Part III

FASE

Semantics of Architectural Connectors(*)

J.L.Fiadeiro and **A.Lopes**
Department of Informatics
Faculty of Sciences, University of Lisbon,
Campo Grande, 1700 Lisboa, PORTUGAL
{llf,mal}@di.fc.ul.pt

Abstract. A categorical semantics is proposed for the notion of architectural connector in the style defined by Allen and Garlan which adopts notions of parameterisation similar to those developed for Abstract Data Type specification, and adapts them to formalisms for parallel program design. We show how many of the claims made in [1] can be formally substantiated, and generalised to formalisms other than CSP. Finally, we show how the categorical formalisation lends itself to useful generalisations of the notion of connector, namely through the use of multiple formalisms in the definition of the glue and the roles.

1 Introduction

Architectural connectors have emerged as a powerful tool for supporting the description of the overall organisation of systems in terms of components and their interactions [18]. According to [1], an architectural connector (type) can be defined by a set of *roles* and a *glue* specification. For instance, a typical client-server architecture can be captured by a connector type with two roles – client and server – which describe the expected behaviour of clients and servers, and a glue that describes how the activities of the roles are coordinated (e.g. asynchronous communication between the client and the server). The roles of a connector type can be *instantiated* with specific components of the system under construction, which leads to an overall system structure consisting of components and connector instances establishing the interactions between the components.

The similarities between architectural constructions as informally described above and parameterised programming [13] are rather striking and have been recently developed in [14]. The view of architectures that is captured by the principles and formalisms used in parameterised programming is reminiscent of Module Interconnection Languages and Interface Definition Languages. This perspective is somewhat different from the one followed in the work of Allen, Garlan and other researchers in Software Architectures which focuses instead on the organisation of the *behaviour* of systems as compositions of components ruled by protocols for communication and synchronisation. As explained in [1], this kind of organisation is founded on *interaction* in the behavioural sense, which explains why formalisms like CSP and CHAM [2] are preferred to the functional flavour of equational specifications.

In this paper, we propose ourselves to show that the mathematical "technology" of

(*) This work was partially supported by the Esprit WG 8319 (MODELAGE) and through contracts PRAXIS XXI 2/2.1/MAT/46/94 (ESCOLA) and PCSH/OGE/1038/95 (MAGO).

parameterisation is also very relevant for the formalisation of architectural connectors in the interaction sense, namely when used in conjunction with recently proposed algebraic approaches to parallel program design [10], in the tradition of the categorical approach to General Systems Theory also developed by Goguen [12]. We extend the preliminary work that we presented in [9, 16], bringing together architectural principles and the categorical approach to reactive system specification and design, and focus explicitly on the semantics of the notion of formal connector by abstracting from the definition given in the language WRIGHT [1] using CSP. We show how many of the claims made in [1] can be formally substantiated, and generalised to formalisms other than CSP. Finally, we show how the categorical formalisation lends itself to useful generalisations of the notion of connector, namely through the use of multiple formalisms in the definition of the glue and the roles.

More concretely, in section 2, we propose a formalisation of the notion of architectural connector in a category of (extended) COMMUNITY programs [10]. In section 3, we abstract the structural properties of the formalisms that are necessary to support the notion of architectural connector and compare the proposed formalisation with the notion of parameterised specification. In section 4, we generalise the proposed notion of architectural connector and corresponding instantiation mechanisms by allowing the roles (the formal parameters) to be defined in a formalism that is more abstract than the one in which the glue is described.

2 Architectural Connectors in COMMUNITY

Formal approaches to software architectures in the interaction sense use languages that are typical of concurrent system specification and design like CSP and CHAM. To illustrate the categorical approach that we wish to put forward for formalising architectural connectors and their relationship to parameterisation, we will use an extension of the program design language COMMUNITY presented in [10] with non-deterministic assignments. COMMUNITY is similar to IP [11] and UNITY [5].

A COMMUNITY program P has the following structure:

$$
\begin{array}{lll}
P \equiv & \textit{data} & \langle \Sigma, \Phi \rangle \\
 & \textit{var} & V \\
 & \textit{read} & R \\
 & \textit{init} & I \\
 & \textit{do} & \mathop{[]}_{g \in \Gamma} \; g: [B(g) \rightarrow \mathop{\|}_{a \in D(g)} a := F(g,a)]
\end{array}
$$

where

- $\langle \Sigma, \Phi \rangle$ represents the data types that the program uses; to support more abstract levels of program design, we work with specifications of these data types, i.e. $\Sigma = \langle S, \Omega \rangle$ is a signature in the usual algebraic sense and Φ is a set of (first-order) axioms over Σ defining the properties of the operations; if we are working at the level of a programming language, we take $\langle \Sigma, \Phi \rangle$ to be an abstraction of the properties of the data types supported by that language;
- V is the set of local attributes (i.e. the program "variables"); each attribute is typed by a data sort in S;

- R is the set of read-only attributes used by the program (i.e. attributes that are to be instantiated with local attributes of other components in the environment); each attribute is typed by a data sort in S;
- Γ is the set of *action names*; each action name has an associated statement (see below) 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 – its *guard*;
- for every action $g \in \Gamma$ and attribute $a \in D(g)$, F(g,a) is an expression denoting a set; each time *g* is executed, *a* is assigned one of the values denoted by F(g,a), chosen in a non-deterministic way.

