Extension Morphisms for CommUnity

Nazareno Aguirre[1], Tom Maibaum[2], and Paulo Alencar[3]

[1] Departamento de Computación, FCEFQyN, Universidad Nacional de Río Cuarto,
Ruta 36 Km. 601, Río Cuarto (5800), Córdoba, Argentina,
naguirre@dc.exa.unrc.edu.ar
[2] Department of Computing & Software, McMaster University,
1280 Main St. West, Hamilton, Ontario, Canada L8S 4K1,
tom@maibaum.org
[3] School of Computer Science, University of Waterloo,
200 University Avenue West, Waterloo, Ontario, Canada N2L 3G1,
palencar@csg.uwaterloo.ca

Abstract. Superpositions are useful relationships between programs or components in component based approaches to software development. We study the application of invasive superposition morphisms between components in the architecture design language CommUnity. This kind of morphism allows us to characterise component extension relationships, and in particular, serves an important purpose for enhancing components to implement certain *aspects*, in the sense of aspect oriented software development. We show how this kind of morphism combines with regulative superposition and refinement morphisms, on which CommUnity relies, and illustrate the need and usefulness of extension morphisms for the implementation of aspects, in particular, certain fault tolerance related aspects, by means of a case study.

1 Introduction

The demand for adequate methodologies for modularising designs and development is increasing rapidly, due to the inherent complexities of modern software systems. Of course, these modularisation methodologies do not affect only the final implementation stages, but they also have an impact on earlier stages of software development processes. Thus, it is generally accepted that, for the modularisation to be effective (and persistent, and resistant to evolution), it needs to be applied from the start, at the level of specification or modelling of systems. Modularising, or structuring, specifications has important benefits. It allows one to divide the specifications into manageable parts, and to evaluate the consequences of our architectural design decisions prior to the implementation of the system. Moreover, it also favours the reuse of parts of the resulting implementations, and their adaptations and extensions for new application domains.

In the area of critical systems, specification languages are required to have a precise meaning (since formal semantics is crucial for eliminating ambiguities in specifications, and for developing tools for verification), and therefore specifications tend to be much longer than those of informal frameworks. Thus,

K. Futatsugi et al. (Eds.): Goguen Festschrift, LNCS 4060, pp. 173–193, 2006.

mechanisms for structuring or modularising specifications and designs are especially important for formal specification languages, as they help to make the specification and verification activities scalable. There exist many formal specification languages which put an emphasis on the way systems are built out of components (e.g., those reported in [19,9,20,18]). CommUnity is one of these languages; it is a formal program design language which puts special emphasis on ways of composing abstract designs of components to form designs of systems [6,5]. CommUnity is based on Unity [3] and IP [8], and its foundations lie in the categorical approach to systems design [10]. Its mechanisms for composing specifications have a formal interpretation in terms of category theoretic constructions [5,6]. Moreover, CommUnity's composition mechanisms combine nicely with a sophisticated notion of refinement, which involves separate concepts of action blocking and action progress.

We are particularly interested in CommUnity because, in our view, its design composition mechanisms make it suitable for the specification and combination (or "weaving") of *aspects*, in the aspect oriented software development sense [7]. Moreover, its rather abstract designs for components allow us to deal with aspects at a design level, in contrast to most of the work on aspects, which concerns implementation related stages (e.g., [14]). Some evidence of the adequacy of CommUnity as a design language for aspects relies on the possibility of defining *higher-order connectors* [16]. As shown in [16], a wide variety of aspects (e.g., fault tolerance, security, monitoring, compression, etc) can be superimposed on existing CommUnity architectures, by building "stacks" of more and more complex connectors between components.

Higher-order connectors provide a very convenient way of enhancing the behaviour of an architecture of component designs, by the superimposition of aspects. A crucial characteristic of CommUnity, which makes this possible, is the *complete externalisation of the definition of interaction* between components (a feature also exhibited by various other architecture description languages). The component coordination mechanism of CommUnity reduces the coupling between components to a minimum, and makes it feasible to superimpose behaviour (related to aspects) on existing systems via superposition and refinement of components. However, higher-order connectors are not powerful enough for defining various kinds of aspects, since some of these, as we will show, require extensions of the components as well as in the connectors. Thus, we are forced to consider another kind of superposition, known as *invasive superposition* [12], which allows us to define *extensions* of components. By combining extension with regulative superposition and refinement, we believe that we obtain a powerful framework in which we can define architectures, and enhance their behaviours by superimposing behaviour through aspects defined in terms of component extension and higher-order connectors. Having the possibility of extending components also provides us with a way of *balancing* the distribution of extended behaviour among connectors and components, which would otherwise be put exclusively on the connector side. This problem has also arisen in the context of object oriented design and programming, attempting to define various forms

of inheritance, resulting in the proposals attempting to characterise the concept of substitutability [15,21]. We believe that this proposal provides a more solid foundation for substitutivity, one that is better structured and more amenable to analysis. We propose a definition of extension in CommUnity, partly motivated by the definitions and proof obligations used to define the structuring mechanisms in B [1,4], that justifies the notion of substitutivity and provides a structuring principle for augmenting components by breaking encapsulation of the component. (Perhaps this should be considered a contradiction in terms!)

We show how extension morphisms combine with the superposition and refinement morphisms already present in CommUnity. We will also illustrate the need and usefulness of extension morphisms for the implementation of aspects, by means of a case study, based on a simple sender/receiver architecture communicating via an unreliable channel, which is then enhanced with some typical aspects, imposing a standard fault tolerance mechanism.

2 The Architecture Design Language CommUnity

In this section, we introduce the reader to the CommUnity design language and its main features, by means of an example. The computational units of a system are specified in CommUnity through *designs*. Designs are abstract programs, in the sense that they describe a *class* of programs (more precisely, the class of all the programs one might obtain from the design by refinement), rather than a single program [23,5].

