Using Explicit State to Describe Architectures(*)

Antónia.Lopes and José.Luiz.Fiadeiro

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

Abstract. In order to achieve higher levels of abstraction in architectural design, we investigate extensions to parallel program design based on the use of explicit state variables to accommodate the action-based discipline of interaction that is typical of architecture description languages. Our study focus on primitives that support non-determinism, choice and fairness in guarded-command based languages, and on refinement principles that are compositional with respect to interconnection.

1 Introduction

Formal approaches for describing software architectures tend to use process-based languages. Typical examples are the architecture description languages WRIGHT [2], based on CSP, Darwin [16], based on the π-calculus, and Rapide [15], based on partially ordered sets of events. The fact that software architectures address the structure of systems in terms of components and interconnection protocols between them suggests the adoption of formalisms in which interaction is event-based. For instance, the ability to specify if a given action of a component is under the control of the component or the environment is paramount for the definition of connector roles – e.g. the fact that, in a client-server architecture, the choice of service is under the control of the client. This form of behaviour has been modelled using external and internal choice operators as, for instance, in WRIGHT [2].

However, such process description languages are notably lacking in abstraction mechanisms. Their primitives are directed to structuring the flow of control in an explicit way and, hence, there are crucial properties of protocols that cannot be captured at a higher level of abstraction. For instance, the description of a pipe-filter architecture with a particular discipline on the pipe (e.g. FIFO), or a client-server architecture with fairness requirements, typically requires a rather involved encoding [2].

In contrast, formalisms that use an explicit notion of state, such as Unity [3], TLA

(*) This work was partially supported through contracts PCSH/OGE/1038/95 (MAGO) and 2/2.1/TIT/1662/95 (SARA).

[12] or fair transition systems [17], provide a more flexible way of modelling the behaviour of components in the sense that execution sequences emerge from the interference of actions on the state rather than an explicit prescription of an ordering on the actions. The problem with such approaches is that interaction between components in a system is modelled through a shared state, exactly the dual of what is favoured by architecture description languages. As a result, these approaches do not support the separation between the description of the components of a system and their interaction – interaction has to be implemented in the programs that model the components.

Our purpose in this paper is to show how the two approaches can be reconciled in a parallel program design language that adopts an explicit state but an action-based discipline of interaction. The language that we discuss is an extension of CommUnity [5]. CommUnity was developed precisely as having the same computational model as Unity but a coordination model based on private state and shared actions. However, it lacked mechanisms for addressing some of the key issues that are required by architecture description languages such as the ability to handle non-determinism and choice, and refinement mechanisms that support role instantiation and are compositional with respect to component interconnection.

Section 2 refines this initial motivation by expanding on the issues that we have identified as fundamental for languages based on explicit state to accommodate architectural description. Section 3 presents the extended language and its model-theoretic semantics. This semantics is based on labelled transition systems that differ from what is traditional in reactive system modelling in the way the environment is taken into account. Section 4 is concerned with component interconnection, for which a categorical semantics is adapted from previous papers., e.g. [6]. Section 5 formalises component refinement in terms of morphisms that differ from the ones used to model interconnection. A notion of compositionality is formalised in this setting. These notions are applied in section 6 to architectural connectors.

2 Motivation

We start by illustrating the ideas briefly described in the introduction and motivating some of the proposed extensions to CommUnity.

Shared state vs shared actions. Consider that, in the development of a given system, the need for a component Merge has been identified that receives two streams of data, merges them, and makes the resulting stream available to the environment.

If the interaction between the component and its environment is uniquely based on shared state, then it is not possible to design Merge without assuming a particular interaction protocol. For instance, it is necessary to program the acknowledgment of the reception of each data item in each input channel and make assumptions on the way the environment acknowledges the reception of data items [1].

In contrast, an approach based on shared actions allows us to describe, at the level of each component, only what the component contributes to each action – its local view of the action – without assuming a fixed protocol. This is because the execution of shared actions is under the control both of the component and the environment. This mode of interaction – synchronous communication via shared actions – is very general in the sense that other modes of communication can be programmed over it [9], for instance as architectural connectors. Indeed, protocols themselves can be developed separately as components with which other components synchronise for communicating with one another, thus promoting the separation between the description of the behaviour of the individual components and their interaction.

Internal vs external choice. When designing a component, it is often important to specify, whenever there is more than one shared action that is enabled, whether the choice between them is determined by the component or by the environment. For instance, in the case of a server, the choice between services should not be decided by the server itself but left to its clients. Process description languages support this distinction through separate internal and external choice operators.

In an approach based on explicit state, conditions on the state of the component may be used to define when the component makes each action available. For instance, in object-based languages enforcing "design by contract" [18], pre-conditions define when methods are available for execution in the sense that, if called by its environment, the object ensures their execution with a certain post-condition. This is why refinement of methods does not allow pre-conditions to be strengthened: strengthening the pre-condition of a method would violate the contract established between the object and its environment according to which the object lets the environment choose the method whenever its pre-condition is true. This means that the programmer who is going to implement the server must make sure that the execution of the services will not be blocked when their pre-conditions hold.

However, in program design languages based on guarded-commands, such conditions cannot be captured through the guards of each action. This is because guards are typically used as mechanisms for ensuring safety by blocking the execution of actions. Hence, refinement does not allow them to be weakened. For instance, in the design of the control system of a boiler in a central heating system, the action that releases hot water into the system may be required to be blocked if the temperature of the water raises above a certain threshold determined by the specification of the piping system. A developer may decide to lower this threshold in an implementation because of physical restrictions imposed by internal components of the controller itself.

CommUnity reconciles these two design mechanisms by guarding actions with two conditions: the safety guard that must be true whenever the action is enabled, and the progress guard that, when true, implies that the action is enabled and, hence, available for the environment to choose it. The key issue here is the separation between guards and enabledeness of actions. Our proposal makes guards a specification mechanism for enabledeness in the sense that they do not fully determine the enabling condition of an action but impose constraints on it.

