the assignments, only permits actions to occur when their guards are true, and for which the initial state satisfies the initialisation constraint.

Loci, as already explained, correspond to models of program behaviour in which encapsulation of local attributes is enforced.

A model is *polite* if actions are allowed to effect transitions in every world in which their guard is satisfied. This notion generalises the notion of *fairness* as used in parallel program design.

This classification of models reflects the existence of different levels of *semantics* for a program (taken as a set of models), depending on which subset of the set of its models is considered. In section 3, we shall see that these different semantics are associated with different notions of superposition (program morphism) that have been used in the literature, namely those used in [Bougé and Francez 88, Chandy and Misra 88, Kurki-Suonio and Järvinen 89, Francez and Forman 90b, Katz 93]. This means that there is no "absolute" notion of semantics for programs – it is always relative to the use one makes of programs. This corresponds to the categorical way of capturing the "meaning" of objects through the relationships (morphisms) that can be defined between them.

2.3 Equivalence between models

In order to explain the algebraic properties of the design techniques to be discussed in later sections, namely superposition, we need a notion of equivalence between models. The proposed notion is similar to the notion of bisimulations used in concurrency theory (e.g. [de Nicola 87]) and so called *zig-zags* between Kripke structures (e.g. [van Benthem 84]).

Definition 2.10: Given a signature $\theta=(A,\Gamma)$, two interpretation structures \mathcal{S} and \mathcal{S}' are said to be *equivalent* ($\mathcal{S} \sim \mathcal{S}'$) iff there exist relations $R \subseteq \mathcal{W} \times \mathcal{W}'$ and $T \subseteq \mathcal{E} \times \mathcal{E}'$ such that:

1. $\text{dom}(R)=\mathcal{W}$ and $\text{img}(R)=\mathcal{W}'$;
2. $w_0 R w_0'$;
3. if $w_1 \xrightarrow{e} w_2$ and $w_1 R w_1'$, then there is $e' \in \mathcal{E}'$ and $w_2' \in \mathcal{W}'$ st $e T e'$, $w_2 R w_2'$, $w_1' \xrightarrow{e'} w_2'$;
if $w_1' \xrightarrow{e'} w_2'$ and $w_1 R w_1'$, then there is $e \in \mathcal{E}$ and $w_2 \in \mathcal{W}$ st $e T e'$, $w_2 R w_2'$, $w_1 \xrightarrow{e} w_2$;
4. if $w R w'$ then $\mathcal{A}(a)(w)=\mathcal{A}'(a)(w')$, for every $a \in A$;
5. if $e T e'$ then $\mathcal{G}(e)=\mathcal{G}'(e')$. ■

Lemma 2.11: Given two equivalent interpretation structures \mathcal{S} and \mathcal{S}' and two states w and w' such that $w R w'$, $\llbracket t \rrbracket^{\mathcal{S}}(w)=\llbracket t \rrbracket^{\mathcal{S}'}(w')$ for every term t and, for every proposition ϕ , $(\mathcal{S},w) \models \phi$ iff $(\mathcal{S}',w') \models \phi$. ■

Proposition 2.12: Two equivalent interpretation structures \mathcal{S} and \mathcal{S}' are models of the same programs. Moreover, given any program P , \mathcal{S} is a locus iff \mathcal{S}' is a locus, and \mathcal{S} is polite iff \mathcal{S}' is polite. ■

3 Program morphisms and superposition

The concept of superposition (or superimposition) has been proposed and used as a structuring mechanism for the design of parallel programs and distributed systems [e.g. Bougé and Francez 88, Chandy and Misra 88, Kurki-Suonio and Järvinen 89, Francez and Forman 90b, Katz 93]. As used in UNITY, it can be viewed as a transformation on programs through the extension of their state space and/or their control activity while preserving their properties. As motivated in the introduction, structure preserving transformations are usually formalised in terms of (homo)morphisms between the

objects concerned, thus justifying the formalisation of superposition in terms of morphisms of programs.

3.1 Signature morphisms

Having defined programs over signatures, we first define signature morphisms as a means of relating the "syntax" of two programs:

Definition/Proposition 1: A *signature morphism* σ from a signature $\theta_1=(A_1=V_1\oplus R_1,\Gamma_1)$ to $\theta_2=(A_2=V_2\oplus R_2,\Gamma_2)$ consists of a pair $(\sigma_\alpha:A_1\rightarrow A_2, \sigma_\gamma:\Gamma_1\rightarrow\Gamma_2)$ of (total) functions such that $\sigma_\alpha(V_1)\subseteq V_2$ and, for every action $g\in\Gamma$, $\sigma_\alpha(D_1(g))\subseteq D_2(\sigma_\gamma(g))$. Program signatures and their morphisms constitute a category SIG . ■

Morphisms are intended to capture the relationship that exists between a program (system) and its parts (components). Hence, a signature morphism maps attributes of a program to attributes of the system of which it is a component, and the same for actions. Because the system "contains" the component, attributes of the component program cannot be read-attributes of the system, thus justifying the restriction $\sigma_\alpha(V_1)\subseteq V_2$. No restriction is put on R_1 because read-attributes of the component program can be attributes of another component program for the same system and, hence, elements of V_2 . The restriction over action domains just means that the type of each action is preserved by the morphism. Notice that more attributes may be included in the domain of an action via a morphism. This is intuitive because, within a system, an action of a component may be shared with other components and, hence, have a larger domain.

For simplicity, we shall omit the indexes α and γ when referring to the components of a morphism.

Signature morphisms provide us with the means for relating a program with its superpositions. However, superposition is more than just a relationship between signatures, i.e. more than "syntax". To capture its semantics, we need a way of relating the models of the two programs as well as the terms and propositions that are used to build them.

Signature morphisms define translations between the languages associated with each signature in the obvious way:

Definition 3.2: Given a signature morphism $\sigma: \theta_1 \rightarrow \theta_2$,

$$\begin{aligned} \sigma(t) &::= \sigma(a) \mid c \mid f(\sigma(t_1), \dots, \sigma(t_n)) \\ \sigma(\phi) &::= (\sigma(t_1)=\sigma(t_2)) \mid (\sigma(\phi_1)\supset\sigma(\phi_2)) \mid (\sigma(\phi_1)\wedge\sigma(\phi_2)) \mid \neg\sigma(\phi) \end{aligned} \quad \blacksquare$$

Definition 3.3: Given a signature morphism $\sigma: \theta_1 \rightarrow \theta_2$ and a θ_2 -interpretation structure $\mathcal{S}=(\mathcal{T},\mathcal{A},\mathcal{G})$, its σ -*reduct*, $\mathcal{S}\upharpoonright_\sigma$, is the θ_1 -interpretation structure $(\mathcal{T},\mathcal{A}\upharpoonright_\sigma,\mathcal{G}\upharpoonright_\sigma)$ where $\mathcal{A}\upharpoonright_\sigma(a) = \mathcal{A}(\sigma(a))$, and $\mathcal{G}\upharpoonright_\sigma(g) = \mathcal{G}(\sigma(g))$. ■

That is, we take the same transition system and interpret attribute and action symbols in the same way as their images under σ . Reducts provide us with the means for relating the behaviour of a program with that of the superposed one. The following proposition establishes that properties of reducts are characterised by translation of properties:

Proposition 3.4: Given a θ_1 -proposition ϕ and a θ_2 -interpretation structure $\mathcal{S}=(\mathcal{W},\mathcal{A},\mathcal{G})$, we have for every $w\in\mathcal{W}$: $(\mathcal{S},w) \models \sigma(\phi)$ iff $(\mathcal{S}\upharpoonright_\sigma,w) \models \phi$. ■

Readers familiar with institutions [Goguen and Burstall 92] will have recognised in this proposition the "satisfaction condition". Although the formalism that we work with in

this paper is not an institution (stricto sensu), we shall make use of many of the categorical techniques that have been popularised by institutions.

3.2 Programmorphisms

With this armory in hand, we can start analysing relationships between the features of two programs related by a signature morphism in order to identify what properties are necessary for morphisms to capture superposition.

