Synthesising Interconnections

J.L.Fiadeiro and A.Lopes
Department of Informatics,
Faculty of Sciences,University of Lisbon
Campo Grande, 1700 Lisboa, PORTUGAL
telephone: 351-1-7500087, fax: 351-1-7500084
{llf,mal}@di.fc.ul.pt

T.S.E.Maibaum
Department of Computing,
Imperial College of Science, Technology and Medicine
180 Queen's Gate, London SW7 2BZ, UNITED KINGDOM
telephone: 44-171-5948281, fax: 44-171-5948282
tsem@doc.ic.ac.uk

Abstract
In the context of the modular and incremental development of complex systems, viewed as interconnections of interacting components, new dimensions and new problems arise in the calculation of programs from specifications. A particularly important aspect for extending existing methods to address composite systems is the ability, given programs that realise component specifications, to synthesise the interconnections between them in such a way that the system specification is realised. Taking our cue from earlier work on General Systems Theory (Goguen, 1973) and more recent work on parallel program design (Fiadeiro and Maibaum, 1996), we discuss, characterise and provide solutions for the synthesis of interconnections using a categorical framework in which components are modelled as objects (either specifications or programs) and morphisms are used to express interconnections between components.

Keywords
Synthesis, interconnection, incremental development, complex systems.

1 INTRODUCTION

Given a class $\mathcal{SPEC}$ of specifications, a class $\mathcal{PROG}$ of programs and a relation *sat* in $\mathcal{PROG}\times\mathcal{SPEC}$, methods for the calculation of programs from specifications aim at supporting the development, for every specification S, of a program P such that P *sat* S. A subclass of such techniques are synthesis methods which produce, in a systematic, algorithmic way, a program $\mathcal{Syn}$(S) for every specification S in such a way that $\mathcal{Syn}$(S) *sat* S. Well known examples are the synthesis of communicating processes and synchronisation skeletons from temporal logic specifications (Emerson and Clarke, 1982, Manna and Wolper, 1984).

To have serious impact in today's software development environment, the methods addressing the calculation/synthesis of individual program units must be extended or supplemented to encompass multicomponent (and even hybrid) systems. Our aim in this paper is to investigate the requirements that the calculation of systems of interconnected components, rather than single, isolated programs, imposes on specifications, programs and the relationship between them.

The calculation of composite systems, viewed as interconnections of simpler, interacting components, raises new and challenging questions, namely in support of reuse and incrementality. In an incremental software development discipline, it should not be necessary to recalculate the whole system each time the specification of a new component is added. It should be possible to calculate only a program that realises the specification of the new component and its interconnections to the old system. We should be able to support situations where some components are automatically synthesised from their specifications, other components are developed from scratch, e.g. using transformational techniques, and yet other components are reused from previous developments.

The fact that, in composite systems, some of the components may correspond to human or physical agents, provides further evidence for the need to integrate program calculation methods into a modular development discipline, namely in conjunction with methods and techniques that exist for the development of various (possibly different kinds of) components (e.g. synthesis methods for hardware components or control systems).

The ability to calculate interconnections between given realisations of component specifications from the interconnections that have been provided at the specification level, seems to be the key ingredient for extending existing methods for calculating programs from specifications to address composite systems and, hence, support incremental development. Ideally, we should be able to *synthesise* interconnections, by which we mean a systematic, hopefully automatable, way of generating interconnections. The question which we propose to address in this paper is the nature of the programming and specification formalisms, and of the relationship between them, which the necessity of synthesising interconnections imposes.

Concerns about the systematic development of composite systems can be found in the literature. The work reported in (Feather, 1987, 1989, Fickas and Helm,

1992) in particular should be noted. However, these approaches are mostly concerned with the problem of *decomposing* a system specification into components, and assume a *closed system* point of view. In this paper, we take the dual point of view, i.e. we address *composition* in an *open systems* context, which we think is more appropriate for the concerns that we have expressed above, namely incrementality and reuse. Our view is also consistent with recent work on *architectural design* (e.g. Shaw and Garlan, 1995), which emphasises the importance of establishing the overall architecture of the system early on in the design process.

In order to analyse the impact that the systems view and incremental development disciplines have on calculation methods, we need a framework in which specifications, programs and the relationships between them can be formally described. The simple-minded view of classes and relations recalled in the first paragraph is clearly not appropriate for capturing systems of interconnected components. Instead, we shall adopt the categorical approach proposed by J.Goguen for General Systems Theory (e.g., Goguen, 1973, Goguen and Ginali, 1978). Capitalising on our recent work on reactive system specification (Fiadeiro and Maibaum, 1992) and parallel program design (Fiadeiro and Maibaum, 1996), we shall view both $\mathcal{PROG}$ and $\mathcal{SPEC}$ as categories in which systems of interconnected components are modelled through diagrams (in the categorical sense).

In this setting, we shall formally characterise synthesis of interconnections and formulate general conditions that guarantee the ability to synthesise interconnections. These conditions are related to the existence of adjunctions between $\mathcal{PROG}$ and $\mathcal{SPEC}$. Because the existence of adjunctions is a very strong property, we explore situations where weaker conditions apply, making synthesis of interconnections more practicable. These weaker conditions rely on stronger structural properties of the formalisms involved and can be related to properties exhibited by so called coordination languages and models (Ciancarini and Hankin, 1996). Examples will be given using propositional linear temporal logic and the parallel program design language COMMUNITY.

2 THE CATEGORICAL VIEW OF SYSTEMS

In the early 70's, J. Goguen proposed the use of categorical techniques in *General Systems Theory* for unifying a variety of notions of system behaviour and their composition techniques (Goguen, 1973, Goguen and Ginali, 1978). His approach has been summarised in a very simple but far reaching principle: "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 illustrate the application of these principles to reactive system specification and parallel program design.

2.1 Interconnecting specifications

The application of the categorical approach to reactive system specification builds on the theory of institutions developed in (Goguen and Burstall, 1992), namely on the use of theories (or theory presentations) of a logic (institution) as building blocks in the construction of structured specifications.

We shall now illustrate the approach using propositional linear temporal logic. See (Fiadeiro and Maibaum, 1992) for a similar account using first-order temporal logic.

Definition/Proposition 2.1: The linear temporal logic institution [redacted] LTL is defined as follows:

- its category of signatures is [redacted]
- the grammar f[redacted]r defin[redacted]ery signature Σ the set of *lin*[redacted]*emporal propositions* L[redacted]) as fol[redacted]

  ϕ ::= [redacted] (¬ϕ) [redacted]) | **beg** | [redacted]

  for a∈Σ. A [redacted] m f:Σ→Σ' [redacted] he [redacted]
  f: LTL(Σ)→LT[redacted]

  f(ϕ) ::[redacted])**U**[redacted]).
- the model f[redacted]
  consists of a[redacted] he [redacted] f a
  models. Give[redacted]
  M|f∈Mod(Σ) [redacted]
- the satisfacti[redacted] ows [redacted] ϕ [redacted] id [redacted]
  true in a Σ-m[redacted] we [redacted]
  – for all a∈Σ [redacted]
  – (M,i) ⊨Σ¬ϕ [redacted] ⊨Σϕ;
  – (M,i) ⊨Σϕ⊃ [redacted] (M,i) ⊨Σ [redacted]
  – (M,i) ⊨Σ**be**[redacted]
  – (M,i) ⊨ΣϕU[redacted] exis[redacted] i such that ([redacted] ⊨Σψ an[redacted] r all k such [redacted]
  i<k≤j, (M,k)[redacted].