Definition 2.1: A *program signature* is a triple $<\Sigma,V,R,\Gamma>$ where

- Σ is a signature $<S,\Omega>$ in the usual algebraic sense [6] – S is a set (of sort symbols) and Ω is an $S^*\times S$-indexed family (of function symbols).
- V and R are S-indexed families of sets where S is the set of sorts.
- Γ is a 2^V-indexed family of sets. We denote by D(g) the type of each g in Γ (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*, i.e., $D(a)=\{g\in\Gamma : a\in D(g)\}$.

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

Definition 2.2: A *program* is a pair $<\theta,\Delta>$ where θ is a signature $<\Sigma,V,R,\Gamma>$ and Δ, the *body* of the program, is a quadruple $<\Phi,I,F,B>$ where

- Φ is a (first-order) axiomatisation of the data type;
- I is a proposition over the local attributes (V);
- F assigns to every action $g \in \Gamma$ a non-deterministic command, i.e. F maps every attribute *a* in D(g) to a set expression F(a);
- B assigns to every action $g \in \Gamma$ a proposition over the attributes (V and R). ∎

For simplicity, whenever Booleans are used as data types, we abbreviate propositions of the form (t=true) to *t*. We also denote any singleton set by its element.

A model-theoretic semantics of COMMUNITY is presented in [10] for the deterministic fragment. Its extension to non-deterministic assignments is straightforward.

As an illustration of the use of COMMUNITY for defining architectural connectors, consider the simple case of a producer-consumer architectural style. It is easy to recognise in such a connector two roles – producer and consumer – which are connected to a buffer – the glue.

The following program captures the behaviour of a bounded buffer. It can store elements of sort *elem* (which are given by the environment through its read-variable *val*), as long as there is space for them, and it can discard stored elements as long as there are such elements in the buffer.

```
program buffer is
var   b : queue(elem), size : nat
read  val : elem
init  b=empty ∧ size=0
do      get : [size>0 → b:=dequeue(b) || size:=size-1]
     [] put : [size<bound → b:=enqueue(val,b) || size:=size+1]
```

For simplicity, we have omitted the specification of the underlying data type, which must include queues and the constant *bound* of sort *integer*. The fact that queues are not readily available in programming languages as data types reinforces the suitability of COMMUNITY for more abstract levels of design.

Consider now the roles of the connector, which must define the intended behaviour of producers and consumers. For the producer, we require a program capable of successively producing new values (which are put in the local attribute *sval*) and sending them. If we do not want the role to commit (yet) to a particular way of producing new elements, we cannot fully specify the effects of the action *produce*. The *instances* of the role should be able to adopt their own discipline of production because such details of production are not relevant for the communication with the buffer. Hence, we have to choose a non-deterministic assignment. In fact, we have to choose the most non-deterministic assignment to allow for arbitrary instantiations. Intuitively, the most non-deterministic assignment is represented by the set of all possible assignments. The corresponding set-expression is the sort symbol itself.

```
program producer is
var    sval : elem, ready : bool
init   ¬ready
do     produce : [¬ready → sval:=elem || ready:=true]
    [] send : [ ready → ready:=false]
```

The consumer role can be deterministically programmed:

```
program consumer is
var    rval : elem
read   b : queue(elem)
init   true
do     receive : [true → rval:=first(b)]
```

It remains to discuss how both roles can be connected to the glue.

In the architectural description language WRIGHT [1], the roles and the glue of a connector are described as CSP processes. The connections (channels) between these different processes arise from the fact that they use the same alphabet – the same name used in the role and the glue means a synchronisation point (a channel). Because locality of names is enforced in Category Theory, COMMUNITY requires name bindings (channels) to be made explicit. Name bindings in COMMUNITY can be easily made via signature morphisms.

Definition/Proposition 2.3: Given program signatures $\theta_1=<\Sigma_1,V_1,R_1,\Gamma_1>$ and $\theta_2=<\Sigma_2,V_2,R_2,\Gamma_2>$, a *signature morphism* σ from θ_1 to θ_2 consists of a morphism between Σ_1 and Σ_2 [6] together with a pair ($\sigma_\alpha: V_1 \cup R_1 \rightarrow V_2 \cup R_2$, $\sigma_\gamma: \Gamma_1 \rightarrow \Gamma_2$) of functions such that,

1. For every $s \in S$ and for every $a \in V_{1_s}$, $\sigma_\alpha(a) \in V_{2_s}$.
2. For every $s \in S$ and for every $a \in R_{1_s}$, $\sigma_\alpha(a) \in (V_{2_s} \cup R_{2_s})$.
3. For every $a \in V_1$, $\sigma_\gamma(D_1(a))=D_2(\sigma_\alpha(a))$.

Program signatures and morphisms constitute a category *SIGN*. ∎

For instance, in the case of the connection between the producer and the buffer, we need the channel (signature)

signature channel **is**
read x:elem
do a

The morphisms that perform the required bindings are <x↦ sval, a↦ send> between *channel* and *producer*, and <x↦ val, a↦ put> between *channel* and *buffer*, meaning that the buffer reads the value of the local attribute *sval* of the producer, and the buffer and the producer synchronise in the pair <send,put>.

The intended semantics of such a connector in WRIGHT [1] is the parallel composition (in CSP) of the glue and the different roles. We have already shown [10] that parallel composition in COMMUNITY is captured through the colimits of the diagrams that depict the interconnections between the components. Hence, it seems intuitive that we take the colimit of the diagram that shows how the roles are connected to the glue for the semantics of the connector.