There are several notions of superposition in the literature [Bougé and Francez 88, Chandy and Misra 88, Kurki-Suonio and Järvinen 89, Francez and Forman 90b, Katz 93], corresponding to different meanings of "preservation of the underlying program". We consider, in the first instance, the simplest form of superposition: *invasive superposition* in the sense of [Francez and Forman 90b].

Viewed as a transformation (which is the view captured by morphisms), invasive superposition requires that the functionality of the base program be preserved in terms of the assignments performed on its variables, but it allows for the guards of its actions to be strengthened. This characterisation leads to the following definition of an invasive superposition morphism:

Definition 3.5: An *invasive superposition morphism* $\sigma: (\theta_1, \Delta_1) \rightarrow (\theta_2, \Delta_2)$ is a signature morphism $\sigma: \theta_1 \rightarrow \theta_2$ such that

1. For every $g_1 \in \Gamma_1$ and $a_1 \in D_1(g_1)$, $\models_{\theta_2} B_2(\sigma(g_1)) \supset \sigma(F_1(g_1, a_1)) = F_2(\sigma(g_1), \sigma(a_1))$.
2. $\models_{\theta_2} (I_2 \supset \sigma(I_1))$.
3. For every $g_1 \in \Gamma_1$, $\models_{\theta_2} (B_2(\sigma(g_1)) \supset \sigma(B_1(g_1)))$. ■

Requirements 1 and 2 correspond to the preservation of the functionality of the base program: (1) the effects of the instructions are preserved and (2) so are the initialisation conditions. Requirement 3 allows guards to be strengthened but not to be weakened. With these requirements, it is indeed trivial to prove:

Proposition 3.6: Let $\sigma: (\theta_1, \Delta_1) \rightarrow (\theta_2, \Delta_2)$ be an invasive superposition morphism. Then, the reduct of every model of (θ_2, Δ_2) is also a model of (θ_1, Δ_1) . ■

However, it is easy to see that the reduct of loci of (θ_2, Δ_2) are not necessarily loci for (θ_1, Δ_1) . Indeed, there is nothing to prevent "old attributes", i.e. translations of attributes of θ_1 , to be changed by "new actions", i.e. actions of θ_2 that are not in the image of σ . Superposition morphisms that preserve locality are called *regulative*:

Definition 3.7: A *regulative superposition morphism* $\sigma: (\theta_1, \Delta_1) \rightarrow (\theta_2, \Delta_2)$ is an invasive superposition morphism that satisfies, for every $a_1 \in V_1$, $D_2(\sigma(a_1)) \subseteq \sigma(D_1(a_1))$. ■

The additional requirement corresponds to the locality condition: new actions cannot be added to the domains of attributes of the source program. Together with the fact that signature morphisms preserve the domains of actions, it implies that the domains of the attributes remain the same up to translation, i.e. $D_2(\sigma(a_1)) = \sigma(D_1(a_1))$ for every $a_1 \in V_1$. This condition implies the following property:

Proposition 3.8: Let $\sigma: (\theta_1, \Delta_1) \rightarrow (\theta_2, \Delta_2)$ be a regulative superposition morphism. Then, the reduct of every locus of (θ_2, Δ_2) is also a locus of (θ_1, Δ_1) . ■

As an example of a regulative superposition morphism, consider the following programs where $\phi, \psi: \text{int}, \text{int} \rightarrow \text{int}$ are operations of the underlying data type:

$ \begin{aligned} P_b &\equiv \text{var } a, b: \text{int} \\ &\text{init } a > 0 \wedge b > 0 \\ &\text{do } f : [\text{true} \rightarrow a := \varphi(a, b)] \\ &\quad \square \quad g : [\text{true} \rightarrow b := \psi(a, b)] \end{aligned} $	$ \begin{aligned} P_s &\equiv \text{var } a, b, a_o: \text{int}; \quad d: \text{bool} \\ &\text{init } a > 0 \wedge b > 0 \wedge d = \text{false} \wedge a_o = 0 \\ &\text{do } fr : [\neg d \wedge a_o \neq a \rightarrow a := \varphi(a, b) \parallel a_o := a] \\ &\quad \square \quad g : [\text{true} \rightarrow b := \psi(a, b)] \\ &\quad \square \quad t : [\neg d \wedge a_o = a \rightarrow d := \text{true}] \end{aligned} $
--	--

All the conditions in definitions 3.6 and 3.7 are satisfied by the mapping $\langle a \mapsto a, b \mapsto b, f \mapsto fr, g \mapsto g \rangle$ meaning that Δ_s is a regulative superposition of Δ_b . Notice that, according to this definition, it is possible for the "old" actions to assign to "new" (superposed) variables. For instance, fr , the image of f , assigns to the new attribute a_o . However, the new actions, like t , cannot assign to the old attributes, like a . Moreover, the guard of an old action, like f , can be strengthened.

It is easy to see that, for regulative superposition morphisms (and, naturally, for invasive ones), the reduct of a polite model of (θ_2, Δ_2) is not necessarily polite for (θ_1, Δ_1) . If, however, guards are not allowed to be strengthened, it is trivial to prove that reducts preserve politeness. Such superposition morphisms are called *spectative* in [Francez and Forman 90b]. They also correspond to the notion of superposition used in UNITY [Chandy and Misra 88].

Definition 3.9: A *spectative superposition morphism* $\sigma: (\theta_1, \Delta_1) \rightarrow (\theta_2, \Delta_2)$ is a regulative superposition morphism such that:

1. σ is injective over attributes and actions.
2. For every proposition ϕ in the language of θ_1 , if $\models_{\theta_2} (I_2 \supset \sigma(\phi))$ then $\models_{\theta_1} (I_1 \supset \phi)$.
3. For every $g_1 \in \Gamma_1$, $\models_{\theta_2} (\sigma(B_1(g_1)) \supset B_2(\sigma(g_1)))$. ■

Condition 3 now requires that guards remain unchanged and condition 2 requires that the strengthening of the initial condition be conservative, i.e. it cannot put further constraints on the initial values of the attributes of θ_1 . Injectivity of σ means that no confusion is introduced among attributes nor among actions.

Definition 3.10: Invasive, regulative and spectative superposition morphisms define categories which we shall denote by \mathcal{INV} , \mathcal{REG} and \mathcal{SPE} , respectively. ■

These three categories just differ on the morphisms. It is, however, the morphisms that characterise the structural properties of a category, meaning that the different notions of superposition have different algebraic properties.

For instance, we can prove a fundamental property of spectative superposition: it is model-expansive. This property means that spectative superposition does not change the base program, i.e., through σ , the base program is extended without affecting its underlying behaviour.

Proposition 3.11: Let $\sigma: (\theta_1, \Delta_1) \rightarrow (\theta_2, \Delta_2)$ be a spectative superposition morphism. Then, for every model \mathcal{S} of (θ_1, Δ_1) there is a model \mathcal{S}' of (θ_2, Δ_2) such that $\mathcal{S} \sim \mathcal{S}' \upharpoonright_{\sigma}$. ■

Model-expansive transformations have been identified as playing a very important role in modularity [Maibaum et al 85, Bergstra et al 90, Diaconescu et al 91]. We shall see in section 5.3 how this property suggests the definition of the notion of superposing an *observer* (or *monitor* [Katz 93]) on a base program, and how it can be used to characterise the notion of "derived attribute" or "auxiliary variable" as used in databases and programming languages.

4 Parallelcomposition

One of the advantages of working in the proposed categorical framework is that mechanisms for building complex systems out of components can be formalised through universal constructs. A general principle is given by J.Goguen in his work on General Systems Theory [Goguen 71, 73, Goguen and Ginali 78]: "given a category of widgets, the operation of putting a system of widgets together to form a super-widget corresponds to taking a colimit of the diagram of widgets that shows how to interconnect them".

In this section, we investigate the applicability of these principles to parallel program design based on superposition. Our purpose is to show that the use of superposition as a program composition operator in the sense of [Francez and Forman 90b], i.e. as a special kind of concurrent composition operation, can be formalised according to these categorical principles.