Underspecification vs Non-determinism. Safety and progress guards establish an interval in which the enabling condition of an action must lie: the safety guard is a lower bound for enabledeness, in the sense that it is implied by the enabling condition, and the progress guard is an upper bound in the sense that it implies the enabling condition. As such, the corresponding action may be underspecified in the sense that its enabling condition is not fully determined, and, hence, subject to refinement by reducing this interval, i.e. weakening the progress guard and strengthening the safety guard. Underspecification is an essential mechanism for ensuring that design can be conducted at the required level of abstraction, avoiding implementation decisions to be made in early stages. CommUnity supports other mechanisms for underspecification such as non-deterministic assignments.

Such forms of underspecification model what is sometimes called allowed non-determinism [11] in the sense that they determine a range of possible alternative implementations whose choice cannot be controlled by the environment of the component. In contrast, mechanisms such as progress guards model required non-determinism in the sense that they specify alternative forms of behaviour that must be exhibited by every single implementation, thus allowing the environment to control it. The distinction between these two forms of non-determinism is essential for supporting abstraction and choice in architectural design.

Fairness. The above mentioned extensions to CommUnity, motivated by the need to support architectural description, are concerned with the coordination aspects of architectures, i.e. with the mechanisms that are made available to coordinate the behaviour of the components within a system. In what concerns the computational side of components, i.e. the mechanisms that guarantee that each component behaves, individually, as intended, it is necessary to make available another notion of progress – one that ensures that certain states are eventually reached. Such progress properties – liveness – are typical of "classical" reactive system specifications based on shared state and private actions. Therefore, we extended CommUnity with locally-controlled actions for which the progress guard ensures liveness in all fair computations.

3 Programming the Individual Components

The syntax of CommUnity with the extensions motivated above is as follows:

program *P* **is**
var **output** *out(V)*
input *inp(V)*
internal *int(V)*
init *I*
do $\Box_{g \in sh(\Gamma)}$ $g : [B(g),U(g) \rightarrow \|_{v \in D(g)} v:\in F(g,v)]$
$\Box_{g \in prv(\Gamma)}$ **prv** $g : [B(g),U(g) \rightarrow \|_{v \in D(g)} v:\in F(g,v)]$

where

- *V* is the set of *variables*. Variables can be declared as *input*, *output* or *internal*. Input variables are read from the environment of the program but cannot be modi-

fied by the program. Output and internal variables are local to the program, i.e., they cannot be modified by the environment. We use *loc(V)* to denote the set of local variables. Output variables can be read by the environment but internal variables cannot. Each variable *v* is typed with a sort *Sort(v)*.

- Γ is the set of *action names*; each action name has an associated guarded command (see below). Actions can be declared either as *private* or *shared* (for simplicity, we only declare which actions are private). Private actions represent internal computations and their execution is uniquely under the control of the program. In contrast, shared actions represent interactions between the program and the environment. Hence, their execution is also under the control of the environment. Each action *g* is typed with a set *D(g)* consisting of the local variables that action *g* can change – its write frame. For every local variable *v*, we also denote by *D(v)* the set of actions that can change *v*. The pair $<V,\Gamma>$ is called the signature of the program.
- *I* is a proposition over the set of local variables – the initialisation condition.
- For every action $g \in \Gamma$, *B(g)* and *U(g)* are propositions over the set of variables – its *guards*. When the safety guard *B(g)* is false, *g* cannot be executed. When the progress guard *U(g)* holds, the program is ready to execute *g*. More precisely, (1) when *g* is a shared action, if *U(g)* holds, the program cannot refuse to execute *g* if the environment requests it, and (2) when *g* is a private action, if *U(g)* holds infinitely often then *g* is taken infinitely often. For simplicity, we write only one proposition when the two guards coincide.
- For every action $g \in \Gamma$ and local variable $v \in D(g)$, *F(g,v)* is a non-deterministic assignment: each time *g* is executed, *v* is assigned one of the values denoted by *F(g,v)*, chosen in a non-deterministic way (when $D(g)=\varnothing$, the only available command is the empty one which we denote by **skip**).

Example 3.1. Consider that, in the context of the development of a given system, the need for two components *Merge* and *Consumer* is identified. The component *Merge* merges two streams of natural numbers and transmits the resulting stream to the environment, guaranteeing that the output is ordered if the input streams are also ordered. The component *Consumer* receives a stream of natural numbers and stores it.

Consider first the component *Consumer*. In order to illustrate the ability of ComMUnity to support the high-level description of system behaviour, assume that, at a given level of abstraction, one is just concerned with coordinating the joint behaviour of the components in the system, ignoring for a moment the details of the internal computation performed by *Consumer* (the storing of the data),

```
program Consumer is
var output    cl: bool
    input     i: nat,  eof: bool
init          ¬cl
do            rec : [ ¬eof ∧ ¬cl, false → skip ]
      [] prv  close : [ true, eof∧¬cl → cl:=true ]
```

The fact that the internal computation related to the storing of data is being abstracted away is reflected in the fact that the body of action *rec* – modelling the reception of values from the environment – is empty, meaning that it does not change the

abstract state, and its progress guard is *false*, meaning that we cannot make precise how the reception is controlled. In section 5, we shall see how refinement mechanisms can be used to extend this program with details on the computational aspects.

The program above is, therefore, primarily concerned with the interface between *Consumer* and its environment. Part of this interface is established via the input channel *i* along which data are transmitted to *Consumer*. The (shared) action *rec* accounts for the reception of such transmissions. This action is shared between *Consumer* and its environment meaning that both *Consumer* and the environment have to give permission for the action to occur. The safety guard of *rec* requires *Consumer* to refuse a transmission when communication has been closed (see below). However, because the progress guard has been set to *false*, an implementation of *Consumer* can arbitrarily refuse to accept transmissions because there is no upper bound on the enabling condition of *rec*.