Before describing in some detail the refinement and composition mechanisms of CommUnity, let us describe the main constituents of a CommUnity design. Let us first assume that we have a fixed set $\mathcal{ADT} = \langle \Sigma_{\mathcal{ADT}}, \Phi_{\mathcal{ADT}} \rangle$ of datatypes, specified as usual via a first-order specification. A CommUnity design is composed of:

- A set V of *channels*, typed with sorts in $\mathcal{ADT}$. V is partitioned into three subsets V_{in}, V_{prv} and V_{out}, corresponding to input, private and output channels, respectively. Input channels are the ones controlled, from the point of view of the component, by the environment. Private and output channels are the local channels of the component. The difference between these is that output channels can be read by the environment, whereas private channels cannot.
- A first-order sentence $Init(V)$, describing the initial states of the design[4].
- A set Γ of actions, partitioned into private actions Γ_{prv} and public actions Γ_{pub}. Each action $g \in \Gamma$ is of the form:

$$g[D(g)] : L(g), U(g) \rightarrow R(g)$$

  where $D(g) \subseteq V_{prv} \cup V_{out}$ is the (write) *frame* of g (the local channels that g modifies), $L(g)$ and $U(g)$ are two first-order sentences such that $\Phi_{\mathcal{ADT}} \vdash$

[4] Some versions of CommUnity, such as the one presented in [17], do not include an initialisation constraint.

$U(g) \Rightarrow L(g)$, called the lower and upper bound guards, respectively, and $R(g)$ is a first-order sentence $\alpha(V \cup D(g)')$, indicating how the action g modifies the values of the variables in its frame. ($D(g)$ is a set of channels and $D(g)'$ is the corresponding set of "primed" versions of the channels in $D(g)$, representing the new values of the channels after the execution of the action g.)

The two guards $L(g)$ and $U(g)$ associated with an action g are related to refinement, in the sense that the actual guard of an action g_r implementing the abstract action g, must lie between $L(g)$ and $U(g)$. As explained in [17], the negation of $L(g)$ establishes a blocking condition ($L(g)$ can be seen as a lower bound on the actual guard of an action implementing g), whereas $U(g)$ establishes a progress condition (i.e., an upper bound on the actual guard of an action implementing g, in the sense that it implies the enabling condition of an action implementing g).

Of course, $R(g)$ might not uniquely determine values for the variables $D(g)'$. As explained in [17], $R(g)$ is typically composed of a conjunction of implications $pre \Rightarrow post$, where *pre* is a precondition and *post* defines a multiple assignment.

To clarify the definition of CommUnity designs, let us suppose that we would like to model the unreliable communication between a sender and a receiver. We will abstract away from the actual contents of messages between these components, and represent them simply by an integer, identifying particular messages. Then, a sender is a simple CommUnity design composed of:

- An output channel `msg:int`, representing the current message of the sender.
- A private channel `rts: bool` ("ready to send"), indicating whether the sender is ready to send the current message or not (messages need to be produced before sending them).
- An action `send`, which, if the sender is ready to send (indicated by the boolean variable above), then goes back to a "ready to produce" state (characterised by the `rts` variable being false).
- An action `prod`, that, if the sender is in a "ready to produce" state, increments by one the `msg` variable (i.e., generates a new message to be sent) and moves to a "ready to send" state.

The CommUnity design corresponding to this component is shown in Figure 1.

In Fig. 1, the actions of the design have a single guard, meaning that their lower and upper bound guards coincide. We will illustrate refinement through more abstract designs below. An important point to notice in the sender design is the way it communicates with the environment through the `send` action. This action does not make a call to an external action, as one might expect; it will be the responsibility of other components to "extract" the value of the output variable `msg`, by synchronising other actions with the `send` action of the sender. This will become clearer later on, when we build architectures and describe in more detail the model of interaction between components in CommUnity.

To complete the picture, let us introduce some further designs. One is a simple component with a single integer typed output variable, used for communication

```
Design Sender
out
    msg: int
prv
    rts: bool
init
    msg=0 ∧ rts=false
do
    prod[msg,rts]: ¬rts → rts'=true ∧ msg'=msg+1
[]  send[rts]: rts → rts'=false
```

Fig. 1. A CommUnity design for a simple sender component.

and for modelling the loss of messages (Figure 2). The other one is a receiver component, somewhat similar in structure to the sender, but with an input variable instead of an output one, and a boolean channel `rtr` (ready to receive) instead of `rts` (Figure 3). To complete the picture, let us introduce some further designs. One is a simple component with a single integer typed output variable, used for communication and for modelling the loss of messages (Figure 2). The other one is a receiver component, somewhat similar in structure to the sender, but with an input variable instead of an output one, and a boolean channel `rtr` (ready to receive) instead of `rts`. To complete the picture, let us introduce some further designs. One is a simple component with a single integer typed output variable, used for communication and for modelling the loss of messages (Figure 2). The other one is a receiver component, somewhat similar in structure to the sender, but with an input variable instead of an output one, and a boolean channel `rtr` (ready to receive) instead of `rts`.

```
Design Communication_Medium
in
    in_msg: int
out
    out_msg: int
prv
    rts: bool
init
    out_msg=0 ∧ rts=false
do
   transmit[out_msg,rts]: ¬rts → out_msg'=in_msg ∧ rts'=true
[] lose[]: ¬rts → true
[] send[rts]: rts → rts'=false
```

Fig. 2. A CommUnity design for an unreliable communication medium.

```
Design Receiver
in
    msg: int
out
    curr_msg: int
local
    rtr: bool
init
    curr_msg=0 ∧ rtr=true
do
       rec[rtr,curr_msg]: rtr → rtr'=false ∧ curr_msg'=msg
[] prv cons[rtr]: ¬rtr → rtr'=true
```

Fig. 3. A CommUnity design for a receiver component.

2.1 Refinement Morphisms

Refinement morphisms constitute an important relationship between CommUnity designs. Not only do these morphisms allow one to establish "realisation" relationships, indicating that a component is a more refined or concrete version of another one, but they also serve an important purpose for parameter instantiation. In particular, refinement morphisms are essential for the implementation of *higher-order connectors* [16].

We will not give a fully detailed description of refinement morphisms here. We refer the interested reader to [6,23,16,17,5] for a detailed account of refinement in CommUnity.