Notice that, except where explicitly mentioned otherwise, regulative superposition will be used throughout the section and the adjective regulative will be dropped.

4.1 Disjointparallelcomposition

In order to explain how colimits of program diagrams work and how they correspond to the activity of putting together a complex system out of its components and interconnections, we start by analysing the simple case of putting together two components without interconnecting them, and analyse interconnections in the following subsection.

Coproducts are the categorical construction that explains how two components can be put together in a system without any interconnection between them. Given two programs P_1 and P_2 , it consists of finding the *minimal* program $P_1 \parallel P_2$ that is a superposition of both P_1 and P_2 . Technically, the coproduct of P_1 and P_2 consists of a third program $P_1 \parallel P_2$ and two morphisms $\iota_i: P_i \rightarrow P_1 \parallel P_2$ ($i=1,2$) such that, given any other program P and morphisms $\sigma_i: P_i \rightarrow P$, there is one and only one morphism $\kappa: P_1 \parallel P_2 \rightarrow P$ such that $\iota_i \kappa = \sigma_i$. Minimality is expressed by the requirement on the existence and uniqueness of κ .

As an example, consider the two following programs:

$ \begin{aligned} P_b \equiv & \text{var } a, b : \text{int} \\ & \text{init } a > 0 \wedge b > 0 \\ & \text{do } f : [\text{true} \rightarrow a := \varphi(a,b)] \\ & [] \quad g : [\text{true} \rightarrow b := \psi(a,b)] \end{aligned} $	$ \begin{aligned} P_r \equiv & \text{read } x : \text{int} \\ & \text{var } a : \text{int}; d : \text{bool} \\ & \text{init } d = \text{false} \wedge a = 0 \\ & \text{do } t : [\neg d \wedge x = a \rightarrow d := \text{true}] \\ & [] \quad r : [\neg d \wedge x \neq a \rightarrow a := x] \end{aligned} $
---	--

The coproduct of these two programs returns the following program:

$$\begin{aligned}
 P_b \parallel P_r \equiv & \text{read } x : \text{int} \\
 & \text{var } a, b, a_0 : \text{int}; d : \text{bool} \\
 & \text{init } a > 0 \wedge b > 0 \wedge d = \text{false} \wedge a_0 = 0 \\
 & \text{do } f : [\text{true} \rightarrow a := \varphi(a,b)] \\
 & [] \quad g : [\text{true} \rightarrow b := \psi(a,b)] \\
 & [] \quad t : [\neg d \wedge x = a_0 \rightarrow d := \text{true}] \\
 & [] \quad r : [\neg d \wedge x \neq a_0 \rightarrow a_0 := x]
 \end{aligned}$$

together with the morphisms $\iota_b: P_b \rightarrow P_b \parallel P_r$ and $\iota_r: P_r \rightarrow P_b \parallel P_r$ given by $\langle a \mapsto a, b \mapsto b, f \mapsto f, g \mapsto g \rangle$ and $\langle x \mapsto x, a \mapsto a_0, d \mapsto d, r \mapsto r, t \mapsto t \rangle$, respectively.

Notice that the attribute a of P_r was renamed. Indeed, because coproducts model parallel composition without interaction, any unintended interference must be removed by

renaming the features whose names were used in both programs. This renaming is part of the coproduct construction, i.e. is enforced by the construct. That is why the coproduct comes with two morphisms connecting the components to their parallel composition: they keep track of the original names. Indeed, universal constructions in Category Theory enforce the principle that any interconnection between objects must be explicitly declared. (In the next subsection, we will show how to specify interconnections between programs.)

From the methodological point of view, this technical aspect of COMMUNITY, motivated by categorical principles, distinguishes it from other languages which, like IP and UNITY, rely on global naming to interconnect programs. Such approaches do not promote reuse as much as COMMUNITY because they rely on some "engineering omniscience" that does not enforce any separation between the activities of programming components and interconnecting them. Locality of names is intrinsic to Category Theory and it forces interconnections to be explicitly established outside the programs. Hence, the categorical framework is much more apt to support the complete separation between the structural language that describes the software architecture and the language in which the components are themselves programmed or specified, as advocated in configuration languages like those of the CONIC-family [Magee et al 89].

It is also interesting to point out that in languages which, like UNITY, adopt global naming, methodological restrictions have to be introduced extralogically, as in the *Restricted Union* rule [Chandy and Misra 88] – "a command r may be added to the underlying program provided that r does not assign to the underlying variables". Such principles are internalised (made logical) in our formalism through the universal properties of the categorical constructs.

Coproducts of programs (as all colimits of program diagrams) are computed by first determining the coproduct of the underlying signatures. Program signatures as defined in the previous sections are based on sets and functions between sets, for which coproducts compute disjoint unions [Barr and Wells 90].

Proposition 4.1: The category SIG of program signatures admits coproducts. The coproduct of two signatures $\theta_1=(A_1=V_1\oplus R_1,\Gamma_1)$ and $\theta_2=(A_2=V_2\oplus R_2,\Gamma_2)$ is given by the signature $\theta_1\|\theta_2=(A=V\oplus R,\Gamma)$ and morphisms ι_1 and ι_2 where $\langle A,\iota_{1\alpha},\iota_{2\alpha}\rangle$ is the disjoint union of A_1 and A_2 , and $\langle \Gamma,\iota_{1\gamma},\iota_{2\gamma}\rangle$ is the disjoint union of Γ_1 and Γ_2 . Because morphisms map local attributes to local attributes, $V=\iota_1(V_1)\cup\iota_2(V_2)$. Because the domains of actions are preserved, $D(\iota_i(g_i))=\iota_i(D_i(g_i))$ for every $g_i\in\Gamma_i$, $i=1,2$. ■

The resulting signature is obtained up to isomorphism. Indeed, there is not a unique way of renaming the features of the two signatures in order to avoid clashes. From the categorical point of view, any such renaming is suitable. That is why, as already pointed out, the coproduct of two objects returns not only an object but also two morphisms. These morphisms keep track of the renamings: in the example above, they trace a back to the attribute a of P_b and the attribute ao back to the attribute a of P_r . From an engineering point of view, this process can be systematised and automated.

At the level of programs, coproducts work like the union operator of UNITY applied to the programs *after* their signatures have been translated by the signature morphisms, i.e. after all conflicts have been removed:

Proposition 4.2: REG admits coproducts. A coproduct of two programs $P_1=(\theta_1,\Delta_1)$ and $P_2=(\theta_2,\Delta_2)$ is given by the program $P_1\|P_2=(\theta_1\|\theta_2,\Delta)$ and morphisms ι_1 and ι_2 obtained as follows:

- $\theta_1\|\theta_2$, ι_1 and ι_2 are a coproduct of θ_1 and θ_2 .

- $\Delta=(I,F,B)$ is computed as follows:
 - I is $\iota_1(I_1)\wedge\iota_2(I_2)$
 - for every $g_i\in\Gamma_i$ and $a_i\in D_i(g_i)$, $F(\iota_i(g_i),\iota_i(a_i))=\iota_i(F_i(g_i,a_i))$, ($i=1,2$)
 - $B(\iota_i(g_i))=\iota_i(B_i(g_i))$ for every $g_i\in\Gamma_i$ ($i=1,2$)

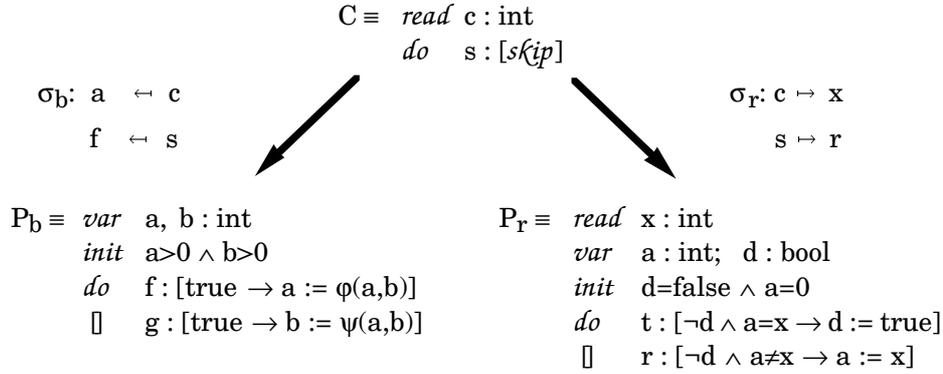
■

4.2 Parallel composition with interaction

As illustrated above, coproducts allow us to put together systems of components that run side by side with no interference between them. However, most systems we can think of are put together by *interconnecting* components. The categorical mechanisms responsible for parallel composition with interconnections are *pushouts*.