The other means of interaction with the environment is concerned with the closure of communication. The component can receive, through the input channel *eof*, a Boolean indicating that transmission along *i* has ceased. This should trigger a communication closure as indicated in the progress guard of *close*. Fairness then guarantees that the assignment is eventually performed. The assignment sets the output variable *cl* thus signaling to the environment that the component has stopped accepting transmissions. This is often important for coordination purposes, namely for managing the configuration of the system. Notice that *Consumer* can arbitrarily decide to execute *close*. This is because, being a private action, the environment cannot interfere with its execution. The only influence of the environment is to set its progress guard, i.e. to trigger the action. Hence, a specific choice of implementation may dictate conditions under which closure is required beyond the reception of a signal on the input variable *eof*, for instance related to the space made available for storing data. This will be illustrated in section 5.

Consider now the component *Merge*. It can be designed in CommUnity as follows.

program *Merge* **is**
var **output** o: *nat*, eof: *bool*
input i_1, i_2: *nat*, eof_1, eof_2: *bool*
internal rd: *bool*
init $\neg eof \wedge \neg rd$
do rec_1 : [$\neg(eof_1 \vee rd) \wedge (eof_2 \vee i_1 \leq i_2) \rightarrow o := i_1 \parallel rd := true$]
[] rec_2 : [$\neg(eof_2 \vee rd) \wedge (eof_1 \vee i_2 \leq i_1) \rightarrow o := i_2 \parallel rd := true$]
[] $send$: [$\neg eof \wedge rd \rightarrow rd := false$]
[] **prv** $close$: [$\neg eof \wedge \neg rd \wedge eof_1 \wedge eof_2 \rightarrow eof := true$]

The two input streams and the corresponding signal of end of data are received through the input variables i_1, i_2, eof_1 and eof_2. The resulting stream is transmitted through the variable o and the environment is informed of the end of data through the variable *eof*. The reception of a message in the input channel i_k is modelled by the shared action rec_k. The execution of this action is blocked when the end of data has already been signaled in this channel or the other channel presents a lower number to be read. Whenever rec_k is executed, the program copies the message presented in channel i_k to the output variable o and the action *send* becomes ready to be executed. The program waits for the reading of that message to become ready again to receive

one more message. Finally, after each channel has signalled that it has ended transmission, the program signals the end of data in the output channel *eof*. Because the closing of the communication does not involve the environment, this is modelled by a private action (the action *close*). Notice that, in this program, the safety and progress guard coincide for every action, meaning that these actions have a unique implementation. ∎

The mathematical model that we adopt for system behaviour is based on labelled transition systems. Because, as discussed in sections 1 and 2, we are interested in capturing the coordination aspects required for software architectures, namely the fact that a component can be guaranteed to make its services available for the environment to choose, the way the environment is taken into account in these models is different from what is traditionally found in the literature. The typical situation in reactive systems [17,12] is to take models that reflect the behaviour of a component in a given environment. That is, a model already exhibts a joint execution of the component and its environment. In our case, a model reflects the behaviour that the component *offers* to its environment. More concretely, the branching structure of transition systems is intended to reflect required non-determinism on the behaviour of the component individually. Internal non-determinism, which accounts for choice between different implementations, is essentially modelled by the existence of more than one such transition system as a model of a program. Openess to the environment is explicitly modelled through a special label $\perp$ that identifies steps performed by the environment.

We assume fixed an algebra $\mathcal{U}$ for the data types used in CommUnity. We denote by V_s, where $s \in S$, the set of variables of sort s. We denote by *Reach(T)* the set of reachable states of a transition system T.

Definition 3.2. A model for a program signature $<V,\Gamma>$ consists of a labelled transition system $T=<W,\rightarrow,W_0>$ over the set $\Gamma_\perp$ and an S-indexed family of mappings $\mathcal{V}_s:V_s \rightarrow [W \rightarrow \mathcal{U}_s]$ s.t.:

1. The set W_0 of initial states is non-empty;
2. For every $w \in W_0$ and S-indexed family of mappings $\mathcal{R}_s:inp(V_s) \rightarrow \mathcal{U}_s$, there exists $w' \in W_0$ s.t. $w \equiv_{loc(V)} w'$ and $\mathcal{V}_s(i)(w')=\mathcal{R}_s(i)$, for every $i \in inp(V)$ and $s \in S$;
3. For every $w,w' \in Reach(T)$ and $g \in \Gamma_\perp$ s.t. $w \xrightarrow{g} w'$ and for every S-indexed family of mappings $\mathcal{R}_s:inp(V_s) \rightarrow \mathcal{U}_s$, there exists $w'' \in W$ s.t. $w' \equiv_{loc(V)} w''$, $\mathcal{V}_s(i)(w'')=\mathcal{R}_s(i)$, for every $i \in inp(V)$ and $s \in S$, and $w \xrightarrow{g} w''$;
4. For every $w \in Reach(T)$ there exists $w' \in W$ s.t. $w \xrightarrow{\perp} w'$;
5. For every $v \in loc(V), w,w' \in Reach(T)$, and $g \notin D(v)$, if $w \xrightarrow{g} w'$ then $\mathcal{V}(v)(w)=\mathcal{V}(v)(w')$;

where $w \equiv_X w'$ abbreviates $\mathcal{V}(v)(w)=\mathcal{V}(v)(w')$ for every $v \in X$. ∎

A model for a program signature consists of a labelled transition system and a map that interprets the variables as functions that return the value that each variable takes in each state. Conditions 1 to 4 ensure that a model cannot constrain the behaviour of the environment. More concretely, a model is such that (1) its set of initial states is non-empty, (2) it does not constrain the initial values of the input variables nor (3) the values that input variables can have in the other reachable states, and (4) it does not prevent the execution of environment steps. Furthermore, condition 5 requires that the values of the local variables remain unchanged during the actions whose do-

main do not contain them. In particular, because $\perp \notin D(v)$, local variables are not subject to interference from the environment.