  A Σ-proposi[redacted] ϕ is said t[redacted] true in M, [redacted] we denote by M ⊨Σϕ [redacted]
  (M,i) ⊨Σϕ fo[redacted]ry i∈ω. G[redacted] Φ⊆LTL(Σ) [redacted] ϕ∈LTL(Σ), Φ ⊨Σϕ iff [redacted]
  every Σ-model M, M ⊨Σϕ wh[redacted]er M ⊨Σψ fo[redacted]ry ψ∈Φ. ∎

Some remarks seem to be in order:

- each signature provides a set of propositional symbols (to be interpreted as the set of actions that a given component can perform);
- every signature has an associated language generated from the propositional symbols, the usual propositional connectives, the propositional constant **beg** (denoting the initial instant of time) and the temporal operator **U** ("until");
- propositions are interpreted over the natural numbers; that is, a linear, discrete temporal structure is chosen;
- the interpretation of the temporal operator **U** ("until") is different from other uses that can be found in the literature, e.g. (Goldblatt, 1987): it is "strict",

i.e. it [redacted] not include the present, and $\phi \mathbf{U} \psi$ requires ϕ to be satisfied when ψ beco[redacted]atisfied;

- the fo[redacted]ing operators can be defined as abbreviations:
  - $\mathbf{F}^{+}$[redacted]true $\mathbf{U}$ ϕ), "s[redacted]ime in the future ϕ, being neutral wrt ϕ now"
  - $\mathbf{F}\phi$ [redacted] the future ϕ, possibly now"
  - $\mathbf{G}^{+}$[redacted]$\neg \mathbf{F}^{+}(\neg\phi)$), "a[redacted]s in the future ϕ but not necessarily now"
  - $\mathbf{G}\phi$ [redacted]e future ϕ, including now"
  - $\phi\mathbf{W}$[redacted] $(\phi\mathbf{U}\psi)\vee\mathbf{G}^{+}($[redacted]$\psi)$, "$\phi$ until ψ but, possibly, never ψ"

Specifica[redacted] are taken as [redacted]ies in this institution:

Definitio[redacted]2: The objec[redacted] of specifications are the pairs (Σ,Φ), w[redacted]Σ is a signat[redacted]nd $\Phi \subseteq LTL(\Sigma)$ is closed under consequence (i.e. $\Phi \ \ {}_{\Sigma}\phi$ im[redacted]s $\phi \in \Phi$). A m[redacted]$_2$) is a signature morphism $f:\Sigma_1 \rightarrow \Sigma_2$ such that $\underline{f}(\Phi_1) \subseteq$[redacted] ∎

A specification is usuall[redacted]en through a finite set of axioms (a presentation). The specification thus prese[redacted] consists of all the consequences of the axioms, i.e. of the closure of the set [redacted]xioms. We denote by $\Phi^{\bullet}$ the closure of Φ, i.e. $\Phi^{\bullet}=\{\phi \in LTL(\Sigma) \mid \Phi \ \ {}_{\Sigma}\phi\}$.

In a specification, the signature indicates what is the language of the component being specified and the axioms correspond to the properties that the component is required to satisfy. Specification morphisms are translations between the languages of the two theory presentations such that the theorems of the first specification are translated to theorems of the second. Notice that the axioms are propositions that are required to be satisfied at every instant of time.

As an example, consider the following specification of a vending machine:

specification vending machine is
sign coin, cake, cigar, reset
axioms beg ⊃ (¬cake∧¬cigar∧¬reset) ∧ (coin ∨ (¬cake∧¬cigar∧¬reset)Wcoin)
coin ⊃ (¬coin ∧ ¬reset)W(cake ∨ cigar)
(cake ∨ cigar) ⊃ (¬cake ∧ ¬cigar ∧ ¬coin)Wreset
reset ⊃ (¬cake ∧ ¬cigar ∧ ¬reset)Wcoin
cake ⊃ (¬cigar)

That is to say, the machine is initialised so as to accept only coins. Once it accepts a coin it can deliver either a cake or a cigar (but not both). After delivering a cake or a cigar it can only be reset, after which it is ready to accept more coins.

In order to illustrate the categorical approach to composition, assume that it becomes necessary to interconnect a regulator to the vending machine in order to prevent it from selling cigars. The specification of a regulator is quite simple:

specification regulator **is**
sign trigger, action
axioms trigger ⊃ G¬action

That is to say, a regulator can perform two actions, *trigger* and *action*. After performing *trigger*, *action* can no longer occur.

It remains to show how to connect the regulator to the vending machine. Intuitively, we want *action* to be synchronised with *cigar* and *trigger* with *coin* so that the action of accepting a coin triggers the cigar option to be blocked.

In the categorical approach, interconnections are established through diagrams. For the purpose at hand, we need a communication channel between *vending machine* and *regulator*. This channel is specified as having two ports with no required properties:

specification channel **is**
sign a, b
axioms

The configuration of the required system can be given by the following diagram:

$$\begin{tikzcd} & \text{channel} \arrow[dl, "\begin{matrix}\text{coin} & \text{a}\\ \text{cigar} & \text{b}\end{matrix}"'] \arrow[dr, "\begin{matrix}\text{a} & \text{trigger}\\ \text{b} & \text{action}\end{matrix}"] & \\ \text{vending machine} & & \text{regulator} \end{tikzcd}$$

The properties of the resulting system can be obtained by the colimit of its configuration diagram:

Proposition 2 The category $\mathcal{SPEC}$ is finitely cocomplete. Its initial object is $(\emptyset,\emptyset)$ and the pushout of $f_1:(\Sigma,\Phi)\rightarrow(\Sigma_1,\Phi_1)$ and $f_2:(\Sigma,\Phi)\rightarrow(\Sigma_2,\Phi_2)$ is given by $g_1:(\Sigma_1,\Phi_1)\rightarrow(\Sigma',\Phi')$ and $g_2:(\Sigma_2,\Phi_2)\rightarrow(\Sigma',\Phi')$, where g_1 and g_2 result from the pushout of f_1 and f_2 as signature morphisms (the resulting signature Σ' is the amalgamated sum of Σ_1 and Σ_2 relative to f_1 and f_2) and $\Phi'=(\underline{f}_1(\Phi_1)\cup\underline{f}_2(\Phi_2))^\bullet$. ∎

This is a property of any institution (Goguen and Burstall, 1992). It tells us that the set of axioms of the resulting specification consists of the union of the translations of the axioms of the components. Because the union of sets of formulae has the same [illegible] approach complies with the "composition as conjunction" idea recently put forward in (Abadi and Lamport, 1993) for parallel composition of reactive systems.

The specification of the system that results from interconnecting the vending machine and the regulator as above is given below, assuming that the morphisms that connect *vending machine* and *regulator* to the new specification are the identity and (trigger coin, action cigar), respectively:

specification regulated vending machine **is**
sign coin, cake, cigar, reset
axioms $\mathbf{beg} \supset (\neg cake \wedge \neg cigar \wedge \neg reset) \wedge (coin \vee (\neg cake \wedge \neg cigar \wedge \neg reset)\mathbf{W}coin)$
$coin \supset (\neg coin \wedge \neg reset)\mathbf{W}(cake \vee cigar)$
$(cake \vee cigar) \supset (\neg cake \wedge \neg cigar \wedge \neg coin)\mathbf{W}reset$
$reset \supset (\neg cake \wedge \neg cigar \wedge \neg reset)\mathbf{W}coin$
$cake \supset (\neg cigar)$
$coin \supset \mathbf{G}\neg cigar$

2.2 Parallel program design in a categorical framework

In (Fiadeiro and Maibaum, 1996), we further showed how parallel program design in the language COMMUNITY (similar to UNITY (Chandy and Misra, 1988) and IP – Interacting Processes (Francez and Forman, 1996)) can be formalised using the same categorical techniques. The underlying computational model is also similar to Action Systems (Back ad Kurki-Suonio, 1988), but we should point out that the Action Systems approach, at least in relation to transformational development, is oriented to *decomposition* of systems into components. As explained in the introduction, our focus is on *composition*, which may explain some differences wrt Action Systems.

From a categorical point of view, objects are programs and morphisms capture superpositions. The colimit construction corresponds to a generalised parallel composition operator with synchronisation constraints (i.e. to superimposition in the sense of (Francez and Forman, 1996)).

We shall now illustrate these points by using a subset of COMMUNITY. The simplification consists in omitting data type declarations and external attributes. More precisely, we shall assume a fixed signature (S,Ω) in the usual algebraic sense (Ehrig and Mahr, 1985), 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 DT of (first-order) axioms over (S,Ω) defining the properties of the operations.

A COMMUNITY program P in this simplified version has the following structure:

$$
\begin{array}{lll}
P \equiv & \textit{var} & V \\
 & \textit{init} & I \\
 & \textit{do} & \mathop{[]}\limits_{g\in\Gamma} \; g: [B(g) \;\rightarrow\; \mathop{\|}\limits_{a\in D(g)} \; a:=F(g,a)]
\end{array}
$$

where

- V is the set of attributes used by the program (i.e. the program "variables"); 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 – 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 attribute $a\in D(g)$, F(g,a) is a term of the same sort as a.