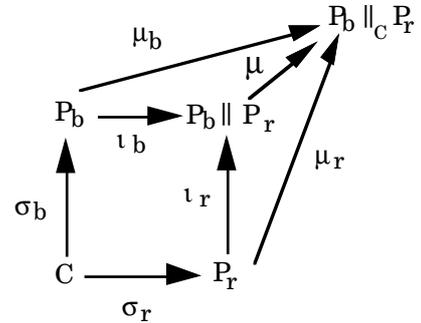
In order to illustrate these mechanisms, consider the programs P_b and P_r again. Instead of juxtaposing them, we are now interested in interconnecting them in the following way: P_r is to read the attribute a from P_b , and the actions f and r are to be synchronised so that the attribute a of P_r denotes the previous value of the attribute a of P_b .

In other words, we want to identify the attributes x and a of $P_b\parallel P_r$ as well as the actions f and r . This identification can be expressed through a (configuration) diagram



The object C and the two morphisms act as a *communication channel* between P_b and P_r : the action s of C establishes a rendez-vous (synchronisation) point and the morphisms σ_r and σ_b identify the actions of P_r and P_b that participate in this point of interaction. The same applies to attributes: the two morphisms are used to bind the external attribute x of P_r with the local attribute a of P_b .

The program that we are looking for, $P_b\parallel_C P_r$, can be characterised as providing the minimal superposition μ_b : $P_b \rightarrow P_b\parallel_C P_r$ and μ_r : $P_r \rightarrow P_b\parallel_C P_r$ of P_b and P_r such that $\sigma_b;\mu_b = \sigma_r;\mu_r$. This equation expresses the required interconnection: because $\mu_b(\sigma_b(c))$ (i.e. $\mu_b(a)$) must be equal to $\mu_r(\sigma_r(c))$ (i.e. $\mu_r(x)$), x and a must be made equal in $P_b\parallel_C P_r$ (and mutatis mutandis for f and r which are identified via s). The triple $\langle P_b\parallel_C P_r, \mu_b, \mu_r \rangle$ is called the *pushout* of σ_b and σ_r .



The resulting program and morphisms are related to the coproduct computed in the previous section by a morphism (coequaliser) μ : $P_b\parallel P_r \rightarrow P_b\parallel_C P_r$ such that $\mu_b = \iota_b;\mu$ and

$\mu_r = \iota_r; \mu$. This morphism computes quotients for the equivalence relations defined by the pairs of actions and attributes identified through the channel C and the morphisms σ_b and σ_r . The equivalence classes provide us with the required synchronisation sets, namely $\{f, r\}$, and attribute bindings, namely $\{a, x\}$. That is to say, μ imposes the required interconnections on top of the disjoint parallel composition. As expected, the unintended interference that results from name clashes is removed as seen before.

The program $P_b \parallel_C P_r$ computed by the pushout of the diagram above is, up to isomorphism, the program P_s given in section 3.2:

```

Ps ≡ var a, b, ao : int; d : bool
      init a > 0 ∧ b > 0 ∧ d = false ∧ ao = 0
      do fr : [¬d ∧ ao ≠ a → a := φ(a, b) ∥ ao := a]
      [] g : [true → b := ψ(a, b)]
      [] t : [¬d ∧ ao = a → d := true]

```

The morphisms returned by the pushout are $\langle a \mapsto a, b \mapsto b, f \mapsto fr, g \mapsto g \rangle: P_b \rightarrow P_s$ and $\langle x \mapsto a, a \mapsto ao, d \mapsto d, r \mapsto fr, t \mapsto t \rangle: P_r \rightarrow P_s$.

The synchronisation set $\{f, r\}$ is represented in this program through the *joint action* fr . Notice that its guard is given by the conjunction of the translations of the guards of b and r and that it performs the multiple assignment that consists of the local assignments of b and r . The binding of the attributes results in the attribute a .

With generality, we can prove:

Proposition 4.3: \mathcal{REG} admits pushouts. A pushout of two morphisms $\sigma_1: (\theta, \Delta) \rightarrow (\theta_1, \Delta_1)$ and $\sigma_2: (\theta, \Delta) \rightarrow (\theta_2, \Delta_2)$ is given by the program $(\theta_1 \perp_\theta \theta_2, \Delta')$ and morphisms $\mu_1: (\theta_1, \Delta_1) \rightarrow (\theta_1 \perp_\theta \theta_2, \Delta')$ and $\mu_2: (\theta_2, \Delta_2) \rightarrow (\theta_1 \perp_\theta \theta_2, \Delta')$ obtained as follows:

- $\theta_1 \perp_\theta \theta_2$, μ_1 and μ_2 are a pushout of σ_1 and σ_2 as signature morphisms. Because signatures are pairs of sets, pushouts of signatures compute amalgamated sums, i.e. $\theta_1 \perp_\theta \theta_2 = (A', \Gamma')$ where A' is the amalgamated sum of A_1 and A_2 relative to A , and Γ' is the amalgamated sum of Γ_1 and Γ_2 relative to Γ . The morphisms σ_1 and σ_2 perform the amalgamation. Because morphisms map local attributes to local attributes, $V' = \mu_1(V_1) \cup \mu_2(V_2)$. Because domains of attributes are preserved, $D'(\mu_i(a_i)) = \mu_i(D_i(a_i))$ for every $a_i \in V_i$, $i=1,2$.
- $\Delta' = (I', F', B')$ is computed as follows; let $\mu: (\theta_1 \perp_\theta \theta_2) \rightarrow (\theta_1 \perp_\theta \theta_2)$ be the morphism given by the coequaliser;
 - I' is $\mu_1(I_1) \wedge \mu_2(I_2)$
 - $F'(\mu(g), \mu(a)) = \mu_j(F_j(g_j, a_j))$ for some $a_j \in D_j(g_j)$, $\mu(g) = \mu_j(g_j)$, $\mu(a) = \mu_j(a_j)$, $i, j=1$ or 2 .
 - for $i=1,2$, $B'(\mu_i(g_i)) =$

$$\bigwedge \{ \mu_j(B_j(g_j)) \mid \mu_i(g_i) = \mu_j(g_j), j=1, 2 \} \wedge$$

$$\bigwedge \{ \mu_i(F_i(g_i, a_i)) = \mu_j(F_j(g_j, a_j)) \mid a_i \in D_i(g_i), a_j \in D_j(g_j), \mu_i(g_i) = \mu_j(g_j), \mu_i(a_i) = \mu_j(a_j), j=1, 2 \}$$

The effects of a pushout can be summarised as follows:

- attributes are bound as specified by the attributes of the middle program (channel) and morphisms; in particular, and as illustrated, read attributes of one of the components can be bound with local attributes of the other component – the resulting attribute is local to the parallel composition;
- actions are synchronised according to the rendez-vous points established by the actions of the middle program (channel) and morphisms; the resulting joint actions have the following properties:
 - their domain is the union of the domains of the joined actions;
 - they perform the parallel composition of the assignments of the joined actions;

- if the interconnecting morphisms are injective (which is usually the case), they are guarded by the conjunction of the guards of the joined actions; otherwise they have to reflect the interference between assignments that is generated locally (see discussion below);
- the initialisation condition of the resulting program is given by the conjunction of the initialisation conditions of the component programs.