A *refinement morphism* $\sigma : P_1 \to P_2$ between designs $P_1 = (V_1, \Gamma_1)$ and $P_2 = (V_2, \Gamma_2)$ consists of a total function $\sigma_{ch} : V_1 \to V_2$ and a partial function $\sigma_{ac} : \Gamma_2 \to \Gamma_1$ such that:

- σ_{ch} preserves the sorts and kinds (output, input or private) of channels; moreover, σ_{ch} is *injective* on input and output channels,
- σ_{ac} maps shared actions to shared actions and private actions to private actions; moreover, every shared action in Γ_1 has at least one corresponding action in Γ_2 (via σ_{ac}^{-1}),
- the initialisation condition is strengthened through the refinement, i.e., $\Phi_{\mathcal{ADT}} \vdash Init_{P_2} \Rightarrow \underline{\sigma}(Init_{P_1})$,
- every action $g \in \Gamma_2$ whose frame $D_2(g)$ includes a channel $\sigma_{ch}(v)$, with $v \in V_1$, is mapped to an action $\sigma_{ac}(g)$ whose frame $D_1(\sigma_{ac}(g))$ includes v,
- if an action $g \in \Gamma_2$ is mapped to an action $\sigma_{ac}(g)$, then $\Phi_{\mathcal{ADT}} \vdash L_2(g) \Rightarrow \underline{\sigma}(L_1(\sigma_{ac}(g)))$ and $\Phi_{\mathcal{ADT}} \vdash R_2(g) \Rightarrow \underline{\sigma}(R_1(\sigma_{ac}(g)))$,
- for every action $g \in \Gamma_1$, $\Phi_{\mathcal{ADT}} \vdash \underline{\sigma}(U_1(g)) \Rightarrow \bigvee_{h \in \sigma^{-1}(g)} U_2(h)$.

As specified by these conditions, the interval determined by the lower and upper bound guards can be reduced through refinement, and the assignments can be strengthened. The interface of a component design, determined by the output and input channels and shared actions, is *preserved* along refinement morphisms,

and the new actions that can be defined in a refinement are not allowed to modify the channels originating in the abstract component. Essentially, one can refine a component by making its actions more detailed and less underspecified (cleverly characterised by the reduction of the guard interval and the strenghthening of the assignments), and possibly adding more detail to the component, in the form of further channels or actions [17].

As an example of refinement, consider the more abstract version of the sender design shown in Figure 4. Notice that the assignment associated with `prod` is more abstract or liberal than the assignment of the same action in the `Sender` design. Also, the lower bound guards of both actions are equivalent to those of the corresponding actions in `Sender`, but the upper bound guards are *strengthened* to `false`. Clearly, `Abstract_Sender` is a more abstract version of the `Sender` (or, equivalently, `Sender` is a refinement of `Abstract_Sender`), and it is not difficult to prove that there exists a refinement morphism between these designs. In fact, `Abstract_Sender` is also a refinement of the `Communication_Medium` component (where the abstract `prod` operation corresponds to the operations `lose` and `transmit`), although it is less evident than in the first case.

```
Design Abstract_Sender
out
    msg: int
prv
    rts: bool
init
    msg=0 ∧ rts=false
do
    prod[msg,rts]: ¬rts, false → rts'=true ∧ msg'∈int
[]  send[rts]: rts, false → rts'=false
```

Fig. 4. A more abstract CommUnity design for a sender component.

2.2 Component Composition

In order to build a system out of the above components, we need a mechanism for composition. The mechanism for composing designs in Community is based on action synchronisation and the "connection" of output channels to input channels (shared memory). Basically, we need to connect the sender and receiver through the unreliable medium. This can be achieved by:

- identifying the output channel `msg` of the sender with the input channel `in_msg` of the medium,
- identifying the input channel `msg` of the receiver with the output channel `out_msg` of the medium,

– synchronising the action `send` of the sender with actions `transmit` and `lose` of the medium,
– synchronising the action `send` of the medium with action `rec` of the receiver.

The resulting architecture can be graphically depicted as shown in Figure 5. In this diagram, the architecture is shown using the CommUnity Workbench [24] graphical notation, where boxes represent designs, with its channels and actions[5], and lines represent the interactions ("cables" in the sense of [17]), indicating how input channels are connected to output channels, and which actions are synchronised. Notice that, in particular, action `send` of the sender is connected to two different actions of the medium; this requires that, in the resulting system, there will be two different actions corresponding to (or "invoking") the `send` action in the sender, one that is synchronised with `transmit` and another one that is synchronised with `lose`. This allows us to model very easily the fact that, sometimes, the sent message is lost (when the action `send-lose` is executed), without using further channels in the communication medium.

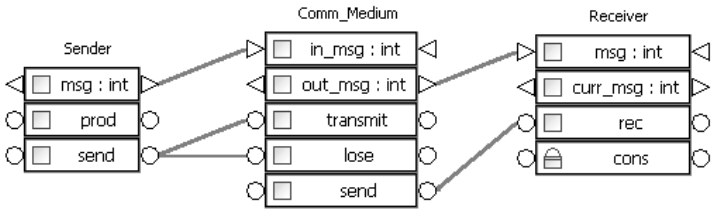

Fig. 5. A graphical view of the architecture of the system.

Semantics of Architectures. CommUnity designs have an operational semantics based on (labelled) transition systems. Architectural configurations, of the kind shown in Fig. 5, also have a precise semantics; they are interpreted as categorical diagrams, representing the architecture [17]. The category has designs as objects and the morphisms are *superposition relationships*. A superposition morphism between two designs A and B captures, in a formal way, the fact that B *contains* A, and uses it while respecting the encapsulation of A (regulative superposition). The interesting fact is that the joint behaviour of the system can be obtained by taking the *colimit* of the categorical diagram corresponding to the architecture [5,6]. Therefore, one can obtain a single design (the colimit object), capturing the behaviour of the whole system.

[5] Private actions are not displayed by the CommUnity Workbench, although we decided to show these actions, conveniently annotated, in the diagrams.

More formally, a *superposition morphism* $\sigma : P_1 \rightarrow P_2$ between designs $P_1 = (V_1, \Gamma_1)$ and $P_2 = (V_2, \Gamma_2)$ consists of a total function $\sigma_{ch} : V_1 \rightarrow V_2$ and a partial function $\sigma_{ac} : \Gamma_2 \rightarrow \Gamma_1$ such that:

- σ_{ch} preserves the sorts of channels; private and output channels must be mapped to channels of the same kind, but input channels can be mapped to output channels,
- σ_{ac} maps shared actions to shared actions and private actions to private actions,
- the initialisation condition is strengthened through the superposition, i.e., $\Phi_{\mathcal{ADT}} \vdash Init_{P_2} \Rightarrow \underline{\sigma}(Init_{P_1})$,
- every action $g \in \Gamma_2$ whose frame $D_2(g)$ includes a channel $\sigma_{ch}(v)$, with $v \in V_1$, is mapped to an action $\sigma_{ac}(g)$ whose frame $D_1(\sigma_{ac}(g))$ includes v,
- if an action $g \in \Gamma_2$ is mapped to an action $\sigma_{ac}(g)$, then $\Phi_{\mathcal{ADT}} \vdash L_2(g) \Rightarrow \underline{\sigma}(L_1(\sigma_{ac}(g)))$, $\Phi_{\mathcal{ADT}} \vdash R_2(g) \Rightarrow \underline{\sigma}(R_1(\sigma_{ac}(g)))$, and $\Phi_{\mathcal{ADT}} \vdash U_2(g) \Rightarrow \underline{\sigma}(U_1(\sigma_{ac}(g)))$.