A model of a program is a model for its signature for which the initial states satisfy the initialisation constraint, the assignments are enforced, actions can only occur when their safety guards hold and are made available whenever their progress guards are true.

Definition 3.3. Given a program P and a model M for its signature, $M \models P$ iff:
1. For every $w_0 \in W_0$, $\mathcal{V}, w_0 \models I$;
2. For every $g \in \Gamma$, $v \in D(g)$, $w, w' \in Reach(M)$, if $w \xrightarrow{g} w'$ then $\mathcal{V}(w')(v) \in [\![F(g,v)]\!]^{\mathcal{V}}(w)$;
3. For every $g \in \Gamma$ and $w, w' \in Reach(M)$, if $w \xrightarrow{g} w'$ then $\mathcal{V}, w \models B(g)$;
4. For every $g \in \Gamma$ and $w \in Reach(M)$, if $\mathcal{V}, w \models U(g)$ then there exists $w' \in W$ s.t. $w \xrightarrow{g} w'$. ∎

As mentioned before, a model M such that $M \models P$ must be regarded as a model of the behaviour that the program offers to its environment: it defines a set of potential initial states and a set of potential transitions at each state — the choice between the alternatives that are not locally controlled, such as initialisation and modification of input variables and the execution of shared actions that are enabled, is left to the environment. In this way, S represents the degree of cooperation of the program P with respect to its environment.

The definition above makes clear that, for a program to admit models, it is necessary that, for every action g, the progress guard $U(g)$ implies the safety guard $B(g)$. Furthermore, because the emptiness of $F(g,v)$ for some $v \in D(g)$ also prevents action g to occur, it is also necessary that $U(g)$ implies the non emptiness of $F(g,v)$, for every $v \in D(g)$. Programs satisfying such conditions are called *realisable*.

For capturing the liveness properties of programs, we also have to model the way in which the locally-controlled actions that are infinitely often enabled are schedulled by the operating system. As usual, we shall consider infinite computations and we require strong fairness for private actions. That is, we consider paths in which any locally-controlled action which is infinitely often enabled cannot be neglected indefinitely. As usual, we say that an action is enabled at a state when there is a transition from that state that is labelled with the action.

Definition 3.4. A model of a program P consists of a model M for its signature s.t. $M \models P$, and a function $\Pi: W \to 2^{Path(M)}$ s.t., for every $w \in Reach(M)$ and $\pi \in \Pi(w)$, π starts at w and, for every $g \in prv(\Gamma)$, if g is enabled infinitely often in π then g is taken infinitely often in π. ∎

Consider again the program *Consumer*. After receiving the signal of end of data, and if the communication has not been closed already, the progress guard of action *close* becomes true. If this signal is stable, then the guard remains true while the action is not taken. Under strong fairness, *close* will eventually be executed.

4 Specifying How Components Interact

Software Architecture is about the modularisation of systems in terms of components and interconnections. Many of our previous papers [e.g., 5,6] have argued in favour

of the use of Category Theory as a mathematical framework for expressing such interconnections following Goguen's work on General Systems Theory [8]. The notion of morphism between programs captures what in the literature on parallel program design is known as superposition. Most of the conditions expressed in the definition below are standard when defining superposition and are more thoroughly discussed in our previous paper [5].

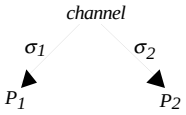
Definition/Proposition 4.1. A program morphism $\sigma: P_1 \to P_2$ consists of a (total) function $\sigma_{var}: V_1 \to V_2$ and a partial mapping $\sigma_{ac}: \Gamma_2 \to \Gamma_1$ s.t.:

1. For every $v \in V_1$, $o \in out(V_1)$, $i \in inp(V_1)$, $h \in int(V_1)$: $Sort_2(\sigma_{var}(v)) = Sort_1(v)$, $\sigma_{var}(o) \in out(V_2)$, $\sigma_{var}(i) \in out(V_2) \cup inp(V_2)$ and $\sigma_{var}(h) \in int(V_2)$;
2. For every $g \in sh(\Gamma_2)$ s.t. $\sigma_{ac}(g)$ is defined and $g' \in prv(\Gamma_2)$ s.t. $\sigma_{ac}(g')$ is defined: $\sigma_{ac}(g) \in sh(\Gamma_1)$ and $\sigma_{ac}(g') \in prv(\Gamma_1)$;
3. For every $g \in \Gamma_2$ s.t. $\sigma_{ac}(g)$ is defined and $v \in loc(V_1)$: $\sigma_{var}(D_1(\sigma_{ac}(g)) \subseteq D_2(g)$ and $\sigma_{ac}(D_2(\sigma_{var}(v)) \subseteq D_1(v)$;
4. For every $g \in \Gamma_2$ s.t. $\sigma_{ac}(g)$ is defined and $v \in D_1(\sigma_{ac}(g))$: $\models (F_2(g, \sigma_{var}(v)) \subseteq \underline{\sigma}(F_1(\sigma_{ac}(g), v)))$;
5. $\models (I_2 \supset \underline{\sigma}(I_1))$;
6. For every $g \in \Gamma_2$ s.t. $\sigma_{ac}(g)$ is defined, $\models (B_2(g) \supset \underline{\sigma}(B_1(\sigma_{ac}(g))))$;
7. For every $g \in \Gamma_2$ s.t. $\sigma_{ac}(g)$ is defined, $\models (U_2(g) \supset \underline{\sigma}(U_1(\sigma_{ac}(g))))$.