Indeed, program morphisms (which are also called superposition morphisms because they capture relationships between programs that are known in the literature as *superposition* or superimposition [5,11]), can be defined such that a category of programs is obtained:

Definition/Proposition 2.4: A *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. $\Phi_2 \models_{\theta_2} \sigma(\Phi_1)$.
2. For all $g_1 \in \Gamma_1, a_1 \in D_1(g_1)$,
   $\Phi_2 \models_{\theta_2} B_2(\sigma(g_1)) \supset (F_2(\sigma(g_1),\sigma(a_1)) \subseteq \sigma(F_1(g_1,a_1)))$.
3. $\Phi_2 \models_{\theta_2} (I_2 \supset \sigma(I_1))$.
4. For every $g_1 \in \Gamma_1$, $\Phi_2 \models_{\theta_2} (B_2(\sigma(g_1)) \supset \sigma(B_1(g_1)))$.

where $\models_\theta$ means validity in the first-order sense.

Programs and superposition morphisms constitute a category $\mathcal{PROG}$. ∎

Requirement 1 provides us with a morphism of data type specifications as usual. Requirements 2 and 3 correspond to the preservation of the functionality of the base program: (2) the effects of the instructions can only be preserved or made more deterministic and (3) initialisation conditions are preserved. Requirement 4 allows guards to be strengthened but not to be weakened, as in regular superposition.

Proposition 2.5: The category $\mathcal{PROG}$ is finitely cocomplete.

Basically, pushouts (which are the elementary configuration diagrams) work as follows [10]:

- actions are synchronised according to the rendez-vous points established by the actions of the channel and the 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 that the joined actions have in common;
  - they are guarded by the conjunction of the guards of the joined actions;
- the initialisation condition of the resulting program is given by the conjunction of the initialisation conditions of the component programs.

However, if channels (signatures) are used in the connections (bindings), the connector does not provide a diagram of COMMUNITY programs but of signatures. Because, as in [1], we want to obtain a program as the semantics of connectors, we have to map the channel (signature) into a program. This mapping is very easily defined because there is a straightforward way of assigning a canonical program to every channel (signature), and likewise for morphisms.

The required relationship between signatures and programs is formalised as follows:

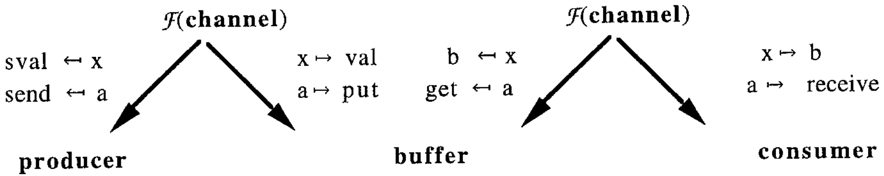
Proposition 2.6: The category $\mathcal{SIGN}$ of program signatures is fully embedded in the category $\mathcal{PROG}$ of COMMUNITY programs.

proof: consider the mapping $\mathcal{F}:\mathcal{SIGN}\rightarrow\mathcal{PROG}$ that, to every signature $<\Sigma,V,R,\Gamma>$, assigns the empty program, i.e.

$$\mathcal{F}(<\Sigma,V,R,\Gamma>) \equiv \begin{array}{ll} \textit{data} & <\Sigma,\emptyset> \\ \textit{var} & V \\ \textit{read} & R \\ \textit{init} & \text{true} \\ \textit{do} & \underset{g\in\Gamma}{[]}\ g: [\text{true} \rightarrow \underset{a\in D(g)}{\|}\ a:=s_a] \end{array}$$

It is easy to see that $\mathcal{F}$ extends to a functor and is a full embedding. ∎

Using this embedding, the configuration diagram that formalises the connector is:

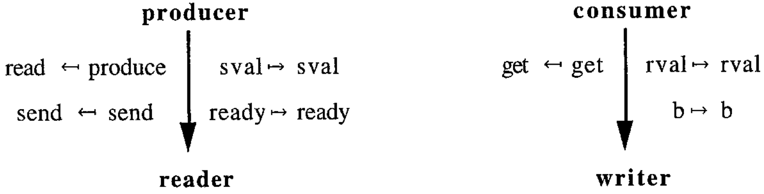


The meaning of the connector represented by this diagram is the program returned by the colimit of this diagram which, according to 2.5, always exists. This colimit corresponds to the parallel composition of *buffer*, *producer* and *consumer* with the following restrictions: *buffer* and *producer* have to synchronise on actions *put* and *send*, *buffer* and *consumer* have to synchronise on actions *get* and *receive*, the read attribute *val* of *buffer* is instantiated with *sval* of *producer* and the read attribute *b* of *consumer* is instantiated with the attribute *b* of *buffer*. This program is given, up to isomorphism, by

```
program producer-buffer-consumer is
var    b : queue(elem), rval,val : elem, size : nat, ready : bool
init   b=empty ∧ size=0 ∧ ¬ready
do       produce : [¬ready → val:=elem || ready:=true]
      [] get : [size>0 → b:=dequeue(b) || rval:=first(b) || size:=size-1]
      [] put : [size<bound ∧ ready →
                        b:=enqueue(val,b) || size:=size+1 || ready:=false]
```

What we have described are connector *types* in the sense that they can be instantiated. More concretely, the roles of a connector type can be instantiated with specific programs. In WRIGHT [1], role instantiation has to obey a compatibility requirement (expressed via a refinement relation) which in COMMUNITY is again captured via morphisms. Hence, role instantiation can be performed in much the same way as in algebraic specifications [3] through a *fitting* morphism.