As an example, consider the following program corresponding to the vending machine specified in the previous section (the correspondence will be formalised later on):

```
program vending machine is
var   ready, served_cake, served_cigar : bool
init  ready ∧ (served_cake ∨ served_cigar)
do    coin : [ready ∧ (served_cake ∨ served_cigar)
                 → ready:=false || served_cake:=false || served_cigar:=false]
      [] cake : [¬ready ∧ ¬(served_cake ∨ served_cigar)
                 → served_cake:=true || served_cigar:=false]
      [] cigar: [¬ready ∧ ¬(served_cake ∨ served_cigar)
                 → served_cigar:=true || served_cake:=false]
      [] reset: [¬ready ∧ (served_cake ∨ served_cigar)
                 → ready:=true]
```

Formally,

Definition 2.4: A *program signature* is a tuple (V,Γ) where

- V is an S-indexed family of sets where S is the set of sorts. In the sequel, V is also taken as the union of all these sets (assumed disjoint).
- 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., $D(a)=\{g\in \Gamma : a\in D(g)\}$. In the sequel, Γ is also taken as the union of all these sets (assumed disjoint).

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

Definition 2.5: Given a signature $\theta=(V,\Gamma)$, the language of terms is defined as follows, for every sort $s\in S$:

$$t_s ::= a \mid c \mid f(t_{s_1},\ldots,t_{s_n})$$

for $a\in V_s$, $c\in \Omega_{<>,s}$ and $f\in \Omega_{<s_1,\ldots,s_n>,s}$.
The language of propositions is defined as follows:

$$\phi ::= (t1_s =_s t2_s) \mid (\phi_1 \supset \phi_2) \mid (\phi_1 \vee \phi_2) \mid (\phi_1 \wedge \phi_2) \mid (\neg\phi)$$ ∎

For simplicity, whenever *t* is a Boolean term, we will abbreviate the proposition (t=true) to *t*. This simplification was used already in the example above.

Definition 2.6: A *program* is a pair (θ,Δ) where θ is a signature (V,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 map that, to every $a\in D(g)$, assigns a term of the same sort as a; if D(g) is empty, we denote F(g) by *skip*;
- B assigns to every action $g\in \Gamma$ a proposition (its guard). ∎

It is easy to recognise in this definition the basic features of parallel programs, namely guarded simultaneous assignments. Model-theoretic semantics for this language can be found in (Fiadeiro and Maibaum, 1995, 1996). A proof-theoretic semantics will be given in section 2.3 below.

A categorical formalisation of program interconnection and composition can also be given based on a notion of morphism that captures what in the literature is known as superposition or superimposition (e.g. Chandy and Misra, 1988, Francez and Forman, 1996).

Definition/Proposition 2.7: Given [redacted]ram signatures $\theta_1=(V_1,\Gamma_1)$ and $\theta_2=(V_2,\Gamma_2)$, [redacted]*gnature morphism* [redacted]om θ_1 to θ_2 consists of a pair $(\sigma_\alpha:V_1\to V_2,$[redacted]$\to\Gamma_2)$ of fun[redacted]s su[redacted]at (1) for every $s\in S$ and for every $a\in V_{1_s}$, $\sigma_\alpha(a)$[redacted]$_{2_s}$, and (2) f[redacted]ery [redacted]n $g\in\Gamma_1$, $\sigma_\alpha(D_1(g))\subseteq D_2(\sigma_\gamma(g))$. Program signa[redacted] and morphism[redacted]nstit[redacted] category *SIG*. ∎

Signature m[redacted]sms define tr[redacted]tions[redacted]ween the languages associated with each signature [redacted]e obvious wa[redacted]

Definition 2.[redacted] σ: $\theta_1\to\theta_2$,

$\sigma(t)$[redacted] $\sigma(a)$ | c | [redacted]t),..

$\sigma(\phi$[redacted]$)$ | $\sigma(\phi_1)\vee\sigma(\phi_2)$

[redacted]| $\sigma(\phi_1)\wedge\sigma(\phi$[redacted] ∎

Definition/Pr[redacted]**ition 2.9:** A[redacted]*erp*[redacted]$)\to(\theta_2,\Delta_2)$ is a signature [redacted]hism σ: $\theta_1\to$[redacted]ich t[redacted]

1. For all [redacted]Γ_1, $a_1\in D_1(g_1$[redacted]$_2$ B$_2$[redacted]$(\sigma(g_1),\sigma(a_1))$.
2. $_{\theta_2}$ $(I_2$[redacted]$(I_1))$.
3. For every $g_1\in\Gamma_1$, $_{\theta_2}$ $(B$[redacted]$_1))$ [redacted]$_1(g_1)))$.
4. For every $a_1\in V_1$, $D_2(\sigma(a_1))\subseteq\sigma(D_1$[redacted].

Programs and superposition morphisms co[redacted]te a category *PROG*. ∎

Given a signature θ, we denote by $_\theta$ the [redacted]sion of the consequence relation of the first-order logic over the language of the data types and attributes with the axioms DT of the data types.

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. Requirement 4 corresponds to a locality condition: new actions cannot be added to the domains of attributes of the source program. That is to say, no new actions can change the old attributes. 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 formalises a key ingredient of program structure and encapsulation. See (Fiadeiro and Maibaum, 1992, 1995, 1996).)

We shall now describe how complex programs can be put together from components in much the same way as we did for specifications:

Proposition 2.10: The category *PROG* is finitely cocomplete. ∎

The proof of this result can be found in (Fiadeiro and Maibaum, 1996). Basically, pushouts work as follows:

- 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, see (Fiadeiro and Maibaum, 1996);

- the initialisation condition of the resulting program is given by the conjunction of the initialisation conditions of the component programs.

As an example, consider the program:

```
program costumer is
var   bal:int
init  bal>0
do    pay: [bal>0 → bal:=bal–1]  []  eat: [true → skip]
```

This program models the behaviour of a customer. We are going to interconnect it with *vending machine* using the following channel:

```
program channel is
var
init  true
do    a: [true → skip]  []  b: [true → skip]
```

The configuration diagram for the required system is as follows:

$$\begin{tikzcd} & \text{channel} \arrow[dl, "{a \mapsto coin,\ b \mapsto cake}"'] \arrow[dr, "{a \mapsto pay,\ b \mapsto eat}"] & \\ \text{vending machine} & & \text{costumer} \end{tikzcd}$$

Its pushout is given, up to isomorphism, by

```
program [illegible]stomer is
[illegible]   ready, serve[illegible]cake, served_cigar:bool, bal:int
init  ready ∧ (ser[illegible]d_cake ∨ served_cigar) ∧ bal>0
do    pay_coin : [[illegible]ady ∧ ([illegible]rved_cake ∨ [illegible]rv[illegible]_cigar) ∧ b[illegible]>0
         → r[illegible]dy:=false || se[illegible]ed_cak[illegible]=fa[illegible] | [illegible]ved_cigar[illegible]fal[illegible] || [illegible]bal-1]
      []  ea[illegible]ake : [¬ready [illegible]
      [illegible]
      []  ci[illegible]
                    → serv[illegible]d_cigar:=true || s[illegible]ed_cake:=false]
      []  re[illegible]t: [¬ready ∧ (s[illegible]ved_cake ∨ se[illegible]ed_cigar)
                    → ready:=true]
```

together with (coin pay_coin, cake eat_cake, cigar cigar, reset reset, ready ready, served_cake served_cake, served_cigar served_cigar) and (pay pay_coin, eat eat_cake) as morphisms connecting *vending machine* and *costumer* to *served customer*, respectively.

This program is a simple superposition of *vending machine* – it strengthens the guard of the action *coin* with the guard of *pay* – (bal>0) and enriches its assignment with (bal:=bal–1).

2.3 Relating programs to specifications

In (Fiadeiro and Maibaum, 1995) we have further shown how the satisfaction relation between programs and specifications can be formalised in this setting.

Definition/Proposition 2.11: Every program signature (V,Γ) defines the temporal signature $V\cup\Gamma$. Because V and Γ are disjoint, this mapping extends to a functor, i.e. every morphism of program signatures extends to a morphism of temporal signatures. ∎

Consider now the first-order extension FLTX of the linear temporal logic defined in 2.1 which also includes the operator **X** (next). (See (Fiadeiro and Maibaum, 1992) for details.)