Programs and program morphisms constitute a category ***c-PROG***. ∎

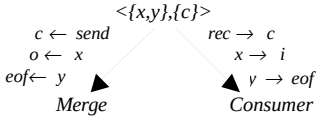
A morphism $\sigma: P_1 \to P_2$ identifies a way in which P_1 is a component of the system P_2. The map σ_{var} identifies, for every variable of the component, the corresponding variable in the system and σ_{ac} identifies the action of the component that is involved in each action of the system, if ever. Condition 1 states that sorts, visibility and locality of variables are preserved. Notice, however, that input variables of P_1 may become output variables of P_2. This is because the result of interconnecting an input variable of P_1 with an output variable of another component of P_2 results in an output variable of P_2. Condition 2 indicates that morphisms respect the type of actions (shared/private). Condition 3 means that the domains of variables are preserved and that an action of the system that does not involve an action of the component cannot change any variable of the component. Conditions 4 and 5 correspond to the preservation of the functionality of the component program: (4) the effects of the actions have to be preserved or made more deterministic and (5) initialisation conditions are preserved. Conditions 6 and 7 allow safety and progress guards to be strengthened but not weakened. Strengthening the safety guard is typical in superposition and reflects the fact that all the components that participate in the execution of a joint action have to give their permission. On the other hand, it is clear that progress for a joint action can only be guaranteed when all the components involved can locally guarantee so.

System configuration in the categorical framework is expressed via diagrams. Morphisms can be used to establish synchronisation between actions of programs P_1 and P_2 labelling diagram nodes as well as the interconnection of input variables of one component with output variables of the other component.



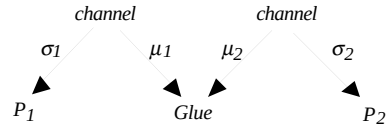
This kind of interaction can be established in a configuration diagram through interconnections of the form depicted above, where *channel* is essentially a set of input variables and a set of shared actions. Each action of *channel* acts as a *rendez-vous* point where actions from the components can meet (synchronise). Hence, action names act as interaction names as in IP [7]. Each variable of the channel provides for an input/output communication to be established between the components. See [4] for more details on the nature of channels.

Example 4.2. The diagram below defines a configuration in which *Merge* and *Consumer* synchronise on actions *send* and *rec*, and the input variables *i* and *eof* of *Consumer* are instantiated with the output variables *o* and *eof* of *Merge*, respectively.



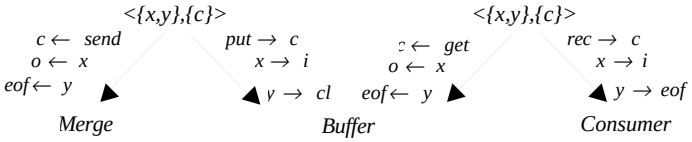
▮

More complex configurations can be described by using other interaction protocols that take the general form



where *Glue* is a program that describes how the activities of both components are coordinated in the intended architecture. Such configurations typically arise from the application of architectural connectors to given components of the system, as explained in section 6.

Example 4.3. Consider now the problem of specifying that *Merge* and *Consumer* communicate asynchronously. In this case, we have to interconnect the two programs through a third component modelling a buffer. The buffer coordinates the transmission of messages and informs the *Consumer* when there is no more data to transmit.



where

```
program Buffer is
var  output     o=head(b), eof: bool
     input      i: nat, cl: bool
     internal   b:List(nat), n:nat
init        b=<>∧n=0∧¬eof
do          put : [n<Limit → b:=b^<i> || n:=n+1 ]
     []     get : [¬(n=0) →b:=tail(b) || n:=n-1 ]
     []  prvsig : [ cl∧(n=0) → eof:=true ]
```

The program *Buffer* models a buffer with limited capacify and a FIFO discipline. The private action *sig* is responsible for signaling the end of data to the *Consumer* as soon as the buffer gets empty and *Merge* has already informed, through the channel *cl*, that it will not send anymore data. ∎

Not every diagram expresses a meaningful configuration in the sense that it describes a well configured system. For instance, diagrams in which output variables of different components are connected to one another do not make sense as configurations. Consider that $\boldsymbol{Var}$ denotes the forgetful functor from $\boldsymbol{c\text{-}PROG}$ to $\boldsymbol{SET}$ that maps programs to their underlying sets of variables. The class of diagrams that represent correct configurations of interconnected components can be formalised as follows.

Definition 4.4. Given a finite J-indexed multi-set of programs $\mathcal{P}$, a configuration diagram of a system with those components is a finite diagram $\iota:\boldsymbol{I}\rightarrow\boldsymbol{c\text{-}PROG}$ s.t.:

1. For every $j\in J$, $j\in|\boldsymbol{I}|$ and $\iota(j)=\mathcal{P}_j$;
2. For every $f:i\rightarrow j$ in $\boldsymbol{I}$, either ($i=j$ and $f=id_i$) or ($j\in J$ and $i\notin J$ and $\iota(i)$ is a channel);
3. For every $i\in|\boldsymbol{I}|\setminus J$ s.t. $\iota(i)$ is a channel, there exist distinct j,k and morphisms $f:i\rightarrow j$ and $g:i\rightarrow k$ in $\boldsymbol{I}$;
4. If $\{\mu_i:\boldsymbol{Var}(\iota(i))\rightarrow V:\ i\in|\boldsymbol{I}|\}$ is a colimit of $\iota;\boldsymbol{Var}$ then, for every $v\in V$, there exists at most one i in $\boldsymbol{I}$ s.t. $\mu_i^{-1}(v)\cap out(V_{\iota(i)})\neq\emptyset$ and, for such i, $\mu_i^{-1}(v)\cap out(V_{\iota(i)})$ is a singleton. ∎

Condition 1 means that every component of a system must be involved in its configuration diagram. Condition 2 states that the elementary interconnections are established through channels. Condition 3 ensures that a configuration diagram does not include channels that are not used. Finally, condition 4 prevents the identification of output variables. The fact that configurations represent systems is proved mathematically by the existence of a colimit for the diagram:

Definition/Proposition 4.5. Given a finite J-indexed multi-set of programs $\mathcal{P}$, modelling the components of a system *Sys*, and a configuration diagram ι, describing how these components interact, the program that models *Sys* is the program given by the colimit of ι, which always exists. We denote this program by $/\!/_\iota\mathcal{P}$. ∎

The colimit of a configuration diagram ι of a system corresponds to the parallel composition of the component programs (the programs that model the components of the system and the glues). Basically, $/\!/_\iota\mathcal{P}$ is defined as follows:

- the set of input variables of $/\!/_\iota\mathcal{P}$ consists of the input variables of each component program that are not identified with any output variable of any other component program and the output and internal variables of $/\!/_\iota\mathcal{P}$ are the (disjoint) union of, respectively, the output and the internal variables of the components;
- the shared actions of $/\!/_\iota\mathcal{P}$ are the joint actions defined by the synchronisation points and the set of private actions of $/\!/_\iota\mathcal{P}$ consists of the joint actions that represent the simultaneous execution of private actions of different components;
- the initialisation condition of $/\!/_\iota\mathcal{P}$ is given by the conjunction of the initialisation conditions of the component programs;
- the actions of $/\!/_\iota\mathcal{P}$ perform the parallel composition of the assignments of the joint actions and are guarded by the conjunction of the guards of the joint actions.

5 Refinement

A key factor for architectural description is a notion of refinement that can be used to support abstraction. In particular, refinement is necessary when, as illustrated in section 3, one wants to conduct the description of the architecture of a system at the level of the coordination that needs to be established between its components, leaving computational concerns for later stages of design. The notion of refinement can be captured by means of a different class of morphisms between programs.

Definition/Proposition 5.1. A program refinement morphism $\sigma: P_1 \rightarrow P_2$ consists of a (total) function $\sigma_{var}: V_1 \rightarrow V_2$ and a partial mapping $\sigma_{ac}: \Gamma_2 \rightarrow \Gamma_1$ s.t. conditions 1 to 6 of definition 4.1 hold and

7. For every $i \in inp(V_1)$, $\sigma_{var}(i) \in inp(V_2)$ and $\sigma_{var}{\downarrow}(out(V_1) \cup inp(V_1))$ is injective;
8. For every $g \in sh(\Gamma_1)$, $\sigma^{-1}(g) \neq \emptyset$;
9. For every $g_1 \in \Gamma_1$, $\models (\underline{\sigma}(U_1(g_1)) \supset \bigvee_{\sigma_{ac}(g_2)=g_1} U_2(g_2))$.

Programs and refinement morphisms constitute a category ***r-PROG***. ∎

A refinement morphism supports the identification of a way in which a program P_1 is refined by another program P_2. Each variable of P_1 has a corresponding variable in P_2 and each action g of P_1 is implemented by the set of actions $\sigma^{-1}(g)$ in the sense that $\sigma^{-1}(g)$ is a menu of refinements for action g. The actions for which σ_{ac} is left undefined (the new actions) and the variables which are not in $\sigma_{var}(V_1)$ (the new variables) introduce more detail in the description of the program.

Condition 7 ensures that an input variable cannot be made local by refinement (refinement does not alter the border between the system and its environment) and that different variables of the interface cannot be collapsed into a single one. Condition 8 ensures that actions that model interaction between the program and its environment have to be implemented. Condition 9 states that progress guards can be weakened but not strengthened. Indeed, because progress guards represent a requirement on the availability of an action for execution, refinement has to preserve that availability under the conditions established by the progress guard of the abstract program. Naturally, the circumstances under which this availability is guaranteed can be widened, which corresponds to the weakening of the progress guard.

Because condition 6 (see definition 4.1) allows safety guards to be strengthened (which corresponds to the preservation of the safety properties of the abstract program), the "interval" of (allowed) non-determinism defined by the two guards can be reduced by refinement. This is intuitive because refinement, pointing in the direction of implementations, should reduce allowed non-determinism. This is the reason why initialisation conditions can be strengthened and the non-determinism of assignments decreased. Notice that when the two guards coincide there cannot be any further refinement: the enabling condition for the action has been fully determined.

Preservation of required properties (including required non-determinism) and reduction of allowed non-determinism are intrinsic to any notion of refinement, and justify the conditions that we have imposed on morphisms. The morphisms that we used for modelling interconnections do not satisfy these properties. In particular, programs are not necessarily refined by the systems of which they are components,

which is consistent with other notions of refinement and parallel composition, e.g. CSP [9].

Example 5.2. In order to illustrate refinement consider the following program.

```
program Consumer2 is
var output    cl : bool
    input     i : nat,  eof : bool
    internal  x : nat,  rd : bool,  s : array(nat,N),  k : nat
init          ¬cl∧rd∧k=1
do            rec   : [ ¬eof∧¬cl∧rd∧k<N → x:=i  || rd:=false  ]
    [] prv    store : [ ¬rd  → rd:=true  || s[k]:=x  || k:=k+1  ]
    [] prv    close : [ ((eof∧¬cl∧rd)∨k≥N)  → cl:=true  ]
```

This program refines the program *Consumer* presented in 3.1. The new private action *store* models the storing of the received data in an array. The program is ready to receive a new number only if the previously received number has already been stored and the array is not full. The degree of internal nondeterminism over the action *close* present in the program *Consumer* given in section 3 was eliminated. The program eventually closes the communication if it receives the signal of end of data or reaches a state in which the array is full. ∎

Refinement and interconnection morphisms, though different as justified above, must be related by two important properties for architectural design to be supported. On the one hand, because composite programs are given by colimits and, hence, are defined up to isomorphism, refinement morphisms must be such that programs that are isomorphic with respect to interconnections, refine, and are refined exactly by, the same programs. The morphisms of ***r-PROG*** satisfy this requirement because isomorphisms in ***c-PROG*** are also isomorphisms in ***r-PROG***.