An example of instantiation is the following:



with programs *reader* and *writer* defined as follows:

```
program reader is
var   val : elem, ready : bool
read  r : elem
init  ¬ready
do    read : [¬ready →
                 val:=r || ready:=true]
   [] send: [ready → ready:=false]
```

```
program writer is
var   rval : elem, ready : bool
read  b : queue(elem)
init  ready
do    get : [ready →
                 rval:=first(b) || ready:=false]
   [] write : [¬ready → ready:=true]
```

The result of the instantiation is, in WRIGHT, the parallel composition of the glue and the role instances which, in COMMUNITY, corresponds to the colimit of the diagram that extends the connector configuration with the role instances. Hence, the proposed formalisation of connectors agrees with the one given in [1] using CSP.

3 Architectural Connectors and Parameterisation

It seems obvious that the semantics of connectors given above in COMMUNITY should be able to be generalised to other formalisms. We now ask ourselves which properties of COMMUNITY are crucial for supporting software architectures.

Definition 3.1: A formalism supporting software architectures (which we shall call an *architectural school*) consists of

- a category $\mathcal{DESC}$ (that gives semantics to parallel program design);
- a full embedding $\mathcal{F}:\mathcal{CHAN}\rightarrow\mathcal{DESC}$ (where $\mathcal{CHAN}$ is a category of "channels");

such that $\mathcal{DESC}$ admits colimits of all finite diagrams in which shared objects are of the form $\mathcal{F}(C)$ with $C:\mathcal{CHAN}$. ∎

For simplicity, we usually identify the architectural school with the embedding.

Definition 3.2: Consider given an architectural school $\mathcal{F}:\mathcal{CHAN}\rightarrow\mathcal{DESC}$.

- A *connection* is a tuple $<C,G,R,\sigma:\mathcal{F}(C)\rightarrow G,\mu:\mathcal{F}(C)\rightarrow R>$ where $C:\mathcal{CHAN}$, $G,R:\mathcal{DESC}$ are called the channel, the glue and the role of the connection, respectively, and σ and μ are morphisms in $\mathcal{DESC}$.
- A *connector* is a finite set of connections with the same glue.
- The semantics of a connector is the colimit of the diagram formed by its connections. ∎

It remains to discuss role instantiation.

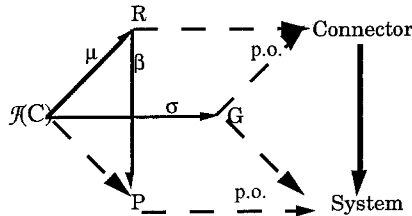
Definition 3.3: Consider given an architectural school $\mathcal{F}:\mathcal{CHAN}\rightarrow\mathcal{DESC}$.

- A *correct instantiation* of a connection $<C,G,R,\sigma:\mathcal{F}(C)\rightarrow G, \mu:\mathcal{F}(C)\rightarrow R>$ with a description P is a morphism $\beta:R\rightarrow P$ in $\mathcal{DESC}$.

- A correct instantiation of a connector is a set $\{\beta_i:R_i\rightarrow P_i\}$ of correct instantiations of its connections.
- The *resulting system* is the colimit of the diagram consisting of the morphisms $\sigma_i:\mathcal{A}(C_i)\rightarrow G$ and the compositions $\mu_i;\beta_i:\mathcal{A}(C_i)\rightarrow P_i$. ∎

Note that, in the diagram over which the colimit is taken, interconnections are made via channels, which implies that such colimits always exist.

The categorical formalisation of these architectural notions facilitates the formulation and proof of some important results. For instance, the universal properties of colimits allows us to prove that the system that results from a correct instantiation of a connector is a superposition of (refines) the (semantics of the) connector. Let us consider, for simplicity, a connector with one role.



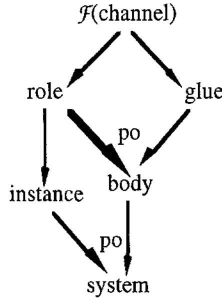
The meaning of the connector is given by the pushout of <μ,σ>. The system that results from the instantiation of the role with a component P is the pushout of <μ;β,σ>. The morphism (that results from the universal property of the first pushout) from the connector to the overall system means that the system satisfies the properties that can be inferred at the architectural level. This is a very important result because it allows the properties of connectors to be understood independently of the specific contexts (instantiations) in which they are used. This is, actually, one of the claims put forward in [1] for the ability of connectors to promote reuse.

It remains to discuss the expressive power of the proposed notion of connector, namely in relation to notions of parameterisation closer to algebraic specifications. There are several forms of parameterisation that have been proposed in the literature [17]. In this paper, we take what has been called the Clear-style [3]. Parameterisation in this simple style can be characterised by a morphism (connecting the formal parameter to the body of the parameterised specification). Like for connectors, the instantiation of the parameter is established via a (fitting) morphism from the formal to the actual parameter. The specification resulting from the instantiation is given through the pushout of the two morphisms.

Hence, the main difference between the notion of connector proposed in 3.2 and a straightforward adaptation of the notion of parameterisation seems to be that, whereas for connectors the formal parameter (role) is connected to the glue via a channel, a parameterised description is given through a morphism from the role to the body.

However, the difference between the two notions is not as big as it may seem. Indeed, if we consider the morphism that connects the role of a connector to the description that results from its semantics (pushout), we obtain a parameterised description that is equivalent to the connector in the sense that, through instantiation, they give rise to

the same systems (in the figure below, if the top diagram is a pushout, the bottom diagram is a pushout iff the outside diagram is a pushout).



But, does the converse hold as well, i.e. can every parameterised description be decomposed into an interaction mediated by a channel?