As for refinement morphisms, superposition morphisms allow assignments to be strengthened, but not weakened. Intuitively, P_2 enhances the behaviour of P_1 via the superposition of additional behaviour, described in other components (and synchronised with P_1). So, the actions of the augmented component P_2 "using" corresponding actions in P_1 do at least what the actions of P_1 originally did. Since actions in P_2 should use the corresponding actions in P_1 within enabledness bounds, the lower bound guards of actions in P_1 must be strengthened when superposed in actions of P_2. Notice however that, as opposed to the case of refinement morphisms, upper bound guards can be strengthened, but not weakened; as explained in [17], this is a key difference between refinement and superposition, and reflects the fact that "all the components that participate in the execution of a joint action have to give their permission for the action to occur." (cf. p. 9 of [17]).

3 Component Extension in CommUnity

In this section we describe the main contribution of this paper, namely, a new kind of morphism between components for CommUnity. This kind of morphism, that we call *extension morphism*, enables us to establish extension relationships between components (of the kind defined by inheritance in object orientation), and is of a different nature, compared to the already existing refinement and superposition morphisms of CommUnity.

In order to illustrate the need for extension morphisms, let us consider the following case. Suppose that, for the existing system of communicating sender and receiver, we would like to superimpose behaviour related to the *monitoring* of the received messages. As explained in [16], this is possible to achieve, in an elegant and structured way, by using higher-order connectors. Essentially, an abstract monitoring structure is defined; this structure is composed of various abstract designs, used for characterising roles of the architecture, like sender, receiver and

monitor, and others necessary for the implementation of the "observed connector". These abstract designs are interconnected as shown in Figure 6. We will not describe these designs in detail, and refer the reader to [16], where a detailed description of this higher-order connector is given. However, it is important to mention that `Abstract_Sender` (which is given in Fig. 4) and `Abstract_Receiver` can be refined by, essentially, any pair of components providing the basic functionality for sending and receiving messages. Then, this higher-order connector is plugged into the existing architecture, through refinement, to obtain the resulting architecture of Figure 7. It is important to notice the difference between Figures 6 and 7; Fig. 6 describes the (abstract, non instantiated) higher-order connector for monitoring, whereas Fig. 7 described the *instantiation* of this higher-order connector (see how `Abstract_Sender`, `Abstract_Receiver` and `Abstract_Monitor` have been instantiated by `Communication_Medium`, `Receiver` and `Simple_Monitor`, respectively). The reader might observe that the actual monitor that we are using, described in Figure 8, simply counts the number of messages received by the receiver component. Notice that the guard of the monitor must be as weak as possible (i.e., `true`), to avoid interfering with the behaviour of the monitored operations.

In [16], several aspects are characterised and superimposed by using this same technique.

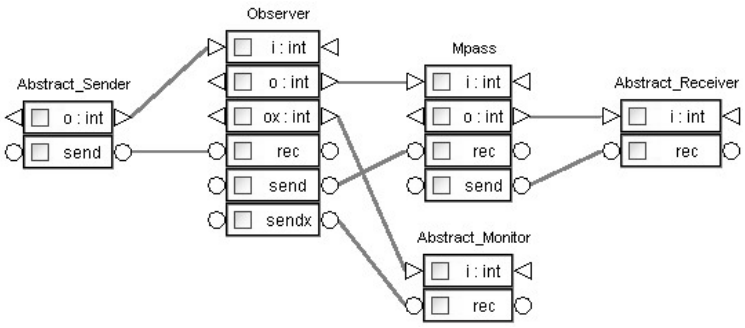

Fig. 6. A higher-order connector for monitoring.

Now suppose that we would like to superimpose a "resend message" mechanism on the architecture, in order to make the communication reliable. We can capture the loss of a "packet" through a monitor, instead of using it simply for counting the messages, as we did before. However, for the sender to reset and start sending the message again, we need to replace it with a slightly more sophisticated sender component, namely one with a reset operation, such as the one shown in Figure 9. Notice that `RES_Sender` cannot be obtained from `Sender` by superposition, since it is clear that the new `reset` operation modifies a channel originating in `Sender`. `RES_Sender` cannot be obtained through the refinement of

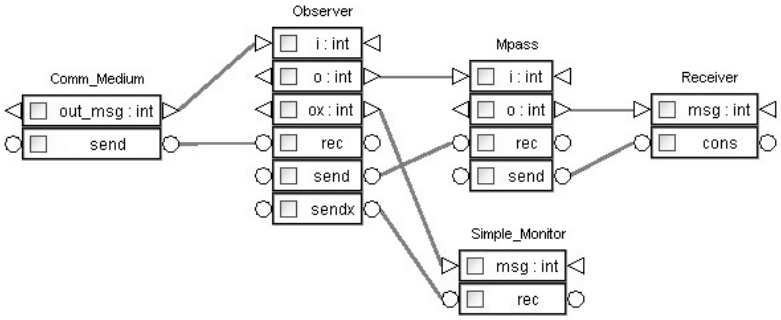

Fig. 7. Communication enhanced with a monitoring system.

```
Design Simple_Monitor
in
    msg: int
prv
    counter: int
init
    counter=0
do
    rec[counter]: true → counter'=counter+1
```

Fig. 8. A simple monitor to count received messages.

`Sender` either, since clearly its `reset` action, which modifies channels originating in `Sender`, should be mapped to a corresponding action in this design, but it does not respect any of the original assignments of actions `prod` and `send`, so it cannot be mapped to any of these.

However, there exists a clear relationship between the original `Sender` component and the new `RES_Sender`: the state of the original is extended, and more operations are provided (which might modify the channels of the original component), but the effect of the original actions is maintained. This relationship is a special case of what is called *invasive superposition* [12].