Notice that the choice of an arbitrary pair $\langle g_j, a_j \rangle$ in the equivalence classes of g and a for the definition of $F(\mu(g), \mu(a))$ is allowed because the resulting program is obtained only up to isomorphism.

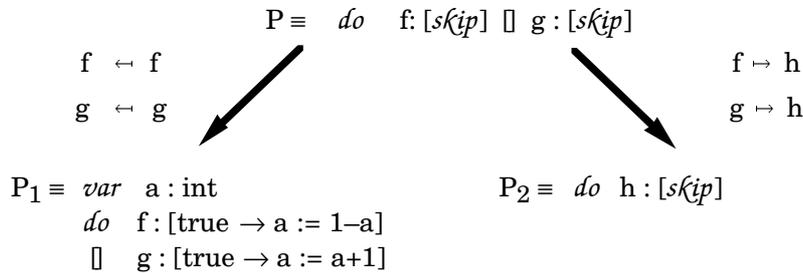
The simplification induced by injective morphisms needs more justification:

Proposition 4.4: Consider a pushout diagram as above. If σ_1 and σ_2 are injective then

1. μ_1 and μ_2 are also injective;
2. for every $g_i \in \Gamma_i$, if $\mu_j(g_j) = \mu_i(g_i)$ for $g_j \in \Gamma_j$, $j \neq i$, $B'(\mu_i(g_i)) = \mu_i(B_i(g_i)) \wedge \mu_j(B_j(g_j))$; otherwise, $B'(\mu_i(g_i)) = \mu_i(B_i(g_i))$. ■

This property and the simplification that it induces on pushouts are not surprising because, if the interconnections are established only across components, then the encapsulation mechanism of regulative superposition prevents interference between the assignments of shared actions over shared attributes.

What is, perhaps, more surprising is the use of non-injective morphisms in pushouts! Methodologically speaking, such situations arise when one forces synchronisation within a component by interconnecting two of its actions to the same action of the other component. If we think of circuits, this can be achieved by "bridging" two ports of the same component (which, of course, may end up producing a short circuit if we are not careful!). This is the case of the following configuration diagram:



The resulting program is, up to isomorphism:

$$P' \equiv \begin{array}{l} \text{var } a : \text{int} \\ \text{do } fgh : [a=0 \rightarrow a := 1] \end{array}$$

Indeed, because the interconnection synchronises f with h and g with h , f and g are also synchronised, implying that the guard of the joint action must guarantee that the assignments are compatible, i.e. it is given by $(1-a) = (a+1)$ which is equivalent to $(a=0)$. Notice the danger of "short circuit" should the terms that f and g assign to a be "incompatible", i.e. not identifiable. Such cases do not imply that there is no resulting program (pushouts always exist), but rather that the joint action is never enabled.

Because the "typical" use of pushouts in system configuration is made for injective morphisms, we shall call a diagram $P_1 \xleftarrow{\sigma_1} P \xrightarrow{\sigma_2} P_2$ *standard* if both σ_1 and σ_2 are injective.

5 Configuration of complex systems

The previous section defined the universal constructions over programs from a mathematical point of view. In this section, we shall investigate the methodological implications of the proposed categorical approach from the point of view of typical constructions in parallel and distributed program design.

5.1 Superposing regulators over programs

The previous sections showed how the categorical formalisation of superposition captures both its use as a transformation between programs as in UNITY (morphism) and a generalised parallel composition operator in the sense of IP. In fact, the example developed in section 4.2 illustrates how pushouts can characterise the operation of superposing a *regulator* over a (closed) program. Adapting from [Francez and Forman 90b], we can define these concepts as follows:

Definition 5.1: A program (θ, P) where $\theta=(A=V\oplus R, \Gamma)$ is said to be *closed* if $R=\emptyset$. A program that is not closed is said to be *open*. ■

Definition 5.2: A diagram $(\theta_1, \Delta_1) \xleftarrow{\mu_1} (\theta, \Delta) \xrightarrow{\mu_2} (\theta_2, \Delta_2)$ defines (θ_1, Δ_1) as a regulator for (θ_2, Δ_2) iff

- (θ_1, Δ_1) is open;
- (θ_2, Δ_2) is closed;
- every read (open) attribute of θ_1 is in the image of μ_1 (i.e. is connected to an attribute of θ_2 through the communication channel). ■

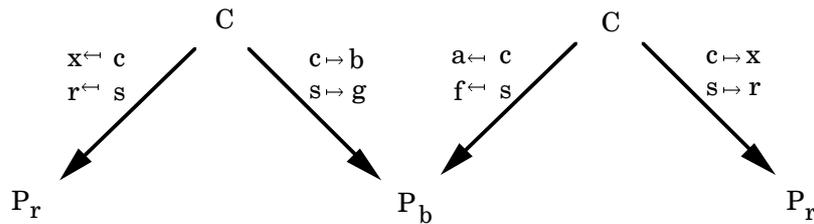
Notice that, in the example above, P_b is "closed" in the sense of IP, i.e. it has no read (open) attributes. On the other hand, P_r is "open" and the way it is interconnected with P_b makes it a *regulator* for P_b : its only read-attribute is connected to P_b through the channel.

This is, in fact, an adaptation of an example used in [Francez and Forman 90b]. By reading a , the regulator detects a pair of values of a and b such that $a=\varphi(a,b)$. When this pair is detected, and because f and r are now synchronised, the base program can no longer assign to a . Indeed, according to the properties of pushouts, the guard of the joint action is given by the conjunction of the synchronised actions.

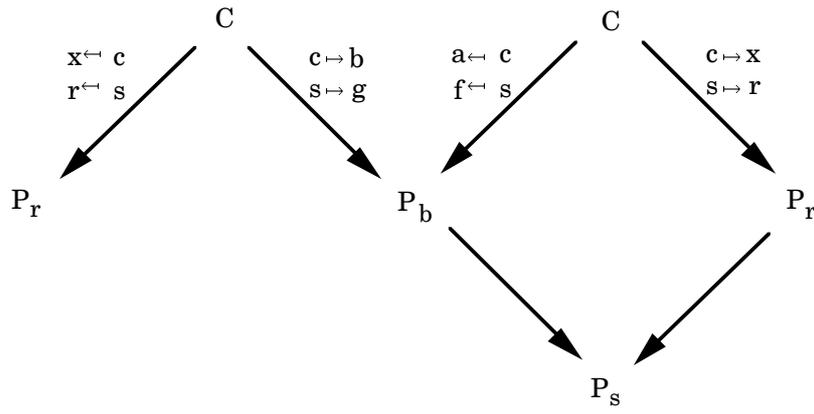
This is just an example of how the categorical techniques can provide semantics for the configuration of complex systems. Indeed, the diagrams over which we compute pushouts can be seen as specifying the configuration of the intended system in terms of its components and their interconnections. Although we have concentrated on the simple case of two components, configuration diagrams can be much more complex, allowing a system to be built from several components interconnected in many different ways.

For instance, we might like to superpose another regulator over P_b to detect a state in which $b=\psi(a,b)$. The situation is entirely symmetrical to the previous one. So, it should be possible to use another instance of the same regulator P_r and of the same channel C but using morphisms that connect the channel to b and g instead of a and f as before.