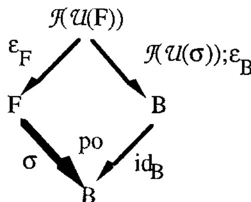
In order to understand the relevance of the question, we should analyse some of the implications of a positive answer. The main intuitive difference between connectors and parameterised descriptions and, hence, the main "novelty" of connectors, is the clear separation that is made between the definition of the "domain" of instantiation and the instantiation mechanism itself. Indeed, as shown in the definition of instantiation, the latter does not involve the role at all, just the channel (and the glue). Hence, the role has the sole purpose of defining the nature of the descriptions that can be used as instances, i.e. it defines the domain of the connector as an operator, but not its functionality. This difference is blurred in the case of parameterised descriptions: the formal parameter plays both roles.

In the context of algebraic specification, it is well known that the distinction between the two roles is supported: only the signature of the formal parameter is used in the computation of the instantiation; the axioms of the formal parameter are only used to select the correct instantiations.

The answer to the question above is positive in the case where the category $\mathcal{DESC}$ satisfies a property that we defined in [7] in the context of program synthesis.

Definition 3.4: An architectural school $\mathcal{J}:\mathcal{CHAN}\to\mathcal{DESC}$ is said to be *coordinated* (and $\mathcal{DESC}$ is said to be coordinated over $\mathcal{CHAN}$) iff $\mathcal{J}$ admits a faithful, right adjoint functor $\mathcal{U}$ for which the units are identities. ∎

In a coordinated architectural school, every parameterised description $\sigma:F\to B$ can be decomposed into a connector by taking F as role and B as glue interconnected through the channel $\mathcal{U}(F)$ and the obvious morphisms – the counit ε_F and $\mathcal{J}(\mathcal{U}(\sigma));\varepsilon_B$. Indeed, the pushout of this diagram returns $\sigma:F\to B$ and $id_B:B\to B$.



Proposition 3.5: In a coordinated architectural school, connectors and parameterised descriptions have the same expressive power in the sense that, through instantiation, they give rise to the same systems (they have the same semantics). ∎

However, even if in coordinated architectural schools the two notions are semantically equivalent, they are quite different in methodological terms. When a connector is seen as a parameterised program, the identification of the interacting parties and their coordination is not explicitly specified in the sense that it may not be possible to abstract a glue that corresponds to the minimal control mechanisms that are necessary to superpose to the role in order to obtain the body.

Indeed, it may not be possible to identify the interactions implicitly defined in the body and isolate them in an independent description and channel. Thus, it is no longer possible to claim that a connector describes an interaction between (independent) components. The separation between the role, the glue and their interaction through a channel is an intrinsic part of the notion of architectural connector and, hence, even if equivalent, the definition given in 3.2 carries more meaning.

The adjective *coordinated* is being used because the ability to provide such a clean separation between individual components and their interaction through channels is typical of coordination models and languages [4].

However, many of the formalisms we know are coordinated in this sense:

Proposition 3.6: Let $\mathcal{THEO}$ and $\mathcal{SIGN}$ be the categories of theories and signatures of an institution [15]. The free functor $\mathcal{F}:\mathcal{SIGN}\rightarrow\mathcal{THEO}$ that generates the empty theory over every signature defines a coordinated architectural school. ∎

COMMUNITY, in its extended form, also provides a coordinated architectural school:

Proposition 3.7: The category $\mathcal{PROG}$ of COMMUNITY programs is coordinated over the underlying category $\mathcal{SIGN}$ of program signatures.

proof: consider the functor $\mathcal{F}:\mathcal{SIGN}\rightarrow\mathcal{PROG}$ defined in the proof of 2.5. We are going to prove that $\mathcal{F}$ is a left adjoint of the forgetful functor $\mathcal{V}:\mathcal{PROG}\rightarrow\mathcal{SIGN}$. Let $\theta=<\Sigma,V,R,\Gamma>$ be any program signature. Because $\mathcal{V}(\mathcal{F}(\theta))=\theta$, we can take identities for units. Hence, it remains to prove that, for any program P, if $\sigma:\theta\rightarrow\mathcal{V}(P)$ is a signature morphism, then σ defines a program morphism $\mathcal{F}(\theta)\rightarrow P$. All the conditions for a signature morphism to be a program morphism are met: the set of axioms is empty, the initialisation and the guards are universal, and for every g in Γ, for every s in S and a in D(g) of sort s, F(g,a)=s and for every expression e of sort s, $e\subseteq s$. Finaly, since $\mathcal{V}$ is trivially faithful it results that $\mathcal{PROG}$ is coordinated over $\mathcal{SIGN}$. ∎

4 Adding Abstraction to Architectural Connectors

As already mentioned, the purpose of the roles in a connector description is to impose restrictions on the local behaviour of the interacting parties. In the approach to architectural design outlined in the previous sections, this is achieved through the notion of correct instantiation: the instantiation of the roles is performed with program morphisms. As also seen above, roles do not play any part in the calculation

of the resulting system. They are only used for defining what a correct instantiation is. This separation of concerns motivates the adoption of a more abstract formalism for the specification of roles.

In this section, we will show how the choice of a specification logic to represent the roles leads to a more abstract notion of connector and we will characterise the formalisms which support this new level of abstraction in architectural design. Notice that, when in section 2 we extended COMMUNITY with non-deterministic assignments, the motivation was to allow for more underspecification in the definition of roles. That is, we were already moving in the direction of more abstract specification formalisms.