Definition/Proposition 2.12: Given a program (θ,Δ) where $\theta=(V,\Gamma)$ and $\Delta=(I,F,B)$, let $\mathcal{Int}((\theta,\Delta))$ be the following set of propositions in the language of $FLTX(V\cup\Gamma)$:

- the proposition $(\mathbf{beg} \supset I)$;
- for every action $g\in\Gamma$, the proposition $(g \supset B(g) \wedge (\bigwedge_{a\in D(g)} \mathbf{X}a=F(g,a)))$;
- for every $a\in V$, the proposition $(\bigvee_{g\in D(a)} g) \vee (\mathbf{X}a=a)$

1. We define $\mathcal{Spec}((\theta,\Delta))=(\Gamma,LTL(\Gamma)\cap(\mathcal{Int}((\theta,\Delta))\cup DT)^{\bullet})$ where DT is the collection of axioms defining the underlying data type.
2. The mapping $\mathcal{Spec}: \mathcal{PROG}\rightarrow\mathcal{SPEC}$ extends to a functor. ∎

The propositions in $\mathcal{Int}(P)$ capture the semantics of P: the first axiom establishes that I is an initialisation condition; the second set of axioms formalises the assignment – if *g* is about to occur, the next value of attribute *a* is the current value of F(g,a) – and establishes B(g) as a necessary condition for the occurrence of *g*; the last set of axioms (the locality axioms) capture locality of attributes: every attribute remains invariant if the next state transition is not performed by an action in its domain.

For instance, $\mathcal{Int}$(vending machine) consists of

$\mathbf{beg} \supset ready \wedge (served\_cake \vee served\_cigar)$
$coin \supset ready \wedge (served\_cake \vee served\_cigar) \wedge$
$\quad\quad \mathbf{X}\neg ready \wedge \mathbf{X}\neg served\_cake \wedge \mathbf{X}\neg served\_cigar$
$cake \supset \neg ready \wedge \neg(served\_cake \vee served\_cigar) \wedge$
$\quad\quad \mathbf{X}served\_cake \wedge \mathbf{X}\neg served\_cigar$
$cigar \supset \neg ready \wedge \neg(served\_cake \vee served\_cigar) \wedge$
$\quad\quad \mathbf{X}\neg served\_cake \wedge \mathbf{X}served\_cigar$
$reset \supset \neg ready \wedge (served\_cake \vee served\_cigar) \wedge \mathbf{X}ready$
$coin \vee reset \vee \mathbf{X}ready=ready$
$coin \vee cake \vee cigar \vee \mathbf{X}served\_cake=served\_cake$
$coin \vee cake \vee cigar \vee \mathbf{X}served\_cigar=served\_cigar$

and $\mathcal{Spec}$(vending machine) contains all the theorems in the language of {coin,cake,cigar,reset} that can be derived from this set, which can easily be shown to include the axioms of the specification of the vending machine.

Definition 2.13: A *realisation* of a specification S is a pair (σ,P) such that P:$\mathcal{PROG}$ and σ is a specification morphism S→Spec(P), i.e. Int(P) ⊨ σ(ϕ) for every ϕ∈S. ∎

Realisations generalise the notion of satisfaction by allowing the program and the specification to be over different signatures. More concretely, the program is allowed to have features that are not relevant to the specification (like the attributes). The morphism from S to Spec(P) corresponds to the way in which P realises S, i.e., intuitively, it records the *design decisions* (namely the choice of attributes) that lead from S to P (seen as a design exercise carried out in $\mathcal{SPEC}$). Notice that the morphism σ in a realisation (σ,P) of a specification S may be the result of a composition of other morphisms, i.e. of a stepwise refinement process that ends in program P.

The notion of realisation can be extended to (configuration) diagrams:

Definition 2.14: Let $\mathcal{S}$: I→$\mathcal{SPEC}$ be a specification diagram and $\mathcal{P}$:I→$\mathcal{PROG}$ 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→j in I, $\mathcal{S}_f;\eta_j=\eta_i;\mathit{Spec}(\mathcal{P}_f)$. ∎

The conditions $\mathcal{S}_f;\eta_j=\eta_i;\mathit{Spec}(\mathcal{P}_f)$ require that, in addition to the individual components, the interconnections be realised as well.

$$\begin{tikzcd}
 & & S_2 \arrow[rr, "\beta_2"] \arrow[dd, "\eta_2"] \arrow[dr, phantom, "\text{p.o.}"] & & S' \arrow[dd, dashed, "\eta'"] \\
 & S \arrow[ur, "\varphi_2"] \arrow[rr, "\varphi_1"] \arrow[dd, "\eta"'] & & S_1 \arrow[ur, "\beta_1"'] \arrow[dd, "\eta_1"] & \\
\mathcal{SPEC} & & \mathit{Spec}(P_2) \arrow[rr, "\mathit{Spec}(\sigma_2)"] & & \mathit{Spec}(P') \\
 & \mathit{Spec}(P) \arrow[ur, "\mathit{Spec}(\mu_2)"] \arrow[rr, "\mathit{Spec}(\mu_1)"'] & & \mathit{Spec}(P_1) \arrow[ur, "\mathit{Spec}(\sigma_1)"'] & \\
 & & P_2 \arrow[rr, "\sigma_2"] \arrow[dr, phantom, "\text{p.o.}"] & & P' \\
\mathcal{PROG} \arrow[uuu, "\mathit{Spec}"] & P \arrow[ur, "\mu_2"] \arrow[rr, "\mu_1"'] & & P_1 \arrow[ur, "\sigma_1"'] &
\end{tikzcd}$$

Theorem 2.15: Let $\mathcal{S}:I\rightarrow\mathcal{SPEC}$ be a specification diagram and $\mathcal{P}:I\rightarrow\mathcal{PROG}$ a program diagram such that $\mathcal{P}$ is a realisation of $\mathcal{S}$ through $(\eta_i)_{i:I}$. Then, the colimit of $\mathcal{P}$ is a realisation of the colimit of $\mathcal{S}$ in an essentially unique way. ∎

The picture in the previous page illustrates this theorem for a simple interconnection diagram. The existence and unicity of the morphism η' is guaranteed by the universal properties of colimits.

We should point out that this theorem holds for any functor $\mathcal{Spec}$ between categories of programs and of specifications. That is to say, it does not depend on the nature of the program design language (as long as it can be defined as a category) or of the specification logic (as long as it can be defined as an institution).

3 SYNTHESIS IN A CATEGORICAL FRAMEWORK

3.1 Motivation

Consider given a category $\mathcal{PROG}$ of programs and a category $\mathcal{SPEC}$ of specifications together with a functor $\mathcal{Spec}$: $\mathcal{PROG}\rightarrow\mathcal{SPEC}$ as above. The intended support for the synthesis of interconnections can be explained as follows.

Given realisations of component specifications (either obtained through traditional transformational methods, or synthesised directly from the specifications, or reused from previous developments), we would like to be able to synthesise the interconnections between the programs in such a way that the program diagram realises the specification diagram. That is to say, given specifications S_1 and S_2 (newly) interconnected via morphisms $\varphi_1:S\rightarrow S_1$ and $\varphi_2:S\rightarrow S_2$, and realisations (σ_1,P_1), (σ_2,P_2) of S_1 and S_2, respectively, one would like to be able to synthesise a realisation (σ,P) of S and morphisms $\mu_1:P\rightarrow P_1$ and $\mu_2:P\rightarrow P_2$ such that $\sigma;\mathcal{Spec}(\mu_i)=\varphi_i;\sigma_i$ (i=1,2).

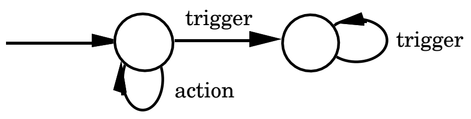
As mentioned in the introduction, synthesis of interconnections appears to be a straightforward way of extending existing methods for calculating programs from specifications, and synthesis methods in particular, to composite systems. The idea is that the individual components are calculated/synthesised using traditional methods, after which the interconnections between them are synthesised.

Another motivation is the ability to support incremental development in the sense that the addition of a component to a system should not require the recalculation (or the resynthesis) of the whole system but only of the new component and its interconnections to the previous system.