Invasive superposition has already been recognised as a possible relationship between CommUnity designs in [6]; moreover, therein it has been shown that CommUnity designs and invasive superpositions constitute a category. However, not much attention has been paid to invasive superposition for the architectural modelling of systems in CommUnity so far. Although not in the context of CommUnity, some researchers have employed various kinds of superpositions for defining architectures of components and augmenting their behaviours, particularly the work in [11]. Here, we propose the use of invasive superposition for characterising component extension in CommUnity.

```
Design RES_Sender
in
    lost-msg: int
out
    msg: int
prv
    rts: bool
init
    msg=0 ∧ rts=false
do
    prod[msg,rts]: ¬rts → rts'=true ∧ msg'=msg+1
[]  send[rts]: rts → rts'=false
[]  reset[msg,rts]: true → rts'=true ∧ msg'=lost-msg
```

Fig. 9. A CommUnity design for a sender component with a *reset* capability.

A distinguishing property typically associated with sound component extension is what is normally known as the *substitutability principle* [15]. This principle requires, in concordance with the now highly regarded "design by contract" approach [21], that if a component P_2 extends another component P_1, then one must be able to replace P_1 by P_2, and the "clients" of the original component must not perceive the difference. In other words, component P_2 should behave exactly as P_1, when put in a context where P_1 was expected. It is our aim to characterise such extensions through the definition of extension morphisms below.

Definition 1. *An extension morphism* $\sigma : P_1 \rightarrow P_2$ *between designs* $P_1 = (V_1, \Gamma_1)$ *and* $P_2 = (V_2, \Gamma_2)$ *consists of a total function* $\sigma_{ch} : V_1 \rightarrow V_2$ *and a partial mapping* $\sigma_{ac} : \Gamma_2 \rightarrow \Gamma_1$ *such that:*

- σ_{ch} *is injective and* σ_{ac} *is surjective,*
- σ_{ch} *preserves the sorts and kinds of channels,*
- σ_{ac} *maps shared actions to shared actions and private actions to private actions,*
- *there exists a formula* α*, using only variables that are contained in* $(V_2 - \sigma_{ch}(V_1))$*, and such that* $\Phi_{\mathcal{ADT}} \vdash \exists \overline{v} : \alpha(\overline{v})$ *and* $\Phi_{\mathcal{ADT}} \vdash Init_{P_2} \Leftrightarrow \underline{\sigma}(Init_{P_1}) \wedge \alpha$*,*
- *for every* $g \in \Gamma_2$ *such that* $\sigma_{ac}(g)$ *is defined, and for every* $v \in V_1$*, if* $\sigma_{ch}(v) \in D_2(g)$*, then* $v \in D_1(\sigma_{ac}(g))$*,*
- *if an action* $g \in \Gamma_2$ *is mapped to an action* $\sigma_{ac}(g)$*, then* $\Phi_{\mathcal{ADT}} \vdash \underline{\sigma}(L_1(\sigma_{ac}(g))) \Rightarrow L_2(g)$ *and* $\Phi_{\mathcal{ADT}} \vdash \underline{\sigma}(U_1(\sigma_{ac}(g))) \Rightarrow U_2(g)$*,*
- *for every* $g \in \Gamma_2$ *such that* $\sigma_{ac}(g)$ *is defined, there exists a formula* α*, using only primed variables that are contained in* $(V_2' - \sigma_{ch}(V_1)')$*, such that* $\Phi_{\mathcal{ADT}} \vdash \underline{\sigma}(L_1(\sigma_{ac}(g))) \Rightarrow (R_2(g) \Leftrightarrow \underline{\sigma}(R_1(\sigma_{ac}(g))) \wedge \alpha)$ *and* $\Phi_{\mathcal{ADT}} \vdash \exists \overline{v} : \alpha(\overline{v})$*, where* $\overline{v}$ *represents the primed variables of* α*.*

The first condition for extension morphisms requires all actions of the original component to be mapped to actions in the extended one, and the preservation

of all the channels of the original component. In particular, it is not allowed for several channels to be mapped to a single channel in the extended component. (Notice that if this was allowed, then the extended component might not be "plugged" into architectures where the original component could be "plugged", due to insufficient "ports" in the interface.) The second and third conditions above require the types and kinds of channels and actions to be preserved. The fourth condition allows the initialisation to be strengthened when a component is extended, but respecting the initialisation of the channels of the original component, and via realisable assignments for the new variables. The fifth condition indicates that "old actions" of the extended component can modify new variables, but the only old variables these can modify are the ones they already modified in the original component (in other words, frames can be expanded only with new channels). The sixth condition establishes that both the lower and upper bound guards can be weakened, but not strengthened. Finally, the last condition establishes that the actions corresponding to those of the original component must preserve the assignments to old variables, if the lower bound guard of the original component is satisfied; this provides the extension with some freedom, to decide how the action might modify old and new variables when executed under circumstances where the original action could not be executed. Again, it is required for the assignments for new variables to be "realisable".

Going back to our example, notice that `RES_Sender` is indeed an extension of `Sender`, where the associated extension morphism $\sigma = \langle \sigma_{ch}, \sigma_{ac} \rangle$ is composed of the identity mappings σ_{ch} and σ_{ac} on channels and actions, respectively. It is clear that these mappings are injective and surjective, respectively, and that sorts and kinds of channels are preserved by σ_{ch}, and the visibility constraints on actions are preserved by σ_{ac}. Moreover, since the initialisation and the write frames, guards and assignments of actions `send` and `prod` are not modified in the extension, the last four conditions in the definition of extension morphisms are trivially met.

Notice that extension morphisms are *invasive*, in the sense that new actions in the extended component are allowed to modify variables of the original component. However, extension morphisms differ from invasive superposition morphisms, as formalised in [6] in various ways. In particular, guards are weakened in extension morphisms, whereas these are strengthened in invasive superposition morphisms. Moreover, our allowed forms of assignment and initialisation strengthening are more restricted than those of invasive superposition morphisms.

It is not difficult to prove the following theorem, showing that, as for other morphisms in CommUnity, designs and extension morphisms constitute a category.

Theorem 1. *The structure composed of CommUnity designs and extension morphisms constitutes a* category*, where the composition of two morphisms σ_1 and σ_2 is defined in terms of the composition of the corresponding channel and action mappings of σ_1 and σ_2.*

Proof. The proof can be straightforwardly reduced to proving that the composition of extension morphisms is an extension morphism (the remaining points to prove that the proposed structure is a category are straightforward). So, let $\sigma_1 : P_1 \rightarrow P_2$ and $\sigma_2 : P_2 \rightarrow P_3$ be extension morphisms. The composition $\sigma_1; \sigma_2$ is defined by the composition of the corresponding mappings of these morphisms.

Let us prove each of the restrictions concerning the definition of extension morphism.