In order to distinguish the roles played by the two formalisms in the definition of abstract connectors, we will call $\mathcal{PROG}$ the category of descriptions (programs) of the architectural school in which we are working, and $\mathcal{SPEC}$ the category of specifications (e.g. the category of theories of an institution). As explained in [8], we take the relationship between programs and specifications to be given through a functor $\mathit{Spec}: \mathcal{PROG} \to \mathcal{SPEC}$. Given such a functor, the usual notion of satisfaction between programs and specifications can be generalised as follows:

Definition 4.1: A *realisation* of a specification S is a pair $<\sigma,P>$ such that $P{:}\mathcal{PROG}$ and σ is a specification morphism $S \to \mathit{Spec}(P)$. ∎

In this way, programs are allowed to have features that are not required by the specification. Intuitively, the morphism records the design decisions that lead from S to P. The notion of realisation can be extended to configurations:

Definition 4.2: Let $\mathcal{S}: I \to \mathcal{SPEC}$ be a specification diagram and $\mathcal{P}: I \to \mathcal{PROG}$ a program diagram with the same shape. Assume that, for every node i:I, $\mathcal{P}_i$ is a realisation of $\mathcal{S}_i$ through a morphism η_i. We say that $\mathcal{P}$ is a *realisation* of $\mathcal{S}$ through $<\eta_i>_{i:I}$ when, for every $f{:}i \to j$ in I, $\mathcal{S}_f;\eta_j = \eta_i;\mathit{Spec}(\mathcal{P}_f)$. ∎

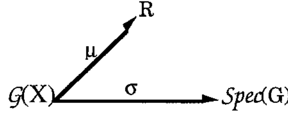
We are also going to assume that the category $\mathcal{SPEC}$ of specifications comes equipped with its own channels, i.e. with a full embedding $\mathcal{G}: \mathcal{X} \to \mathcal{SPEC}$, and that it is coordinated over $\mathcal{X}$ in the sense of 3.4. Again, the typical case will be one in which $\mathcal{SPEC}$ is the category of theories of an institution which we know is coordinated over the category of signatures.

We can now generalise definition 3.2:

Definition 4.3: Consider given two coordinated architectural schools $\mathcal{G}{:}\mathcal{X} \to \mathcal{SPEC}$ and $\mathcal{F}{:}\mathcal{Y} \to \mathcal{PROG}$ and a functor $\mathit{Spec}: \mathcal{PROG} \to \mathcal{SPEC}$.

- A *connection* is a tuple $<X,G,R,\sigma{:}\mathcal{G}(X) \to \mathit{Spec}(G),\mu{:}\mathcal{G}(X) \to R>$ where $X{:}\mathcal{X}$, $G{:}\mathcal{PROG}$, $R{:}\mathcal{SPEC}$ are called the channel, the glue and the role of the connection, respectively, and σ and μ are morphisms in $\mathcal{SPEC}$.
- A *connector* is a finite set of connections with the same glue.
- The semantics of a connector is the colimit of the diagram formed by its connections. ∎

The usefulness of these more abstract connectors depends on the ability to synthesise the interconnections between correct instantiations of the roles and the given glue. That is, given a connection



we should be able to synthesise an interconnection between the glue G and any correct instantiation of the role, i.e., any realisation of R. This means that, given $\beta: R \rightarrow \mathit{Spec}(P)$ we should be able to synthesise a channel $\mathit{Syn}(X)$ in $\mathcal{Y}$ and $\sigma': \mathcal{F}(\mathit{Syn}(X)) \rightarrow G$, $\mu': \mathcal{F}(\mathit{Syn}(X)) \rightarrow P$ in $\mathcal{PROG}$ in such a way that the given interconnection is respected, i.e., there exists $\eta_{\mathcal{G}(X)}: \mathcal{G}(X) \rightarrow \mathit{Spec}(\mathcal{F}(\mathit{Syn}(X)))$ s.t. $\sigma = \eta_{\mathcal{G}(X)}; \mathit{Spec}(\sigma')$ and $\mu; \beta = \eta_{\mathcal{G}(X)}; \mathit{Spec}(\mu')$.

The problem of synthesising interconnections was adressed in [7]. The theorem below, one of the results proved therein, determines conditions on the schools involved which guarantee that the required synthesis of interconnections is supported.

Theorem 4.4: Let $\mathcal{G}: \mathcal{X} \rightarrow \mathcal{SPEC}$ and $\mathcal{F}: \mathcal{Y} \rightarrow \mathcal{PROG}$ be two coordinated architectural schools with right adjoints $\mathcal{U}: \mathcal{SPEC} \rightarrow \mathcal{X}$ and $\mathcal{V}: \mathcal{PROG} \rightarrow \mathcal{Y}$. Let $\mathit{Spec}: \mathcal{PROG} \rightarrow \mathcal{SPEC}$ and $\mathit{Chan}: \mathcal{Y} \rightarrow \mathcal{X}$ be functors such that $\mathit{Spec}; \mathcal{U} = \mathcal{V}; \mathit{Chan}$. Then, if Chan admits a left adjoint $\mathcal{H}$ such that $\mathcal{H}; \mathcal{F}; \mathit{Spec} = \mathcal{G}$, we can synthesise interconnections – given objects S_1 and S_2 of $\mathcal{SPEC}$ interconnected via morphisms $\varphi_1: \mathcal{G}(X) \rightarrow S_1$ and $\varphi_2: \mathcal{G}(X) \rightarrow S_2$, and realisations $<\sigma_1, P_1>$, $<\sigma_2, P_2>$ of S_1 and S_2, respectively, we can synthesise an interconnection $\mu_1: Y \rightarrow P_1$ and $\mu_2: Y \rightarrow P_2$ that realises $<\varphi_1, \varphi_2>$ as follows:

- Y is $\mathcal{F}(X)$, which realises $\mathcal{G}(X)$ through the identity morphism;
- $\mu_i = \mathcal{F}(\mathcal{U}(\varphi_i; \sigma_i)); \varepsilon_{P_i}$ where ε is the counit of the adjunction between $\mathcal{F}$ and $\mathcal{V}$. ∎

Definition 4.5: A formalism supporting abstract software architectures (which we shall call an *abstract architectural school)* consists of

- two coordinated architectural schools $\mathcal{G}: \mathcal{X} \rightarrow \mathcal{SPEC}$ and $\mathcal{F}: \mathcal{Y} \rightarrow \mathcal{PROG}$;
- a functor $\mathit{Spec}: \mathcal{PROG} \rightarrow \mathcal{SPEC}$;

such there exists a functor $\mathit{Chan}: \mathcal{Y} \rightarrow \mathcal{X}$ satisfying the following properties

- $\mathit{Spec}; \mathcal{U} = \mathcal{V}; \mathit{Chan}$
- Chan admits a left adjoint $\mathcal{H}$ such that $\mathcal{H}; \mathcal{F}; \mathit{Spec} = \mathcal{G}$. ∎

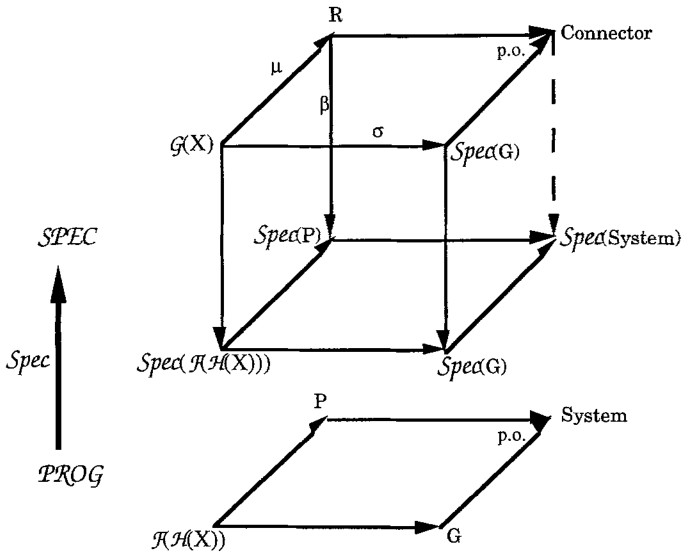
The instantiation of connectors is, as before, defined by the instantiation of their roles. Compatibility of a component with a role is, as expected, captured by the notion of realisation defined in 4.1.

Definition 4.6: Consider given an abstract architectural school $\mathcal{G}: \mathcal{X} \rightarrow \mathcal{SPEC}$, $\mathcal{F}: \mathcal{Y} \rightarrow \mathcal{PROG}$ and $\mathit{Spec}: \mathcal{PROG} \rightarrow \mathcal{SPEC}$.

- A *correct instantiation* of a connection $<X, G, R, \sigma: \mathcal{G}(X) \rightarrow \mathit{Spec}(G), \mu: \mathcal{G}(X) \rightarrow R>$ is a realisation of R.
- A correct instantiation of a connector is a set $\{\beta_i: R_i \rightarrow \mathit{Spec}(P_i)\}$ of correct instantiations of its connections.
- The *resulting system* is the colimit of the diagram synthesised according with theorem 4.4 in order to realise $<\sigma_i, \mu_i; \beta_i>$. ∎

Consider again a connector with one role (see figure below). Its meaning is given by the pushout of $<\mu, \sigma>$. Because synthesis of interconnections is supported, given an instantiation of the role with a program P, it is possible to synthesise an

interconnection between programs P and G agreeing with the interconnection <μ;β,σ> of their specifications. The system which results from the instantiation of the connector is given by the pushout of the synthesised diagram.



Theorem 4.4 applied to this situation says that the resulting system is a realisation of the connector, that is, it satisfies the properties that can be inferred at the architectural level. As stressed before, this means that the properties of connectors can be understood independently of the specific context in which they are used.

An illustration of an abstract architectural school can be given in terms of COMMUNITY as defined in section 2 and linear temporal logic. We have shown in [8] that a functor can be defined between the category of (deterministic) COMMUNITY programs and the category of temporal theories. In [7], we further showed that the two formalisms satisfy 4.4. The extension to non-deterministic programs is trivial.

5 Concluding Remarks

In this paper, we proposed a formalisation for the notion of architectural connector in the sense of [1] which adopts categorical techniques developed for the parameterisation of algebraic specifications [17]. These techniques were adapted to formalisms for parallel program specification and design, namely the language COMMUNITY [10] in the style of UNITY [5] and Interacting Processes (IP) [11]. The proposed formal notion of architectural connector consists of an object *glue* connected to a collection of *role* objects (the formal parameters) through channels. Channels were explicitly modelled through a full embedding into the category of program designs, capturing the way interconnections are established in process design languages such as CSP and IP.

The instantiation of roles was defined through (fitting) morphisms and the resulting system was defined through colimits, much in the tradition of parameterised

specifications in the Clear-style [3]. This semantics was shown to agree with the formalisation of connector given in [1] using CSP in the sense that colimits of configuration diagrams capture parallel composition of concurrent programs [10]. We also showed how the proposed formalisation fulfils the requirements stated in [1] for the ability to understand the behaviour of a connector independently of its use in specific contexts, and to reason about the compatibility between roles and instances.