In order to illustrate these points, consider the system consisting of the vending machine and the regulator as specified in section 2.1. In section 2.2, we provided a realisation of the specification of the vending machine. We would now like to use the synthesis method of (Manna and Wolper, 1984) to realise the specification of the regulator and, then, synthesise the interconnections between the two programs in order to obtain a realisation of the specification diagram.

$$\begin{tikzcd}
 & & & S_2 \arrow[d, "\sigma_2"] & \\
 & & S \arrow[ur, "\varphi_2"] \arrow[rr, "\varphi_1"] \arrow[dd, dashed, "\sigma"'] & & S_1 \arrow[dd, "\sigma_1"] \\
\mathcal{SPEC} & & & Spec(P_2) & \\
 & & Spec(\mathbf{P}) \arrow[ur, dashed, "Spec(\mu_2)"] \arrow[rr, dashed, "Spec(\mu_1)"'] & & Spec(P_1) \\
 & & & P_2 & \\
\mathcal{PROG} \arrow[uuu, line width=2pt, "Spec"] & & \mathbf{P} \arrow[ur, dashed, "\mu_2"] \arrow[rr, dashed, "\mu_1"'] & & P_1
\end{tikzcd}$$

The application of the synthesis method to the formula **G**(trigger ⊃ **G**¬action)[*] returns the following automaton where the state on the left corresponds to the satisfiability of {**G**(trigger ⊃ **G**¬action)} and the state on the right to the satisfiability of {**G**¬action}:



This automaton is easily formalised as a COMMUNITY program: the two states are modelled through the values of an attribute *state*; the guards and assignments are translated directly from the automaton.

[*] The formula is prefixed with **G** because the procedure used in (Manna and Wolper, 1984) is based on an anchored version of temporal logic whereas the axioms of a specification express requirements that have to hold in every state of the component.

```
program regulator is
var   state:[1,2]
init  state=1
do    trigger: [true → state:=2] [] action: [state=1 → skip]
```

It remains to synthesise the interconnections between this program and the vending machine program. This is the problem that we are going to address in the rest of the paper.

3.2 Synthesising morphisms

The general statement of what it means to synthesise interconnections makes it clear that it is both necessary to synthesise the middle program (the "channel") and the morphisms[**] that are required to interconnect the given programs. Because, in the general case, any object can be used in an interconnection, this suggests, rather obviously, that a functor $\mathcal{Syn}: \mathcal{SPEC} \rightarrow \mathcal{PROG}$ is required. The properties that a functor is required to satisfy on identities and composition of morphisms make enough sense to be accepted without any discussion.

The fact that we now have a functor implies that we have a way of associating a program with every specification. That is to say, we now have an implicit way of synthesising programs from specifications. In particular, given a specification diagram $\mathcal{S}: \mathcal{I} \rightarrow \mathcal{SPEC}$, the functor $\mathcal{Syn}$ generates (synthesises), by composition, a program diagram $\mathcal{P}=\mathcal{S};\mathcal{Syn}:\mathcal{I} \rightarrow \mathcal{PROG}$. Because nothing can be assumed about the algorithmic nature of this association, $\mathcal{Syn}$ can also be interpreted as the result of some choices made among the possible transformations that a particular method provides for calculating programs from specifications.

More than programs, synthesis must return realisations. That is to say, $\mathcal{Syn}(S)$ must be provided together with a morphism $\eta_S:S \rightarrow \mathcal{Spec}(\mathcal{Syn}(S))$. The existence of η_S also expresses the correctness criterion for $\mathcal{Syn}$. Moreover, $\mathcal{Syn}$ must respect the interconnections in the following sense: given a specification diagram $\mathcal{S}: \mathcal{I} \rightarrow \mathcal{SPEC}$, it is necessary that the program diagram $\mathcal{P}=\mathcal{S};\mathcal{Syn}$ be a realisation of $\mathcal{S}$ through $(\eta_{\mathcal{S}_i})_{i:\mathcal{I}}$ as defined in 2.14, i.e. we must have, for every arrow $f:i \rightarrow j$ in $\mathcal{I}$, $\mathcal{S}_f;\eta_{\mathcal{S}_j}=\eta_{\mathcal{S}_i};\mathcal{Spec}(\mathcal{P}_f)$.

But these are the ingredients that define a natural transformation. Hence, $\mathcal{Syn}$ must be provided together with a natural transformation $\eta:1_{\mathcal{SPEC}} \rightarrow \mathcal{Syn};\mathcal{Spec}$.

As motivated in the case of the vending machine and the regulator, the ability to reuse an existing system when adding a component to it should also be supported, i.e. synthesis should be able to be integrated into an incremental development process. That is to say, given a configuration diagram

[**] Note that synthesising morphisms has already attracted some attention in the literature in the context of the KIDS system where the focus is on transforming (first-order) specifications into efficient algorithms (Smith, 1990, 1993). See also (Dimitrakos, 1996) for further development of these ideas.

$$\begin{tikzcd}
 & S & \\
C \arrow[ur] \arrow[rr, "\sigma"] & & \mathit{Spec}(P)
\end{tikzcd}$$

where P represents the existing system (e.g. the program *vending machine*) and S the specification of the component that is to be added (e.g. the regulator), we should be able to synthesise an interconnection between P and any program that realises S. This means that, given $\sigma{:}C\rightarrow\mathit{Spec}(P)$ we should be able to synthesise $\sigma'{:}\mathit{Syn}(C)\rightarrow P$ in such a way that the interconnection is respected, i.e. $\sigma=\eta_C;\mathit{Spec}(\sigma')$. This is equivalent to defining a (natural) bijection between the morphisms $C\rightarrow\mathit{Spec}(P)$ and the morphisms $\mathit{Syn}(C)\rightarrow P$. But this is, precisely, the property that characterises the existence of an adjunction from $\mathcal{SPEC}$ to $\mathcal{PROG}$ (MacLane, 1971). Hence:

Thesis 3.1: Given a category $\mathcal{PROG}$ of programs and a category $\mathcal{SPEC}$ of specifications together with a functor $\mathit{Spec}\colon \mathcal{PROG}\rightarrow\mathcal{SPEC}$, synthesis of interconnections is characterised by the existence of a left adjoint of *Spec* (which we call *Syn*). ▮

Notice that the counit of the adjunction $\varepsilon_P{:}\mathit{Syn}(\mathit{Spec}(P))\rightarrow P$ is not necessarily an isomorphism because *Spec*(P) may not be powerful enough to fully characterise P (we cannot guarantee that the specification domain is expressive enough to capture the semantics of P in full). The direction of the morphism reflects the fact that if we synthesise from the strongest specification of a program P, we obtain a program that cannot be stronger than P. Hence, the morphism ε_P provides a sort of "universal adaptor" between the program synthesised from *Spec*(P) and P itself.

The existence of a left adjoint to the functor $\mathit{Spec}\colon \mathcal{PROG}\rightarrow\mathcal{SPEC}$ is quite a strong property, which is not surprising because the ability to synthesise any specification is itself, in intuitive terms, a very strong property. In the literature, examples of synthesis from temporal logic specifications can be found, both from propositional linear temporal logic as above (Manna and Wolper, 1984) and from branching time logic (Emerson and Clarke, 1982). These approaches synthesise finite state automata, which are easily implemented as COMMUNITY programs as illustrated in section 3.1.

Difficulties are certain to arise in attempting to generalise these methods to the systems view. On the one hand, the functor *Spec* as defined in section 2.3 does not handle liveness properties. In order to extend this mapping to include liveness, we know that the functorial property is lost (more specifically, program morphisms do not necessarily map to specification morphisms). There are methods for dealing with this situation (Fiadeiro, 1996) but it is not yet clear how they can be combined with synthesis. On the other hand, it is not clear that the way synthesis methods generate states is compatible with the minimality requirements of adjunctions. This is an area of current research.

$$\begin{tikzcd}
 & & S \arrow[dd] & & \\
 & C \arrow[ur] \arrow[rrr, "\sigma"] \arrow[ddd, "\eta_C"'] & & & \mathit{Spec}(P) \arrow[ddd, "id_{\mathit{Spec}(P)}"] \arrow[ddl, dashed, "\eta_{\mathit{Spec}(P)}"'] \\
\mathcal{SPEC} & & \mathit{Spec}(P') & & \\
 & & & \mathit{Spec}(\mathit{Syn}(\mathit{Spec}(P))) \arrow[dr, dashed, "\mathit{Spec}(\varepsilon_P)"'] & \\
 & \mathit{Spec}(\mathit{Syn}(C)) \arrow[uur] \arrow[urr, dashed] \arrow[rrr, "\mathit{Spec}(\sigma')"'] & & & \mathit{Spec}(P) \\
 & & P' & & \\
\mathcal{PROG} \arrow[uuuu, "\mathit{Spec}"] & & & \mathit{Syn}(\mathit{Spec}(P)) \arrow[dr, dashed, "\varepsilon_P"] & \\
 & \mathit{Syn}(C) \arrow[uur] \arrow[urr, dashed, "\mathit{Syn}(\sigma)"] \arrow[rrr, "\sigma'"'] & & & P
\end{tikzcd}$$

In any case, the limited expressive power of the specification formalisms for which program synthesis has been shown to be possible (for instance, the results on synthesis mentioned above are only valid for propositional logics) means that synthesis of interconnections based on the existence of an adjunction will have little impact on program development. We shall see in the next section that a wide class of specification formalisms allow for a much weaker characterisation of synthesis of interconnections.