In the categorical approach, creating another instance of a program is simply achieved by adding another node to the configuration diagram and labelling it with the same program. Programs behave as *types* and nodes of the diagram as *instances*, very much in the same sense that in a programming language we may declare several variables of the same type. Hence, the configuration diagram that we are looking for is the following:

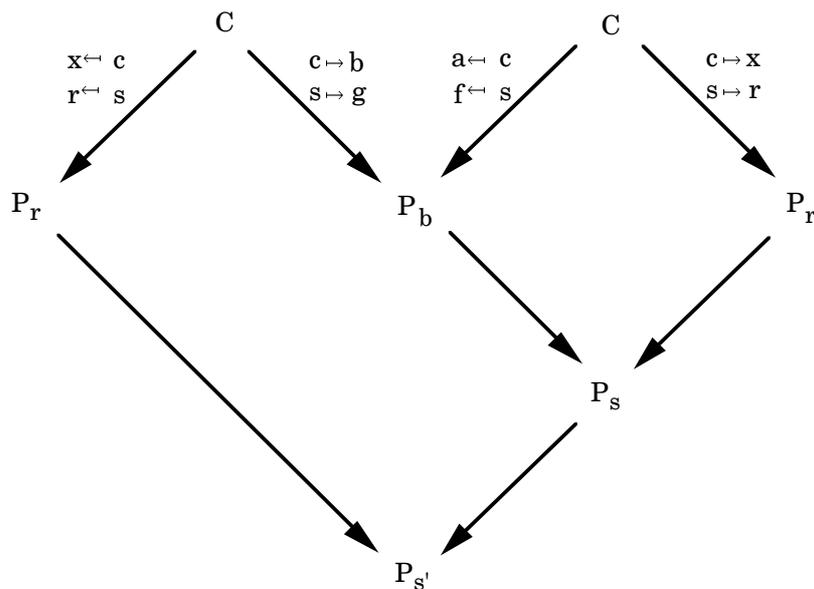


The generalisation of the pushout operation to complex diagrams like this one is called a *colimit* and a category that admits (finite) colimits is said to be (finitely) *cocomplete*. Actually, colimits can be computed through a sequence of pushouts. For instance, in the diagram above, we can compute the first pushout as before:



Indeed, we might have interconnected the second regulator directly over the previous superposition P_s to obtain the same result: the order in which the pushouts are performed is not relevant. In fact, it is better to identify the system with its configuration diagram (as suggested in [Goguen 71, 73]) and identify the evolution of the structure of the system with that of its configuration diagram. Hence, in this sense, the categorical approach supports incremental design.

The second pushout can now be computed:



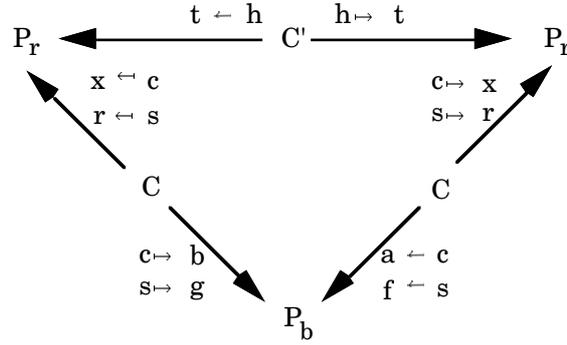
The program $P_{s'}$ is the result of the double superposition. It is isomorphic to:

```

var a, b, ao, bo : int;
    ad, bd : bool
init a>0 ∧ b>0 ∧ ad=false ∧ bd=false ∧ ao=0 ∧ bo=0
do fr : [¬ad ∧ ao≠a → a := φ(a,b) || ao := a]
  [] gr : [¬bd ∧ bo≠b → b := ψ(a,b) || bo := b]
  [] ft : [¬ad ∧ ao=a → ad := true]
  [] gt : [¬bd ∧ bo=b → bd := true]

```

This program now detects both situations in which $b=\psi(a,b)$ and situations in which $a=\phi(a,b)$. However, it does not necessarily detect a situation in which both $a=\phi(a,b)$ and $b=\psi(a,b)$. In order to achieve that, we need to synchronise ft and gt , i.e. the actions that detect the local fixpoints. This can be done by adding another communication channel to the configuration diagram below, where $C' \equiv do\ h:[skip]$:



The colimit of the configuration diagram provides a program isomorphic to:

```

var a, b, ao, bo : int;
    ad, bd : bool
init a>0 ∧ b>0 ∧ ad=false ∧ bd=false ∧ ao=0 ∧ bo=0
do fr : [¬ad ∧ ao≠a → a := φ(a,b) || ao := a]
  [] gr : [¬bd ∧ bo≠b → b := ψ(a,b) || bo := b]
  [] stop : [¬ad ∧ ao=a ∧ ¬bd ∧ bo=b → ad := true || bd := true]

```

This program now terminates when it detects a situation in which $(a,b)=(\phi(a,b),\psi(a,b))$.

The following results states that these operations can be performed over any finite configuration diagram:

Proposition 5.3: \mathcal{REG} is finitely cocomplete. ■

5.2 Design principles enforced by categories

Summarising, \mathcal{REG} supports an incremental program design discipline by allowing us to interconnect programs to form complex systems. It also supports a discipline of reuse in the sense that programs can be developed independently and interconnected at system configuration time. The use of local names, as opposed to the usual approach of a global name space, is essential to support such a degree of reusability and incrementality. The resulting systems are also *structured* because they are connected to their components (to their configuration diagram) through the colimit morphisms.

This ability to characterise the structure of objects in terms of relationships (morphisms) with other objects and to define operations of composition that preserve that structure is one of the reasons that make the categorical framework so useful for formalising disciplines of decomposition and organisation of systems into components. That is,

choosing a particular notion of morphism, we define a way of establishing relationships between objects and, hence, of structuring our world according to the components that these relationships allow us to identify.

Indeed, one of the basic principles of the categorical approach [Goguen 91] is that, for every notion of structure, there is a corresponding notion of transformation (morphism) that preserves that structure. For instance, with respect to \mathcal{REG} , one of the structural notions enforced is encapsulation of local state (attributes): the fact that morphisms are required to preserve the locality of program attributes implies that any operation on programs defined, like colimits, in terms of universal properties of morphisms, will guarantee that the attributes of the component programs remain local.

In this sense, we can claim that categories can be used to formalise program design disciplines. By changing from one category to another, for instance by keeping the same objects (programs) but changing the way we can interconnect them (morphisms), we obtain a different paradigm.

For instance, one might wonder how the union of two programs P_1 and P_2 in the sense of UNITY could be characterised in our setting. The union of P_1 and P_2 given by

$$\begin{array}{ll} P_1 \equiv \text{var } a, b : \text{int} & P_2 \equiv \text{var } a, c : \text{int} \\ \text{do } f_1 : (p \wedge p_1 \rightarrow a, b := 1, a+1) & \text{do } f_2 : (p \wedge p_2 \rightarrow a, c := 1, a-1) \\ \square \quad g_1 : (\neg p_1 \rightarrow b := 1) & \square \quad g_2 : (\neg p_2 \rightarrow c := 1) \end{array}$$

is

$$\begin{array}{ll} P_1 \square P_2 \equiv \text{var } a, b, c : \text{int} & \\ \text{do } f_1 : (p \wedge p_1 \rightarrow a, b := 1, a+1) & \\ \square \quad f_2 : (p \wedge p_2 \rightarrow a, c := 1, a-1) & \\ \square \quad g_1 : (\neg p_1 \rightarrow b := 1) & \\ \square \quad g_2 : (\neg p_2 \rightarrow c := 1) & \end{array}$$

Clearly, we do not have (regulative) superposition morphisms from P_1 and P_2 into $P_1 \square P_2$. Indeed, for instance, f_2 assigns to the attributes of P_1 thus violating the locality condition. Hence, we are not able to obtain $P_1 \square P_2$ from P_1 and P_2 in the context of \mathcal{REG} , i.e. \mathcal{REG} does not provide us with the right notion of "structure" for explaining arbitrary union. We have to switch to *invasive* superposition morphisms, i.e. move to the category \mathcal{INV} .

5.3 Observer and modularity

We have already discussed the role of regulative superposition as a mechanism for building complex systems out of components. Spectative superposition also plays a very important role in program development. Indeed, in this section, we show how spectative superposition morphisms satisfy algebraic properties that have been recognised as the source of "modularity" in program development [Bergstra et al 90, Diaconescu et al 91, Maibaum et al 85].