On the other hand, refinement must be compositional with respect to parallel composition, i.e. it is necessary that a refinement of a composite system can be obtained from arbitrary refinements of its components. This property also holds for the proposed notions of interconnection and refinement. More precisely, given a finite J-indexed multi-set of programs $\mathcal{P}$, a configuration diagram ι of a system with those components, and refinement morphisms $(\eta_j:\mathcal{P}_j\to\mathcal{P}_j')_{j\in J}$, there exists a refinement morphism $\eta:\|_\iota\mathcal{P}\to\|_{\iota'}\mathcal{P}'$, where ι' is the diagram obtained from ι by replacing the subdiagrams of the form

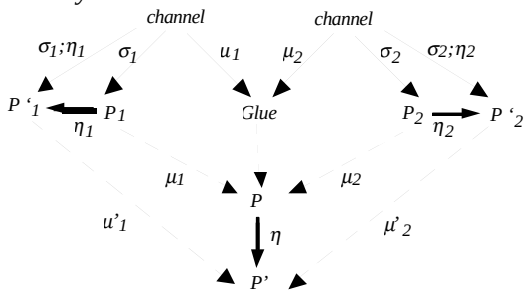
$$channel \xrightarrow{\sigma_i} P_i \qquad \text{by} \qquad channel \xrightarrow{\sigma_i;\eta_i} P'_i$$

where $(\sigma_i;\eta_i)$ denotes the program morphism defined by the composition of the underlying signature morphisms (it is not difficult to show that this construction gives rise to a configuration diagram for $\mathcal{P}'$).

The morphism η is unique if we require the preservation of the design decisions that lead from each $\mathcal{P}_i$ to $\mathcal{P}_i'$, i.e., $\mu_j;\eta=\eta_j;\mu'_j$, for any $j\in J$ where μ_j and μ'_j are the program morphisms which identify, $\mathcal{P}_j$ and $\mathcal{P}_j'$, as a component of, respectively, $\|_\iota\mathcal{P}$ and $\|_{\iota'}\mathcal{P}'$.

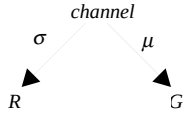
Let us consider, for instance, a system *Sys* with two components modelled by programs P_1 and P_2 interconnected through a third program as depicted above. The colimit of this diagram is a program P that models *Sys*. By picking up arbitrary re-

finements of programs P_1 and P_2, we obtain by composition a program P' that refines P and, hence, also models *Sys*.



6 Defining and Applying Architectural Connectors

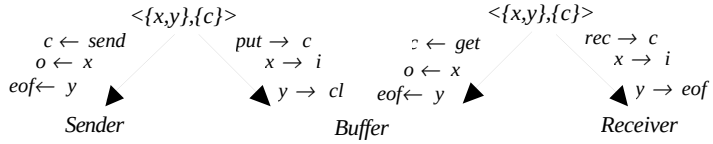
In a previous paper [6], we have shown how architectural connectors in the style of Allen and Garlan [2] can be given a categorical semantics in general, and in the original version of CommUnity in particular.



A connector (type) is defined by an object G – the glue – and a finite set of diagrams of the form depicted above, where each object R models a role of the connector. The roles describe the behaviour required of the components to which the connector can be applied. The glue describes how the activities of these components are coordinated once they instantiate the roles.

Role instantiation was modelled in [6] through the morphisms used for interconnection. However, the correct notion of morphism that models instantiation is the one that corresponds to refinement as argued in [2]. Because refinement is compositional with respect to parallel composition as explained in the previous section, the results developed in [6] are still valid when refinement is used instead of interconnection morphisms to model instantiation. In particular, the meaning of a connector is correctly given by the colimit of its configuration diagram in the sense that the properties inferred at the architectural level are guaranteed to hold in any system that results from the instantiation. Hence, by reasoning about the program resulting from the colimit, we may infer the properties of the underlying protocol.

Example 6.1. As an example, consider the following connector.



The glue of this connector is the program *Buffer* defined in example 4.4. The con-

nector has two roles – *Sender* and *Receiver* .

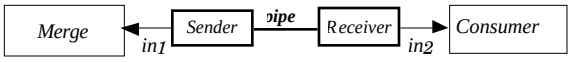
program *Sender* **is**
var output $o : nat,\ eof : bool$
input
internal $rd : bool$
init $\neg eof \wedge \neg rd$
do $prod : [\neg rd, false \rightarrow o :\in nat \,\|\, rd := true\ ]$
[] $send : [\neg eof \wedge rd, false \rightarrow rd := false]$
[] **prv** $close : [\neg eof, false \rightarrow eof := true\ ]$

program *Receiver* **is**
var output $cl : bool$
input $i: nat,\ eof: bool$
internal
init $\neg cl$
do $rec : [\ \neg eof \wedge \neg cl, false \rightarrow$ **skip** $\ ]$
[] **prv** $close : [\neg cl, \neg cl \wedge eof \rightarrow cl := true\ ]$

This connector is usually called a *pipe* [19] and describes the interaction protocol we used in example 4.4. More concretely, it defines asynchronous message passing through a channel with limited capacity that preserves the order of transmission. Furthermore, it defines that the sender has to signal the end of data (through the output variable *eof*) and the receiver is obliged to close the communication as soon as it is informed that there will be no more data. Notice that progress guards are essential to impose this restriction on the behaviour of the receiver and also to leave unspecified when and how many messages the sender (receiver) will send (receive). We chose the least deterministic assignment for the production of messages (denoted by the sort symbol *nat*) in order to avoid committing to a particular discipline of production.