- First, $\sigma_{1_{ch}}; \sigma_{2_{ch}}$ must be injective, and $\sigma_{2_{ac}}; \sigma_{1_{ac}}$ must be surjective; this is easy to show, since the composition of injective mappings is injective, and the composition of surjective mappings is surjective.
- It is clear that since both $\sigma_{1_{ch}}$ and $\sigma_{2_{ch}}$ preserve the sorts and kinds of channels, so does the composition $\sigma_{1_{ch}}; \sigma_{2_{ch}}$.
- We have as hypotheses that there exist two formulas α_1 and α_2, referring to variables in $(V_2 - \sigma_{1_{ch}}(V_1))$ and $(V_3 - \sigma_{2_{ch}}(V_2))$ respectively, and such that $\Phi_{\mathcal{ADT}} \vdash Init_{P_2} \Leftrightarrow \underline{\sigma_1}(Init_{P_1}) \wedge \alpha_1$ and $\Phi_{\mathcal{ADT}} \vdash Init_{P_3} \Leftrightarrow \underline{\sigma_2}(Init_{P_2}) \wedge \alpha_2$; moreover, both these formulas are "satisfiable", in the sense that $\Phi_{\mathcal{ADT}} \vdash \exists \overline{v_1} : \alpha_1(\overline{v_1})$ and $\Phi_{\mathcal{ADT}} \vdash \exists \overline{v_2} : \alpha_2(\overline{v_2})$. We must show that there exists a formula α_3, using only variables that are contained in $(V_3 - \sigma_{1_{ch}}; \sigma_{2_{ch}}(V_1))$, such that $\Phi_{\mathcal{ADT}} \vdash \exists \overline{v_3} : \alpha_3(\overline{v_3})$ and $\Phi_{\mathcal{ADT}} \vdash Init_{P_3} \Leftrightarrow \underline{\sigma_1; \sigma_2}(Init_{P_1}) \wedge \alpha_3$. We propose $\alpha_3 \hat{=} \sigma_{2_{ch}}(\alpha_1) \wedge \alpha_2$.
  - The fact that α_3 refers only to variables in $V_3 - \sigma_{1_{ch}}; \sigma_{2_{ch}}(V_1))$ is obvious.
  - Let us prove that α_3 is satisfiable. First, since α_1 is satisfiable, so is $\sigma_{2_{ch}}(\alpha_1)$ (satisfiability is preserved under injective language translation). Second, it is easy to see that $\sigma_{2_{ch}}(\alpha_1)$ and α_2 refer to disjoint sets of variables; therefore (and since the only free variables allowed in initialisation conditions are the ones corresponding to channels), the safisfiability of the conjunction $\sigma_{2_{ch}}(\alpha_1) \wedge \alpha_2$ is guaranteed.
  - Let us now prove that $\Phi_{\mathcal{ADT}} \vdash Init_{P_3} \Leftrightarrow \underline{\sigma_1; \sigma_2}(Init_{P_1}) \wedge \alpha_3$. We know that $\Phi_{\mathcal{ADT}} \vdash Init_{P_3} \Leftrightarrow \underline{\sigma_2}(Init_{P_2}) \wedge \alpha_2$, and that $\Phi_{\mathcal{ADT}} \vdash Init_{P_2} \Leftrightarrow \underline{\sigma_1}(Init_{P_1}) \wedge \alpha_1$. Combining these two hypotheses, we straightforwardly get that $\Phi_{\mathcal{ADT}} \vdash Init_{P_3} \Leftrightarrow \underline{\sigma_2}(\underline{\sigma_1}(Init_{P_1}) \wedge \alpha_1) \wedge \alpha_2$, which leads us to $\Phi_{\mathcal{ADT}} \vdash Init_{P_3} \Leftrightarrow (\underline{\sigma_2}(\underline{\sigma_1}(Init_{P_1})) \wedge \underline{\sigma_2}(\alpha_1)) \wedge \alpha_2)$, as we wanted.
- We have to prove that for every $g \in \Gamma_3$ such that $\sigma_{2_{ac}}; \sigma_{1_{ac}}(g)$ is defined, and for every $v \in V_1$, if $\sigma_{1_{ch}}; \sigma_{2_{ch}}(v) \in D_3(g)$, then $v \in D_1(\sigma_{1_{ac}}; \sigma_{2_{ac}}(g))$. This is straightforward, thanks to our hypotheses regarding frame preservation of morphisms σ_1 and σ_2.
- To prove that the composition of the morphisms σ_1 and σ_2 weakens both the lower and the upper bound guards is also straightforward.
- We have as hypotheses that:
  - for every $g \in \Gamma_2$ such that $\sigma_{1_{ac}}(g)$ is defined, there exists a formula α_1 whose referring primed variables are contained in $(V_2' - \sigma_{1_{ch}}(V_1)')$ such that: $\Phi_{\mathcal{ADT}} \vdash \exists \overline{v_1} : \alpha_1(\overline{v_1})$ and $\Phi_{\mathcal{ADT}} \vdash \underline{\sigma}(L_1(\sigma_{1_{ac}}(g))) \Rightarrow (R_2(g) \Leftrightarrow \underline{\sigma_1}(R_1(\sigma_{1_{ac}}(g))) \wedge \alpha_1)$,
  - for every $g \in \Gamma_3$ such that $\sigma_{2_{ac}}(g)$ is defined, there exists a formula α_2 whose referring primed variables are contained in $(V_3' - \sigma_{2_{ch}}(V_2)')$ such

that: $\Phi_{\mathcal{ADT}} \vdash \exists \overline{v_2} : \alpha_2(\overline{v_2})$ and $\Phi_{\mathcal{ADT}} \vdash \underline{\sigma}(L_2(\sigma_{2_{ac}}(g))) \Rightarrow (R_3(g) \Leftrightarrow \underline{\sigma_2}(R_2(\sigma_{2_{ac}}(g))) \wedge \alpha_2)$.