We have already seen in section 3.2 that spectative superposition morphisms are *model expansive*. That is, by means of spectative superposition, a program can be extended without affecting its underlying behaviour. The following proposition shows that spectative superposition is preserved by pushouts i.e. by program composition:

Proposition 5.4: Given a standard configuration diagram $(\theta_1, \Delta_1) \xleftarrow{\mu_1} (\theta, \Delta) \xrightarrow{\mu_2} (\theta_2, \Delta_2)$, i.e. μ_1 and μ_2 are injective, if μ_1 is spectative and $(\theta_1, \Delta_1) \xrightarrow{\sigma_1} (\theta', \Delta') \xleftarrow{\sigma_2} (\theta_2, \Delta_2)$ is a pushout of that diagram, σ_2 is also spectative. ■

From a methodological point of view, this proposition suggests how to superpose an *observer* (or *monitor* in the sense of [Katz 93]) over a base program.

Definition 5.5: A configuration $(\theta_1, \Delta_1) \xleftarrow{\mu_1} (\theta, \Delta) \xrightarrow{\mu_2} (\theta_2, \Delta_2)$ defines (θ_1, Δ_1) as an *observer* of (θ_2, Δ_2) iff

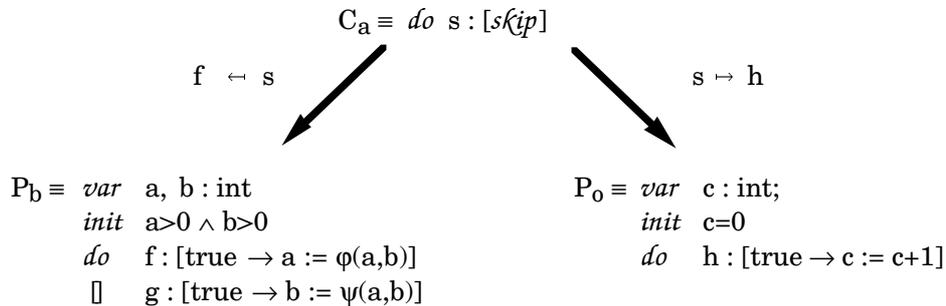
- μ_1 is spectative;
- μ_1 is surjective on actions;
- μ_1 is surjective on read attributes. ■

As a result of proposition 5.4, and of the fact that, in a pushout of sets and functions, the morphism opposite a surjective morphism is also surjective, the system obtained by superposing an observer over a base program returns a spectative extension of the base program with only new program attributes. Because there are no new actions, this means that only new ways of observing the state of the underlying program are introduced. Because the resulting system is a spectative superposition of the base program, this means that no new behaviour is being induced on the base program. This construction also corresponds to what is usually known in programming as addition of "auxiliary variables", or "derived attributes" in databases.

As an example, assume that we would like to count the number of assignments to a that is necessary to reach the fixpoint. A program that counts the number of times an action occurs is given by:

```
P0 ≡ var c : int;
      init c=0
      do h : [true → c := c+1]
```

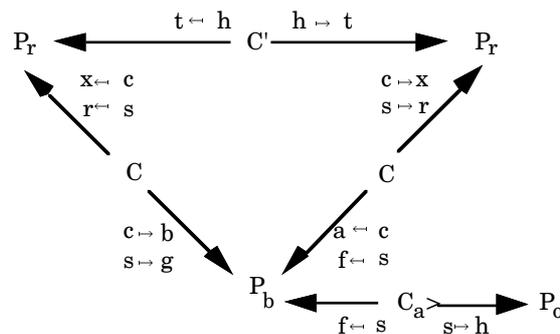
Hence, all we need to do is to connect P_0 to P_b by synchronising incrementing c with f :



The pushout of this diagram gives us the program

```
var a, b, c : int;
init a > 0 ∧ b > 0 ∧ c = 0
do fh : [true → a := φ(a, b) || c := c + 1]
□ g : [true → b := ψ(a, b)]
```

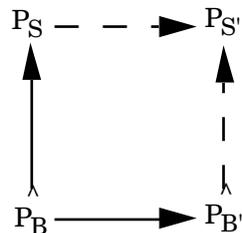
which is a spectative superposition of P_b . When incorporated within the given system,



it counts the number of assignments to a necessary to reach the fixpoint, as required.

One of the main purposes of this construction is to introduce new attributes that may account for the observations that are required by the specification of some intended system. The ability to reuse an existing piece of software (program) to satisfy a specification should allow for both the superposition of a regulator to tune the behaviour of the underlying program to the behavioural requirements of the specification, and the superposition of an observer over the regulator+program system to account for the state observations required by the specification.

The importance of proposition 5.4 is that, given such a spectative superposition P_S of a base program P_B , if P_B is independently extended to $P_{B'}$ (e.g. as a result of superposing a regulator) then there is a canonical spectative superposition $P_{S'}$ of $P_{B'}$ that provides for the observations added to P_B through P_S .



This property, called the modularisation property in [Maibaum et al 85, Veloso and Maibum 95], implies that any spectative superposition of a program is reflected in a unique way on any system of which the program is a component. Hence, it is possible to identify a system with its configuration diagram as done before in the context of regulative superpositions. That is to say, in the interconnection of P_0 as done above, the order in which the superpositions are made, including the spectative one, is immaterial. This means that the superposition of regulators and of monitors "commutes", i.e. both configuration techniques can be used as part of an incremental development process. We can superpose a monitor over a base program and later on superpose a regulator over the same base program without affecting the "status" of the first extension as a spectative superposition.

6 Concluding remarks

In this paper, we have shown how concepts and techniques for parallel program design can be formalised in a categorical framework. The perceived benefits of this effort are as follows.

First, the categorical formalisation showed how languages like UNITY and IP can be enhanced so as to make programs more open and, thus, support a discipline of modular and incremental system development that promotes reuse. Indeed, the idea that every interconnection between objects has to be made explicit, forces programs to be developed without explicit references to specific components of the system, i.e. interconnections have to be explicitly established outside the programs. Hence, the categorical framework is much more apt to support the explicit definition of the architecture of the intended system as a configuration of interconnected components, a discipline that has been advocated in the area of Distributed Systems, namely through configuration languages like those of the CONIC-family [Magee et al 89].

In this respect, we should stress that categorical formalisations of other paradigms exist, namely object-based ones, in which attributes cannot be read directly but only via actions explicitly included in the signature [Fiadeiro and Maibaum 92], i.e. supporting local fully private attributes. Furthermore, it is clear that programs as used in the paper are not the right structures on which to base system design. More general design structures are usually necessary that provide for the ability to define interfaces and hide features, support inheritance, etc. We have shown in [Fiadeiro and Maibaum 96] how such design structures can be formalised in the proposed categorical framework using temporal specifications. The adaptation to the category of COMMUNITY programs should be straightforward.

Second, it helped to clarify the nature of superposition and its role in program development. On the one hand, we showed how the two known aspects of superposition, as a transformation as used in UNITY and as a generalised parallel composition operator as in IP, can be unified in a natural way: the transformation is captured by the morphism and the operator results from the colimit construction. On the other hand, the algebraic properties of different notions of superposition were clarified. In particular, it was shown that spectative superposition is model expansive and that spectative morphisms are preserved by pushouts, a property that we showed to have important methodological consequences as already proved in other contexts [Maibaum et al 85, Bergstra et al 90, Diaconescu et al 91]. However, more general notions of superposition exist [Back and Sere 92, Butler 93] based on the use of invariants to relate the "new" and the "old" features. We intend to investigate how this more general notion can be incorporated in the categorical framework.

Third, it showed that the modularisation and composition techniques captured by superposition can be seen as instances of more general principles that apply not only to programs but also to specifications and mathematical models of system behaviour, including that of physical components. As shown in [Fiadeiro and Maibaum 95], the proposed categorical approach provides us with a natural framework to relate not only the different kinds of objects (programs, specifications, abstractions of behaviours, etc) that are intrinsic to the variety of formalisms present during software development but also, and more importantly, the structuring principles that are implied by each formalism. In particular, by working with a category of temporal logic specifications as defined in [Fiadeiro and Maibaum 92], we can formalise the relationship of satisfaction/realisation between programs and specifications in functorial terms [Fiadeiro and Maibaum 95], and distinguish between several degrees of *compositionality* according to the algebraic properties that the satisfaction relation satisfies [Fiadeiro and Maibaum 95, Fiadeiro 96].