The semantics of the connector is given by the parallel composition of *Sender*, *Receiver* and *Buffer* with the restrictions defined by the configuration diagram. By reasoning about the behaviour of this program it is possible to conclude, for instance, that the correctness of the transmission/reception of data does not depend on the order in which each role processes the data.

The programs *Merge* and *Consumer*, defined in 3.1, refine, respectively, the role *Sender* and *Receiver* (through inclusion morphisms in_1 and in_2) and, hence, the connector *pipe* can be used to interconnect these components.



7 Concluding Remarks

This paper was motivated by the need to promote higher levels of abstraction in architectural design while maintaining the separation between the description of component behaviour and the interaction protocols that coordinate their interaction. For this purpose, we investigated extensions to parallel program design based on the use of explicit state variables to accommodate the action-based discipline of interaction that is typical of process-based languages. The idea was to bring together the benefits of the two approaches: the explicit modelling of state facilitates the description of the components of a system in terms of some abstract state whereas the communication features of process languages are ideal for specifying the interaction of the components of a system in a given architecture.

In order to discuss specific proposals and illustrate them over typical examples, our

study focused on an extension of the language CommUnity proposed in [5] as an action-based version of Unity. The proposed extension integrates primitives that support non-determinism, choice and fairness, and refinement principles that are compositional with respect to interconnection. The distinction between allowed and required non-determinism motivated another fundamental change to CommUnity. Following Goguen's categorical approach to General Systems Theory [8], we had already proposed in previous papers (e.g. [6]) the adoption of Category Theory for formalising architectural principles and constructions. The work reported in this paper made it clear that separate notions of morphism are necessary for modelling component interconnection during system configuration, and refinement for moving between different levels of abstraction. Similar distinctions were already recognised in [14] for specifications of system behaviourin temporal logic. Work is underway towards the integration of specifications and CommUnity designs in the proposed categorical framework.

A different approach for combining process-based languages and state-based languages is the integration of existing languages, e.g., Z and Lotos [10] or Object-Z and CSP [20]. The advantages of the approach developed herein over such integrations are the following. On the one hand, it is not necessary to consider different specification languages for different aspects of the same system. Instead, CommUnity supports the description of the system at different levels of abstraction. Typically, the initial stages of design are concerned with the architecture of the system (at the coordination level), leaving the detailed description of the functionality of the components for later stages of design. On the other hand, our framework supports the description of connectors as well as their instantiation, at system configuration time, with concrete components for the incremental construction of structured systems, something that is out of the scope of the hybrids mentioned above.

On the contrary, an approach that seems more promising is the extension of existing languages and methods in ways similar to the proposed extension of Community. The relationship between program design based on guarded-commands and specification methods such as B, VDM or Z suggests that the use of private actions and progress guards could be integrated in those methods. The impact of adopting progress guards (for shared actions) on specification languages like B and VDM^{++} was already analysed in [13]. Further work is in progress that investigates the feasibility of these extensions.

References

1. M.Abadi and L.Lamport, "Composing Specifications", *ACM TOPLAS*, 15(1):73-132, 1993.
2. R.Allen and D.Garlan, "A Formal Basis for Architectural Connectors", *ACM TOSEM*, 6(3):213-249,1997.
3. K.Chandy and J.Misra, *Parallel Program Design - A Foundation*, Addison-Wesley 1988.
4. J.L.Fiadeiro and A.Lopes, "Algebraic Semantics of Coordination", in *AMAST'98*, Springer-Verlag, in print.

5. J.L.Fiadeiro and T.Maibaum, "Categorical Semantics of Parallel Program Design", *Science of Computer Programming*, 28:111-138,1997.
6. J.L.Fiadeiro and A.Lopes, "Semantics of Architectural Connectors", in *TAPSOFT'97*, LNCS 1214, Springer-Verlag 1997, 505-519.
7. N.Francez and I.Forman, *Interacting Processes*, Addison-Wesley 1996.
8. 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.
9. C.A.R Hoare, *Communicating Sequential Processes*, Prentice-Hall, 1985.
10. ITU Recommendation X.901-904, *Open Distributed Processing - Ref. Model*, July 1995.
11. R.Kuiper, "Enforcing Nondeterminism via Linear Temporal Logic Specifications using Hiding", in B.Banieqbal, H.Barringer and A.Pnueli (eds), *Temporal Logic in Specification*, LNCS 398, Springer-Verlag 1989, 295-303.
12. L.Lamport, "The Temporal Logic of Actions", *ACM TOPLAS*, 16(3):872-923, 1994.
13. K.Lano, J.Bicarregui, J.L.Fiadeiro and A.Lopes, "Specification of Required Non-determinism", in J.Fitzgerald, C.Jones and P.Lucas (eds), *Formal Methods Europe 1997*, LNCS 1313, 298-317, Springer-Verlag, 1997.
14. A.Lopes and J.L.Fiadeiro,"Preservation and Reflection in Specification", in *AMAST'97*, M.Johnson (ed), LNCS 1349, 380-394, Springer-Verlag, 1997.
15. D.C.Luckham and J.Vera, "An event-based architecture definition language", *IEEE TOSE*, 21(9):717-734,1995.
16. J.Magee and J.Kramer, "Dynamic Structure in Software Architecures", in *4th Symp. on Foundations of Software Engineering*, ACM Press 1996, 3-14.
17. Z.Manna and A.Pnueli, *The Temporal Logic of Reactive and Concurrent Systems*, Springer-Verlag, 1991.
18. B.Meyer, "Applying Design by Contract", *IEEE Computer*, Oct.1992, 40-51.
19. M.Shaw and D.Garlan, *Software Architecture: Perspectives on an Emerging Discipline*, Prentice Hall, 1996.
20. G.Smith, "A Semantic Integration of Object-Z and CSP for the Specification of Concurrent Systems", in J.Fitzgerald, C.Jones and P.Lucas (eds), *Formal Methods Europe 1997*, LNCS 1313, 62-81, Springer-Verlag, 1997.