Moreover, the proposed categorical formalisation of architectural connector was shown to be flexible enough to allow for more abstract notion of connector in which the glue and the roles are defined in different formalisms or languages. We studied the case in which the roles are described in a formalism that is more abstract than the glue, e.g. a specification logic (institution) for the roles and a program design language for the glue. Recent results on the ability to synthesise interconnections [7] were used to define the corresponding instantiation mechanisms. The use of temporal logic and COMMUNITY as in [8] was suggested as an example.

A comparison between the proposed notion of connector and notions of parameterised specification, in the sense of a morphism from the formal parameter (role) to a body [3], revealed that the body in parameterised specifications corresponds to the parallel composition of the glue with its roles, i.e. to the "semantics" of the connector. The converse representation of parameterised specifications into connectors was shown to be possible for categories of descriptions that are *coordinated* over the category of channels, a concept that we introduced in [7] and which captures structural properties of formalisms typical of coordination languages and models [4]. An open problem remains which consists in being able to isolate the glue from a body which intends to capture the joint behaviour of the roles interconnected to the glue, showing that there are methodological implications in the way connectors are formalised, making the approach based on the explicit identification of the glue and roles seem to be better suited for the interaction-based architectural structures in the sense that it is directly compositional on the structure of the system.

Further work is indeed needed on the relationship between architectural notions in the interaction sense [1] and the architectural notions that are intrinsic to the use of Module Interconnection and Interface Definition Languages and which have been formalised using notions of parameterisation typical of algebraic specifications [14]. The formalisation of connector that we proposed in COMMUNITY actually requires an integration of the two perspectives. For instance, the program that modelled the bounded buffer relied on a specification of queues of elements. Hence, we could say that the program buffer was also parameterised by *elem* (the data type of elements) as well as by *bound*. This kind of parameterisation serves the module-based notion of architecture and is orthogonal to the interaction-based one: the latter focuses on *control*, i.e. on the scheduling and synchronisation of the different actions, whereas the former addresses the modules that are required to provide the data context (namely the operations) in which the transformations operated by the actions are defined. We intend to further develop these relationships, namely in the context of the parameterisation mechanisms developed in [17] for program modules as algebras, as a means of providing an integrated methodology for system specification and design.

Acknowledgements

We wish to thank Tom Maibaum and Carlos Paredes for many useful discussions.

References

1. R.Allen and D.Garlan, "Formalising Architectural Connection", in *Proc.* 16th ICSE, 1994, 71-80. (See also *Formal Connectors*, CMU-CS-94-115.)
2. G.Berry and G.Boudol, "The Chemical Abstract Machine", *Theoretical Computer Science* 96, 1992, 217-248.
3. R.Burstall and J.Goguen, "The Semantics of CLEAR, a Specification Language", in *Proc. Advanced Course on Abstract Software Specification*, LNCS 86, Springer-Verlag 1980, 292-332.
4. P.Ciancarini and C.Hankin, *Coordination Languages and Models*, LNCS 1061, Springer-Verlag 1996.
5. K.Chandy and J.Misra, *Parallel Program Design - A Foundation*, Addison-Wesley 1988.
6. H.Ehrig and G.Mahr, *Fundamentals of Algebraic Specification 1: Equations and Initial Semantics*, Springer-Verlag 1985.
7. J.Fiadeiro, A.Lopes and T.Maibaum, "Synthesising Interconnections", in D.Smith and J.P.Finance (eds) *Proc. IFIP TC 2 Working Conference on Algorithmic Languages and Calculi*, Chapman Hall, in print.
8. J.Fiadeiro and T.Maibaum, "Interconnecting Formalisms: supporting modularity, reuse and incrementality", in G.E.Kaiser (ed) *Proc. 3rd Symp. on Foundations of Software Engineering*, ACM Press 1995, 72-80.
9. J.Fiadeiro and T.Maibaum, "A Mathematical Toolbox for the Software Architect", in J.Kramer and A.Wolf (eds)*Proc. 8th International Workshop on Software Specification and Design*, IEEE Computer Society Press 1996, 46-55.
10. J.Fiadeiro and T.Maibaum, "Categorical Semantics of Parallel Program Design", *Science of Computer Programming*, in print.
11. N.Francez and I.Forman, *Interacting Processes*, Addison-Wesley 1996.
12. J.Goguen, "Categorical Foundations for General Systems Theory", in F.Pichler and R.Trappl (eds) *Advances in Cybernetics and Systems Research*, Transcripta Books 1973, 121-130.
13. J.Goguen, "Principles of Parametrised Programming", in Biggerstaff and Perlis (eds) *Software Reusability*, Addison-Wesley 1989, 159-225.
14. J.Goguen, "Parametrised Programming and Software Architecture", in *Symposium on Software Reusability*, IEEE 1996.
15. J.Goguen and R.Burstall, "Institutions: Abstract Model Theory for Specification and Programming", *Journal of the ACM* 39(1), 1992, 95-146.
16. C.Paredes, J.Fiadeiro and F.Costa, "Architectural Specifications: Modeling and Structuring Behavior through Rules", in H.Kilov and W.Harvey (eds) *Object-Oriented Behavioral Specifications*, Kluwer Academic Publishers 1996, 221-240.
17. D.Sannella, S.Sokolowski and A.Tarlecki, "Toward Formal Development of Programs from Algebraic Specifications: Parameterisation Revisited", *Acta Informatica* 29, 1992, 689-736.
18. M.Shaw and D.Garlan, *Software Architecture: Perspectives on an Emerging Discipline*, Prentice Hall, 1996.