3.3 Synthesising channels

The conclusion, suggested by section 3.2, that the ability to synthesise interconnections is difficult to achieve, results from the fact that interconnections were taken to be expressed through arbitrary diagrams. In such cases, because any object can be used as a "channel" in an interconnection, synthesis of arbitrary interconnections requires the ability to synthesise programs from arbitrary specifications and, thus, arbitrary morphisms, which necessitates an adjunction.

However, the very examples used in section 2 suggest that we may explore the notion of interconnection a little further.

We start by recalling that finite cocompleteness – the ability to compute the colimit of every finite diagram – derives from the existence of initial objects and pushouts. We can, therefore, concentrate on pushouts as elementary interconnections.

On the other hand, the example that we used in section 2.1 for illustrating interconnection at the specification level used for channel an empty theory presentation, i.e. just a signature. This makes sense because the axioms of the middle specification (channel) play no role in the computation of the pushout. Indeed, recalling proposition 2.3, the theorems of the resulting specification are computed directly from the axioms of the components. Only the signature of the middle specification has a non redundant role in computing the pushout, namely in establishing the required interconnections.

Hence, it seems that, in circumstances that we have to determine, "channels" constitute just a subclass of the objects, which implies that the synthesis of interconnections only requires that a subclass of the specifications be synthesisable, as well as only the morphisms that have their origin in this subclass, i.e. channels. This is the direction that we are going to explore in the rest of the paper.

The relationship between theories and channels as empty theory presentations is also characterised by the existence of a faithful functor that "forgets" the theorems. In fact, this functor is a right adjoint of the (full) inclusion functor, i.e. channels define a coreflective subcategory. This property allows us to prove that the pushout of any two morphisms σ_1:C$\rightarrow$C$_1$ and σ_2:C$\rightarrow$C$_2$ can be made through a diagram in which the middle object is a channel, i.e. results from an empty theory presentation. Indeed, consider the morphisms $\sigma'_i=\varepsilon_C;\sigma_i$ where ε_C is the counit of the coreflection. Let μ_i:C$_i\rightarrow$C' be a pushout of the σ_i. We prove that μ_i:C$_i\rightarrow$C' are also a pushout of the σ'_i.

$$\begin{tikzcd}
 & & C_1 \arrow[dr, "\mu_1"] \arrow[drr, "\varphi_1"] & & \\
\mathcal{U}(C) \arrow[r, "\varepsilon_C"] \arrow[urr, dashed, "\sigma'_1"] \arrow[drr, dashed, "\sigma'_2"'] & C \arrow[ur, "\sigma_1"] \arrow[dr, "\sigma_2"'] & & C' \arrow[r, dashed] & C'' \\
 & & C_2 \arrow[ur, "\mu_2"'] \arrow[urr, "\varphi_2"'] & &
\end{tikzcd}$$

For this purpose, let φ_i:C$_i\rightarrow$C" be s.t. $\sigma'_1;\varphi_1=\sigma'_2;\varphi_2$, i.e. $\varepsilon_C;\sigma_1;\varphi_1=\varepsilon_C;\sigma_2;\varphi_2$. Because, when the right adjoint is faithful, each counit is an epi (MacLane, 1971), we obtain $\sigma_1;\varphi_1=\sigma_2;\varphi_2$. The result then follows from the fact that the μ_i:C$_i\rightarrow$C' are a pushout of the σ_i.

The straightforward generalisation of the case in which channels are a coreflective subcategory of the category of components is the existence of a full embedding $\mathcal{F}$: $\mathcal{X}\rightarrow\mathcal{C}$ from a category $\mathcal{X}$ of channels to a category $\mathcal{C}$ of components that admits a right adjoint $\mathcal{U}$. This is what happens if we take channels to be signatures instead of empty theory presentations. This also corresponds to the notion of coreflection used in (Sassone et al, 1993) for models of concurrency.

Definition 3.2: A category $\mathcal{C}$ is said to be *coordinated over* a category $\mathcal{X}$ iff it admits a faithful coreflector $\mathcal{U}:\mathcal{C}\rightarrow\mathcal{X}$, i.e. a faithful, right adjoint functor $\mathcal{U}$ for which the units are identities. ∎

Notice that, in those conditions, the left adjoint is a full embedding (MacLane, 1971).

Theorem 3.3: Let $\mathcal{C}$ be a category coordinated over $\mathcal{X}$ and $\mathcal{F}$ be the left adjoint. Then, any interconnection between objects of $\mathcal{C}$ can be made through a diagram of the form

$$\begin{tikzcd}
& \mathcal{F}(X) \arrow[dl] \arrow[dr] & \\
C_1 & & C_2
\end{tikzcd}$$

where X is an object of $\mathcal{X}$. ∎

This theorem means that the category $\mathcal{X}$ provides the *channels* through which any interconnection between objects of $\mathcal{C}$ can be established.

The ability to provide such a clean separation between individual components and their interaction is typical of coordination models and languages (Ciancarini and Hankin, 1996). Hence, the use of the adjective *coordinated* to characterise such situations.

Hence, it seems that, for the purpose of synthesising interconnections in coordinated specification formalisms, all that we are required is to synthesise from objects of the form $\mathcal{F}(X)$ and morphisms of the form $\mathcal{F}(X)\rightarrow S$. The following theorem proves it:

Theorem 3.4: Let $\mathcal{PROG}$ and $\mathcal{SPEC}$ be two categories together with a functor $\mathcal{S}pec$: $\mathcal{PROG}\rightarrow\mathcal{SPEC}$ such that

1. $\mathcal{SPEC}$ is coordinated over a category $\mathcal{X}$ with coreflector $\mathcal{U}$ and embedding $\mathcal{F}$;
2. $\mathcal{PROG}$ admits a right adjoint $\mathcal{V}:\mathcal{PROG}\rightarrow\mathcal{X}$ with left adjoint $\mathcal{G}$;
3. $\mathcal{S}pec;\mathcal{U}=\mathcal{V}$ and $\mathcal{G};\mathcal{S}pec=\mathcal{F}$.

$$\begin{tikzcd}
\mathcal{SPEC} \arrow[dr, shift right, "\mathcal{U}"'] & \\
& \mathcal{X} \arrow[ul, shift right, "\mathcal{F}"'] \arrow[dl, shift left, "\mathcal{G}"] \\
\mathcal{PROG} \arrow[uu, "\mathcal{S}pec"] \arrow[ur, shift left, "\mathcal{V}"] &
\end{tikzcd}$$

Then, given objects S_1 and S_2 of $\mathcal{SPEC}$ interconnected via morphisms $\varphi_1:\mathcal{F}(X)\rightarrow S_1$ and $\varphi_2:\mathcal{F}(X)\rightarrow S_2$, and realisations (σ_1,P_1), (σ_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 (φ_1,φ_2) as follows:

– Y is $\mathcal{G}(X)$, which realises $\mathcal{F}(X)$ through the identity morphism;

– $\mu_i=\mathcal{G}(\mathcal{U}(\varphi_i;\sigma_i));\varepsilon_{P_i}$ where ε is the counit of the adjunction between $\mathcal{G}$ and $\mathcal{V}$.

proof:

– from 3, $\mathcal{Spec}(\mathcal{G}(X))=\mathcal{F}(X)$, hence $\mathcal{G}(X)$ does realise $\mathcal{F}(X)$ through the identity;

– $\varphi_i;\sigma_i:\mathcal{F}(X)\rightarrow\mathcal{Spec}(P_i)$, so $\mathcal{U}(\varphi_i;\sigma_i)$: $\mathcal{U}(\mathcal{F}(X))\rightarrow\mathcal{U}(\mathcal{Spec}(P_i))$;
but $\mathcal{U}(\mathcal{F}(X))=X$ and $\mathcal{U}(\mathcal{Spec}(P_i))=\mathcal{V}(P_i)$, so $\mathcal{U}(\varphi_i;\sigma_i))$: $X\rightarrow\mathcal{V}(P_i)$ which implies $\mathcal{G}(\mathcal{U}(\varphi_i;\sigma_i));\varepsilon_{P_i}$: $\mathcal{G}(X)\rightarrow P_i$