Let $g \in \Gamma_3$ such that $\sigma_{2_{ac}}; \sigma_{1_{ac}}(g)$ is defined. We have to find a formula α_3 whose referring primed variables are contained in $(V_3' - \sigma_{1_{ch}}; \sigma_{2_{ch}}(V_1)')$ such that: $\Phi_{\mathcal{ADT}} \vdash \exists \overline{v_3} : \alpha_3(\overline{v_3})$ and $\Phi_{\mathcal{ADT}} \vdash \underline{\sigma}(L_1(\sigma_{2_{ac}}; \sigma_{1_{ac}}(g))) \Rightarrow (R_3(g) \Leftrightarrow \underline{\sigma_1; \sigma_2}(R_1(\sigma_{2_{ac}}; \sigma_{1_{ac}}(g))) \wedge \alpha_3)$. We propose $\alpha_3 \hat{=} \sigma_{2_{ch}}(\alpha_1) \wedge \alpha_2$. The justification of the "satisfiability" of α_3 is justified, as for the case of the initialisation, by the fact that both $\sigma_2(\alpha_1)$ and α_2 are "satisfiable", and they refer to disjoint sets of variables. Proving that $\Phi_{\mathcal{ADT}} \vdash \underline{\sigma}(L_1(\sigma_{2_{ac}}; \sigma_{1_{ac}}(g))) \Rightarrow (R_3(g) \Leftrightarrow \underline{\sigma_1; \sigma_2}(R_1(\sigma_{2_{ac}}; \sigma_{1_{ac}}(g))) \wedge \alpha_3)$ is also simple; having in mind that $\underline{\sigma}(L_1(\sigma_{2_{ac}}; \sigma_{1_{ac}}(g)))$ is stronger than $\underline{\sigma}(L_2(\sigma_{2_{ac}}(g)))$ and $L_3(g)$, we can "expand" $R_3(g)$ into $\underline{\sigma_2}(R_2(\sigma_{2_{ac}}(g))) \wedge \alpha_2)$, and this into $(\underline{\sigma_1; \sigma_2}(R_1(\sigma_{2_{ac}}; \sigma_{1_{ac}}(g)) \wedge \alpha_1) \wedge \alpha_2)$, obtaining what we wanted.

The rationale behind the definition of extension morphisms is the characterisation of the substitutability principle (a property that can be shown to fail for invasive superposition as defined in [6]). The following result shows that, if there exists an extension morphism σ between two designs P_1 and P_2 (and this extension is realisable), then all behaviours exhibited by P_1 are also exhibited by P_2. Since superposition morphisms, used as a representation of "clientship" (strictly, the existence of a superposition morphism between two designs indicates that the first is part of the second, as a component is part of a system when the first is used by the system), restrict the behaviours of superposed components, it is guaranteed that all behaviours exhibited by a component when this becomes part of a system will also be exhibited by an extension of this component, if replaced by the first one in the system. Of course, one can also obtain *more behaviours*, resulting from the explicit use of new actions of the component. But if none of the new actions are used, then the extended component behaves exactly as the original one.

Theorem 2. *Let P_1 and P_2 be two CommUnity designs, and $\sigma : P_1 \to P_2$ an extension morphism between these designs. Then, every run of P_1 can be embedded in a corresponding run of P_2.*

Proof. For this theorem, we consider a semantics based on runs, i.e., infinite sequences of interpretations such that they all coincide on the interpretation for $\mathcal{ADT}$, the first interpretation in the sequence satisfies the initial condition and any pair of consecutive interpretations in the sequence either only differ in the interpretation of input variables (stuttering), or they are in the "'consequence" relation for one of the actions of the component.

Let $P1$ and P_2 be two CommUnity designs, and $\sigma : P_1 \to P_2$ an extension morphism between these designs. Let $s = s_0, s_1, s_2, \ldots$ be a run for P_1. We will inductively construct a sequence $s' = s'_0, s'_1, s'_2, \ldots$ which is a run for P_2, and such that, for all i, $(s'_i)_{|\sigma_{ch}(V_1)} \equiv s_i$, i.e., the reduct of each s'_i to the symbols originating in P_1 coincides with the interpretation s_i.

– Base case. The initialisation of P_2 is of the form $\underline{\sigma}(Init_{P_1}) \wedge \alpha$, with α a formula satisfying $\Phi_{\mathcal{ADT}} \vdash \exists \overline{v} : \alpha(\overline{v})$, and whose variables are "new vari-

ables", in the sense that they differ from those appearing in the initialisation of P_1. Then, there exists an interpretation I_α of the variables in α that makes it true. We define s'_0 as the extension of the interpretation s_0, appropriately translated via σ, with the interpretation I_α for the remaining variables. Clearly, this interpretation satisfies the initial condition of P_2, and its reduct to the language of P_1 coincides with s_0.

- Inductive step. Assuming that we have already constructed a prefix $s' = s'_0, s'_1, s'_2, \ldots, s'_i$ of the run s', we build the interpretation s'_{i+1} in the following way. We know that s_{i+1} is in one of the following two cases:
  - s_{i+1} is reached from s_i via stuttering. In such a case, we define $s'_{i+1} \hat{=} s'_i$, and clearly, by inductive hypothesis, we have that the reduct of s'_{i+1} to the variables of P_1 coincides with s_{i+1}, and (s'_i, s'_{i+1}) are in the "stuttering relationship".
  - there exists some action $g \in \Gamma_1$ such that (s_i, s_{i+1}) are in the consequence relationship corresponding to g. If this is the case, notice that, under the "stronger guard" $L_1(g)$, the assignment of an action g_2 in $\sigma_{ac}^{-1}(g)$ (which is nonempty, since σ_{ac} is surjective) is of the form $\underline{\sigma}(R_1(\sigma_{ac}(g))) \wedge \alpha)$, for a formula α referring only to the primed versions of new variables. Since we know that $\Phi_{\mathcal{ADT}} \vdash \exists \overline{v} : \alpha(\overline{v})$, there exists an interpretation I_α of the variables in α that makes is true. We define s'_{i+1} as the extension of the interpretation s_{i+1}, with symbols appropriately translated via σ, with I_α for the interpretation of the remaining variables. It is straightforward to see that (s'_i, s'_{i+1}) are in the consequence relation of g_2, and that the reduct of s'_{i+1} to the variables originating in P_1 coincides with s_{i+1}.

3.1 Replacing Components by Extensions in Configurations

The intention of extension morphisms is to characterise component extension, respecting the substitutability principle. One might then expect that, if a component C can be "plugged" into an architecture of components, then we should be able to plug an extension C' in the architecture, instead of C. Due to the restrictions for valid extension, it can be guaranteed that a design in a well formed diagram can be replaced with an extension of it, preserving the wellformedness of the diagram (although it is necessary to consider an "open system" semantics, since extensions might introduce new input variables, which would be "disconnected" after the replacement). Moreover, we can also prove that the colimit of the new diagram (where a component was replaced by an extension of it) is actually an extension of the colimit of the original diagram. This basically means that the joint behaviour of the original system is *augmented* by the extension of a component, but never restricted (i.e., the resulting system exhibits all the behaviours of the original one, and normally also more behaviours).