Acknowledgments

We would like to thank our colleagues Félix Costa and Kevin Lano for the many discussions that we had on the topics of the paper, and the referees for many challenging comments. Special thanks are due to Georg Reichwein with whom the formalisation of superposition was initiated.

References

- [Back and Kurki-Suonio 88]
R.Back and R.Kurki-Suonio, "Distributed Cooperation with Action Systems", *ACM TOPLAS* 10(4), 1988, 513-554.
- [Back and Sere 92]
R.Back and K.Sere, "Superposition Refinement of Parallel Algorithms", in *FORTE'91*, North-Holland 1992, 475-493.
- [Barr and Wells 90]
M.Barr and C.Wells, *Category Theory for Computing Science*, Prentice-Hall International 1990.
- [Barringer and Kuiper 84]
H.Barringer and R.Kuiper, "Hierarchical Development of Concurrent Systems in a Temporal Framework", in S.Brookes, A.Roscoe and G.Winskel (eds) *Seminar on Concurrency*, LNCS 197, Springer-Verlag 1984, 35-61.
- [Bergstra et al 90]
J.Bergstra, J.Heering and P.Klint, "Module Algebra", *Journal of the ACM* 37(2), 1990, 335-372.
- [Bougé and Francez 88]
L.Bougé and N.Francez, "A Compositional Approach to Superimposition", in *Proc. 15th ACM Symposium on Principles of Programming Languages*, ACM Press 1988, 240-249.
- [Burstall and Goguen 77]
R.Burstall and J.Goguen, "Putting Theories together to make Specifications", in R.Reddy (ed) *Proc. Fifth International Joint Conference on Artificial Intelligence*, 1977, 1045-1058.
- [Butler 93]
M.Butler, "Refinement and Decomposition of Value-Passing Action Systems", in E.Best (ed) *CONCUR'93*, LNCS 715, Springer-Verlag 1993, 217-232.
- [Chandy and Misra 88]
K.Chandy and J.Misra, *Parallel Program Design – A Foundation*, Addison-Wesley 1988.
- [Costa et al 92]
F.Costa, A.Sernadas, C.Sernadas and H.-D.Ehrich, "Object Interaction", in I.Havel and V.Koubek (eds) *Mathematical Foundations of Computer Science'92*, LNCS 629, Springer-Verlag 1992, 200-208.
- [de Nicola 87]
R. de Nicola, "Extensional Equivalences for Transition Systems", *Acta Informatica* 24, 1987, 211-237.
- [Diaconescu et al 91]
R.Diaconescu, J.Goguen and P.Stefaneas, "Logical Support for Modularisation", in H.Huet and G.Plotkin (eds) *Proc. 2nd BRA Logical Frameworks Workshop*, Edinburgh 1991.
- [Ehrich et al 91]
H.-D.Ehrich, J.Goguen and A.Sernadas, "A Categorical Theory of Objects as Observed Processes", in J.deBakker, W.deRoever and G.Rozenberg (eds) *Foundations of Object-Oriented Languages*, LNCS 489, Springer Verlag 1991, 203-228.
- [Ehrig and Mahr 85]
H.Ehrig and G.Mahr, *Fundamentals of Algebraic Specification 1: Equations and Initial Semantics*, Springer-Verlag 1985.
- [Fiadeiro 96]
J.Fiadeiro, "On the Emergence of Properties in Component-based Systems", in M.Wirsing and M.Nivat (eds) *Proc. AMAST'96*, LNCS 1101, Springer-Verlag 1996, 421-443.
- [Fiadeiro and Costa 95]
J.Fiadeiro and F.Costa, "Institutions for Behaviour Specification", in E.Astesiano, G.Reggio and A.Tarlecki (eds) *Recent Trends in Data Type Specification*, LNCS 906, Springer Verlag 1995, 271-289.
- [Fiadeiro and Maibaum 92]
J.Fiadeiro and T.Maibaum, "Temporal Theories as Modularisation Units for Concurrent System Specification", *Formal Aspects of Computing* 4(3), 1992, 239-272.
- [Fiadeiro and Maibaum 95]
J.Fiadeiro and T.Maibaum, "Interconnecting Formalisms: supporting modularity, reuse and incrementality", in G.E.Kaiser (ed) *Proc. 3rd Symposium on Foundations of Software Engineering*, ACM Press 1995, 72-80.

- [Fiadeiro and Maibaum 96]
 J.Fiadeiro and T.Maibaum, "Design Structures for Object-Based Systems", in S.Goldsack and S.Kent (eds) *Formal Methods in Object Technology*, Springer-Verlag, in print.
- [Francez and Forman 90a]
 N.Francez and I.Forman, "Conflict Propagation", in *IEEE Int. Conf. on Computer Languages (ICCL'90)*, IEEE Press 1990, 155-168.
- [Francez and Forman 90b]
 N.Francez and I.Forman, "Superimposition for Interacting Processes", in *CONCUR'90*, LNCS 458, Springer-Verlag 1990, 230-245.
- [Goguen 71]
 J.Goguen, "Mathematical Representation of Hierarchically Organised Systems", in E.Attinger (ed) *Global Systems Dynamics*, Krager 1971, 112-128.
- [Goguen 73]
 J.Goguen, "Categorical Foundations for General Systems Theory", in F.Pichler and R.Trapp (eds) *Advances in Cybernetics and Systems Research*, Transcripta Books 1973, 121-130.
- [Goguen 91]
 J.Goguen, "A Categorical Manifesto", *Mathematical Structures in Computer Science* 1(1), 1991, 49-67.
- [Goguen 92]
 J.Goguen, "Sheaf Semantics for Concurrent Interacting Objects", *Mathematical Structures in Computer Science* 2, 1992, 159-191.
- [Goguen and Burstall 92]
 J.Goguen and R.Burstall, "Institutions: Abstract Model Theory for Specification and Programming", *Journal of the ACM* 39(1), 1992, 95-146.
- [Goguen and Ginali 78]
 J.Goguen and S.Ginali, "A Categorical Approach to General Systems Theory", in G.Klir (ed) *Applied General Systems Research*, Plenum 1978, 257-270.
- [Katz 93]
 S.Katz, "A Superimposition Control Construct for Distributed Systems", *ACM TOPLAS* 15(2), 1993, 337-356.
- [Kurki-Suonio and Järvinen 89]
 R.Kurki-Suonio and H.Järvinen, "Action System Approach to the Specification and Design of Distributed Systems", in *Proc. 5th Int. Workshop on Software Specification and Design*, IEEE Press 1989, 34-40.
- [Magee et al 89]
 J.Magee, J.Kramer and M.Sloman, "Constructing Distributed Systems in Conic", *IEEE TOSE* 15 (6), 1989, 663-675.
- [Maibaum et al 85]
 T.Maibaum, P.Veloso and M.Sadler, "A Theory of Abstract Data Types for Program Development: Bridging the Gap?", in H.Ehrig, C.Floyd, M.Nivat and J.Thatcher (eds) *TAPSOFT'85*, LNCS 186, 1985, 214-230.
- [Sannella and Tarlecki 88]
 D.Sannella and A.Tarlecki, "Building Specifications in an Arbitrary Institution", *Information and Control* 76, 1988, 165-210.
- [Sassone et al 93]
 V.Sassone, M.Nielsen and G.Winskel, "A Classification of Models for Concurrency", in E.Best (ed) *CONCUR'93*, LNCS 715, Springer-Verlag 1993, 82-96.
- [van Benthem 84]
 J. van Benthem, "Correspondence Theory", in D.Gabbay and F.Guenther (eds) *Handbook of Philosophical Logic II*, Reidel 1984, 167-247.
- [Veloso and Maibaum 95]
 P.Veloso and T.Maibaum, "On the Modularisation Theorem for Logical Specifications", *Information Processing Letters* 53, 1995, 287-293.