– it remains to prove that the interconnections are respected, i.e. $\mathcal{Spec}(\mu_i)=\varphi_i;\sigma_i$;
but $\mathcal{Spec}(\mu_i) = \mathcal{Spec}(\mathcal{G}(\mathcal{U}(\varphi_i;\sigma_i)));\mathcal{Spec}(\varepsilon_{P_i}) = \mathcal{F}(\mathcal{U}(\varphi_i;\sigma_i));\mathcal{Spec}(\varepsilon_{P_i})$;
this implies $\mathcal{U}(\mathcal{Spec}(\mu_i)) = \mathcal{U}(\mathcal{F}(\mathcal{U}(\varphi_i;\sigma_i)));\mathcal{U}(\mathcal{Spec}(\varepsilon_{P_i})) = \mathcal{U}(\varphi_i;\sigma_i);\mathcal{V}(\varepsilon_{P_i})$;
the equations in 3 imply that the units of the adjunction between $\mathcal{G}$ and $\mathcal{V}$ are also identities, which in turn implies that $\mathcal{V}(\varepsilon_{P_i})$=id;
hence, $\mathcal{U}(\mathcal{Spec}(\mu_i))=\mathcal{U}(\varphi_i;\sigma_i)$;

because $\mathcal{U}$ is faithful, we obtain $\mathcal{Spec}(\mu_i)=\varphi_i;\sigma_i$. ▮

Notice that, in the conditions of this theorem, $\mathcal{Spec}$ does not need to be right adjoint and, hence, this theorem covers more situations than thesis 3.1. In order to get a clearer view of the relationship between theorem 3.4 and the characterisation given in 3.1, consider the following corollary:

Corollary 3.5: Let $\mathcal{PROG}$ and $\mathcal{SPEC}$ be two coordinated categories with coreflectors $\mathcal{U}:\mathcal{SPEC}\rightarrow\mathcal{X}$ (with embedding $\mathcal{F}$) and $\mathcal{V}:\mathcal{PROG}\rightarrow\mathcal{Y}$ (with embedding $\mathcal{G}$). Let $\mathcal{Spec}$: $\mathcal{PROG}\rightarrow\mathcal{SPEC}$ and $\mathcal{Chan}$: $\mathcal{Y}\rightarrow\mathcal{X}$ be functors such that $\mathcal{Spec};\mathcal{U}=\mathcal{V};\mathcal{Chan}$. Then, if $\mathcal{Chan}$ admits a left adjoint $\mathcal{H}$ such that $\mathcal{H};\mathcal{G};\mathcal{Spec}=\mathcal{F}$, we can synthesise interconnections in the sense of theorem 3.4. ▮

This corollary simply means that, in order to be able to synthesise interconnections between coordinated formalisms, it is sufficient to have an adjunction between the two categories of channels involved. Naturally, this case covers thesis 3.1 in the trivial case when both $\mathcal{PROG}$ and $\mathcal{SPEC}$ are taken to be coordinated over themselves. Notice that theorem 3.4 is also recovered from 3.5 in the trivial case in which $\mathcal{PROG}$ is taken to be coordinated over itself. However, we should stress that, in theorem 3.4, $\mathcal{PROG}$ is not necessarily coordinated over $\mathcal{X}$: although the equations make $\mathcal{V}$ a coreflector, it does not have to be faithful, meaning that $\mathcal{X}$ does not have to cover all possible interconnections in $\mathcal{PROG}$.

$$\begin{tikzcd}
\mathcal{SPEC} \arrow[r, shift right, "\mathcal{U}"'] & \mathcal{X} \arrow[l, shift right, "\mathcal{F}"'] \arrow[d, dashed, shift left, "\mathcal{H}"] \\
\mathcal{PROG} \arrow[u, "\mathcal{Spec}"] \arrow[r, shift left, "\mathcal{V}"] & \mathcal{Y} \arrow[l, shift left, "\mathcal{G}"] \arrow[u, dashed, shift left, "\mathcal{Chan}"]
\end{tikzcd}$$

It remains to discuss the applicability of these results.

Proposition 3.6: Let $\mathcal{SPEC}$ be the category of theories of an institution. Then, $\mathcal{SPEC}$ is coordinated over the underlying category $\mathcal{SIGN}$ of signatures. ∎

That is to say, any specification formalism that results from an institution satisfies the requirements of theorem 3.4.

Concerning the conditions on the "programming" category, general results cannot be given in the sense that there is not a notion which, like institutions, abstracts away from program design languages. Hence, we have to analyse each case individually, which we will illustrate using the category of COMMUNITY programs defined in section 2.2.

Proposition 3.7: The category $\mathcal{PROG}$ of COMMUNITY programs admits a right adjoint $\mathcal{V}:\mathcal{PROG}\rightarrow\mathcal{SET}$ that maps every program to the set of its actions.
<u>proof</u>: consider the mapping $\mathcal{G}$ that, to every set Γ, assigns the empty program over the signature $(\varnothing,\Gamma)$, i.e.

$$\begin{array}{lll} \mathcal{G}(\Gamma) \equiv & \textit{var} & \varnothing \\ & \textit{init} & \text{true} \\ & \textit{do} & \underset{g\in\Gamma}{[]}\ g\text{: [true} \rightarrow \textit{skip}] \end{array}$$

It is easy to see that $\mathcal{G}$ extends to a functor. We are going to prove that $\mathcal{G}$ is a left adjoint of $\mathcal{V}$. Let Γ be any set. Because $\mathcal{V}(\mathcal{G}(\Gamma))=\Gamma$, we can take identities for units. Hence, it remains to prove that, for any program P, if $\sigma:\Gamma\rightarrow\mathcal{V}(P)$ is a total function, then σ defines a morphism $\mathcal{G}(\Gamma)\rightarrow P$. It is clear that σ defines a signature morphism between the two programs because the signature of $\mathcal{G}(\Gamma)$ contains no attributes. On the other hand, all the conditions for a signature morphism to be a program morphism are met: the initialisation and the guards are universal, and the fact that there are no attributes implies that there is nothing else to prove. ∎

Proposition 3.8: Linear temporal logic and COMMUNITY support synthesis of interconnections.
<u>proof</u>: in order to apply theorem 3.4, and using 3.6 and 3.7, it remains to prove conditions 3. But these are easily checked once we notice that $\mathcal{S}pec$ projects every program signature to the corresponding set of actions. On the other hand, at the semantic level, it is easy to see that $\mathcal{G}(\Gamma)$ only generates tautologies. ∎

Notice that linear temporal logic and COMMUNITY do not illustrate corollary 3.5: the functor $\mathcal{V}$ defined in 3.7 is not faithful and, hence, $\mathcal{PROG}$ is not coordinated over temporal signatures. Indeed, it is simple to see that actions alone cannot be used as channels for programs – attributes are needed as well. We can extend COMMUNITY to a formalism coordinated over program signatures, for which it is trivial to obtain an adjunction wrt temporal signatures. This extension is, however, outside the scope of this paper.

We can now use theorem 3.4 to synthesise the interconnections between the program *vending machine* and the program *regulator* synthesised using Manna and Wolper's methods from the specification given in section 2.1.

According to 3.4, the program synthesised from the specification

```
**specification** channel **is**
    *sign*      a, b
    *axioms*
is
    **program** channel **is**
    *var*
    *init*   true
    *do*     a: [true → *skip*]  []  b: [true → *skip*]
```

and the synthesised morphisms are

$$\begin{tikzcd}
 & \text{channel} \arrow[dl, "\begin{array}{ll}\text{coin} & a\\ \text{cigar} & b\end{array}"'] \arrow[dr, "\begin{array}{ll}a & \text{trigger}\\ b & \text{action}\end{array}"] & \\
\text{vending machine} & & \text{regulator}
\end{tikzcd}$$

Notice that the pushout of this diagram returns the program:

```
    program regulated vending machine is
    var    ready, served_cake, served_cigar:bool; state:[1,2]
    init   ready [illegible] (served_cake ∨ served_cigar) ∧ state=1
    do     coin: [ready ∧ (served_cake ∨ served_cigar)
              → ready:=false || served_cake:=false || served_cigar:=false || state:=2]
           []  cake : [¬ready ∧ ¬(served_cake ∨ served_cigar)
                         → served_cake:=true || served_cigar:=false]
           []  cigar: [state=1 ∧ ¬ready ∧ ¬(served_cake ∨ served_cigar)
                         → served_cigar:=true || served_cake:=false]
           []  reset: [¬ready ∧ (served_cake ∨ served_cigar)
                         → ready:=true]
```

which strengthens the guard of the action *cigar* with the guard of *action* – (state=1). Hence, this action will never be undertaken after *coin* occurs.

4 CONCLUDING REMARKS

We addressed the problem of synthesising interconnections in the development of systems of interacting components. The main motivation for this study has been the integration of calculation methods in general, and synthesis in particular, within an incremental and reuse-based approach to system development. In the context of such development disciplines, it is essential to be able to synthesise the interconnections that are required for extending an existing system with new components from the interconnections provided at the specification level, so that the system does not need to be recalculated or resynthesised each time it needs to be extended.

In order to address the problem independently of specific specification and program design languages, we chose to work within a categorical framework in the spirit of Goguen's approach to General Systems Theory. In the adopted framework, components are modelled as objects, and complex systems are modelled as diagrams that depict the way in which the components of the system are interconnected. The morphisms of the different categories are used to express interconnections.

We showed that the ability to synthesise interconnections in such a general framework depends on the existence of a left adjoint to the functor that maps every program to its "strongest" specification. Because the existence of an adjunction is quite a restrictive condition, certainly for more expressive specification formalisms, we searched for weaker conditions based on stronger structural properties of the individual formalisms.

We showed that if the specification formalism is coordinated over a category of channels, a concept that we introduced for the first time in this paper and which reflects the properties of an emerging class of languages and models (Ciancarini and Hankin, 1996), then any interconnection can be synthesised provided that an adjunction can be found between the categories of programs and of channels. In the case where the two formalisms involved are coordinated over corresponding categories of channels, this amounts to the existence of an adjunction between the two categories of channels (specification and program channels), which is a much weaker requirement than the existence of an adjunction between the formalisms themselves.

Because every specification formalism that results from an institution is coordinated over the category of signatures, this result seems to have a wide application. On the program side, we showed that although the category of COMMUNITY programs is not coordinated, it admits an adjunction wrt the category of temporal signatures and, hence, supports the synthesis of interconnections. COMMUNITY is a language for parallel program design similar to UNITY (Chandy and Misra, 1988) and IP (Francez and Forman, 1996), which shows that our result is at least applicable to reactive system development in this class of languages. In this respect, we also discussed how existing methods for the calculation of programs from specifications, such as the synthesis methods proposed in (Emerson and Clarke, 1982, Manna and Wolper, 1984), can be combined with the proposed techniques for synthesising interconnections.

An obvious question to consider is the extent to which these ideas can be realised in practical software development environments. We are aware of several strands of work which are based on similar principles or which could be extended to realise these techniques. The work at Kestrel Institute on the Specware™ environment (Srinivas and Jüllig, 1995) (an extension of the earlier KIDS ideas (Smith, 1990)) is based on the use of category theory as a framework for representing and analysing specification and program structure. We are also aware of a project at the Laboratório de Métodos Formais at PUC in Rio de Janeiro, where Armando Haeberer is building a software development environment to support object-oriented design using ideas related to those outlined above as the semantic basis for formalising object-oriented methods.

Acknowledgements

We would like to thank the anonymous referees for many enlightening comments which, we are sure, have helped to clarify the messages in the paper.

This work was partially supported by the Esprit WG 8319 (MODELAGE), JNICT through contract PRAXIS XXI 2/2.1/MAT/46/94 (ESCOLA) and PCSH/OGE/1038/95 (MAGO), and the EPSRC through contracts GR/K67331 and GR/K68783.

5 REFERENCES

Abadi, M. and Lamport, L. (1993) Composing Specifications. *ACM Transactions on Programming Languages and Systems* **15**(1), 73-132.

Back, R. and Kurki-Suonio, R. (1988) Distributed Cooperation with Action Systems. *ACM Transactions on Programming Languages and Systems* **10**(4), 513-554.

Ciancarini, P. and Hankin, C. (1996), *Coordination Languages and Models*, in LNCS 1061, Springer-Verlag.

Chandy, K. and Misra, J. (1988), *Parallel Program Design - A Foundation*. Addison-Wesley.

Dimitrakos, T. (1996), *The Implementation of **PO**-specifications*, Research Report, Imperial College.

Ehrig, H. and Mahr, B. (1985), *Fundamentals of Algebraic Specification 1: Equations and Initial Semantics*. Springer-Verlag.

Emerson, E. and Clarke, E. (1982), Using Branching Time Logic to Synthesize Synchronisation Skeletons. *Science of Computer Programming* **2**, 241-266.

Feather, M. (1987), Language Support for the Specification and Development of Composite Systems. *ACM Transactions on Programming Languages and Systems* **9**(2), 198-234.

Feather, M. (1989) Constructing Specifications by Combining Parallel Elaborations. *IEEE Transactions on Software Engineering***15**(2), 198-208.

Fiadeiro, J. (1996) On the Emergence of Properties in Component-Based Systems, in M.Wirsing and M.Nivat (eds) *AMAST'96*, LNCS 1101, Springer-Verlag, 421-443.

Fiadeiro, J. and Maibaum, T. (1992) Temporal Theories as Modularisation Units for Concurrent System Specification. *Formal Aspects of Computing* **4**(3), 1992, 239-272.

Fiadeiro, J. and Maibaum, T. (1995) Interrconnecting Formalisms: supporting modularity, reuse and incrementality, in G.E.Kaiser (ed) *Proc. 3rd Symp. on Foundations of Software Engineering*, ACM Press, 72-80.

Fiadeiro, J. and Maibaum, T. (1996) Categorical Semantics of Parallel Program Design. *Science of Computer Programming*, in print.

Fickas, S. and Helm, B. (1992), Knowledge Representation and Reasoning in the Design of Composite Systems. *IEEE Transactions on Software Engineering* **18**(6), 470-482.

Francez, N. and Forman, I. (1996), *Interacting Processes*. Addison-Wesley.

Goguen, J. (1973), Categorical Foundations for General Systems Theory, in F.Pichler and R.Trappl (eds) *Advances in Cybernetics and Systems Research*, Transcripta Books, 121-130.

Goguen, J. and Burstall, R. (1992), Institutions: Abstract Model Theory for Specification and Programming. *Journal of the ACM* **39**(1), 95-146.

Goguen, J. and Ginali, S. (1978) A Categorical Approach to General Systems Theory, in G.Klir (ed) *Applied General Systems Research*, Plenum, 257-270.

Goldblatt, R. (1987) *Logics of Time and Computation*. CSLI.

MacLane, S. (1971) *Category Theory for the Working Mathematician*. Springer-Verlag.

Manna, Z. and Wolper, P. (1984) Synthesis of Communicating Processes from Temporal Logic Specifications. *ACM Transactions on Programming Languages and Systems* **6**(1), 68-93.

Sassone, V., Nielsen, M. and Winskel, G. (1993), A Classification of Models for Concurrency, in E.Best (ed) *CONCUR'93*, LNCS 715, Springer-Verlag, 82-96.

Shaw, M. and Garlan, D. (1995), Formulations and Formalisms in Software Architectures, in J.Leeuwen (ed) *Computer Science Today*, LNCS 1000, Springer-Verlag, 307-323.

Smith, D. (1990), KIDS – a Semi-Automatic Program Development System. *IEEE Transactions on Software Engineering* **16** (9), 1024-1043.

Smith, D. (1993) Constructing Specification Morphisms. *Journal of Symbolic Computation* **15** (5-6), 571-606.

Srinivas, Y. and Jüllig, R. (1995), Specware™: Formal Support for Composing Software, in B.Möller (ed) *Mathematics of Program Construction*, LNCS 947, Springer-Verlag.

BIOGRAPHY

José Luiz Fiadeiro is Associate Professor in Computing Science at the University of Lisbon. His research interests include software specification formalisms and methods, especially as applied to component-based, reactive systems, and their integration in the wider area of General Systems Theory.

Antónia Lopes is Assistant Lecturer at the University of Lisbon. She received an MSc degree in Applied Mathematics from the Technical University of Lisbon (1993) and she is now working towards a PhD on compositionality in the modular specification of reactive systems.

Tom Maibaum is Professor and Head of the Computing Department of Imperial College. His research interests are in formal methods for software engineering, including theories of specification and implementation, object-oriented systems, and non-classical logics in Computing.