We are not in a similar situation when combining extension and refinement. As we mentioned, refinement plays an important role in the implementation of higher-order connectors, since it allows us to "instantiate" roles with actual components. Roles, as the `Abstract_Sender` example, specify the minimum requirements that have to be satisfied in order to be able to plug components using

a higher-order connector. Notice that, in particular, the interval determined by the guards of actions of the role has to be preserved or reduced by the actual parameter, i.e., the component with which the role is instantiated. Consider, for instance, the case of the `Abstract_Sender`. As we mentioned, the design `Sender` refines this more abstract `Abstract_Sender`, and therefore we can instantiate the abstract sender with the concrete one. Moreover, `RES_Sender`, an extension of `Sender`, also refines `Abstract_Sender`, so it also can instantiate this role. However, since extensions weaken both guards, it is not difficult to realise that, if a component B refines a component A, and B' is an extension of B, then it is not necessarily the case that B' also refines A. With respect to configurations of systems, this means that, when replacing components by corresponding extensions, one might lose the possibility of applying or using some higher-order connectors.

Although this might seem an unwanted restriction, it is actually rather natural. The conditions imposed by roles of a higher-order connector are a kind of "rely-guarantee" assumptions. When extending a component we might lose some properties the role requires for the component.

4 An Example Using Extension

Let us go back to our example of communicating components via an unreliable channel. As we explained in previous sections, we would like to superpose behaviour on the existing architecture, to make the communication reliable by implementing a reset in the communication when packets are lost. The mechanism we used was very simple, and required a "reset" operation on the sender, which, as we discussed, can be achieved by component extension. In order to complete the enhanced architecture to implement the reset acknowledgement mechanism, we need a monitor that, if it detects a missing packet, issues a call for reset. The idea is that, if a message is not what the monitor expected (characterised by the `msg-exp`), then it will go to a "reset" cycle, and wait to see if the expected packet arrives. If the expected packet arrives, then the component will start waiting for the next packet. Notice that, for the sake of simplicity, we assume that the communication between the monitor and the extended sender is reliable. The monitor used for this task is shown in Figure 10. The final architecture for the system is shown in Figure 11.

Notice that, since the superposed monitor is *spectative*, we can guarantee that, if the augmented system works without the need for reset in the communication, i.e., no messages are lost, then its behaviour is exactly the same as the one of the original architecture with unreliable communication.

5 Related Work

The original work on CommUnity took its inspiration from languages like Unity [3] and IP [8] and on related software engineering research [12] using superimposition/superposition as structuring principles. Recently, research by Katz and

```
Design RES_Monitor
in
    msg: int
out
    msg-rst: int
prv
    msg-exp: int
    w: bool
init
    msg-rst=0 ∧ msg-exp=0 ∧ w=true
do
 rec1[msg-exp]: w ∧ msg-exp=msg → msg-exp'=msg-exp+1
 rec2[msg-exp,msg-rst,w]: w ∧ msg-exp≠msg →
                        msg-exp'=msg-exp ∧ msg-rst'=msg-exp ∧ w'=false
 rec3[msg-exp]: ¬w → msg-exp'=msg-exp
 res[w]: ¬w → w'=true
```

Fig. 10. A monitor for detecting lost packets.

his collaborators has recognised the usefulness of superimposition as a way of characterising aspects [13,11,22]. Especially in [11], there is a recognition of the same principles we espouse in this paper, namely that aspects should be characterised and applied at the architectural level of software development. Aspects are seen as patterns to be applied to underlying architectures (which may already have been modified by the application of previous concerns), based on specifications of the aspects. These specifications include descriptions of components and connectors used to define the aspect, as well as "dummy" components defining required services in order to be able to apply the aspect. The relationships and structuring mechanisms and the instantiation of the "dummy" components are explained in terms of superimpositions.

The motivation for our research is very similar, we want to lift the treatment of aspects to the architectural level and view the application of aspects to the design of some underlying system as the application of a transformation defined by the aspect design to the underlying architecture, resulting in an augmented architecture. The application of various aspects can be seen as the application of a sequence of transformations to the underlying architecture (see [2]). This raises concerns analogous to those discussed in [11]. In order to develop this framework, we found it necessary to come to a better understanding of invasive superpositions in the context of CommUnity. In particular, we needed to characterise a structured use of invasive superpositions, which allows arbitrary changes breaking encapsulation of the component being superimposed. As noted earlier, this problem has also arisen in the context of object oriented design and programming, resulting in the various proposals attempting to characterise the concept of substitutivity ([15]). We believe that this proposal provides a more solid foun-

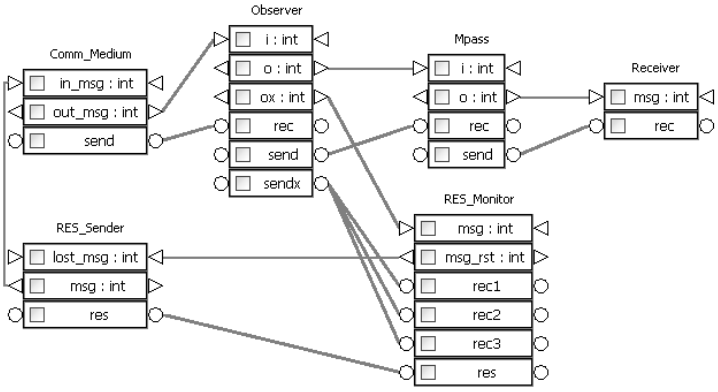

Fig. 11. The architecture of the system, with the reset mechanism.

dation for substitutivity, one that is better structured and more amenable to analysis.

Of course, the work reported in [16,17] is related to our work, both because it is based on CommUnity and because it recognises that the concept of higher order connector (a kind of parameterised connector that can be applied to other connectors to obtain more sophisticated connectors) can be used to characterise certain aspects. Again the emphasis is on using the specification of the aspect, as a higher order connector, to transform an existing architectural pattern in order to apply the aspect. As we demonstrate in this paper, some interesting aspects cannot be characterised in terms of this mechanism alone and it is necessary to consider transformations that apply uniformly to connectors and to the components they connect. Furthermore, some of the transformations require the use of invasive superpositions, as in the main example used in this paper. This is a subject that has received very little scrutiny in the CommUnity literature.

6 Conclusion

We have studied a special kind of invasive superposition for the characterisation of extensions between designs in the CommUnity architecture design language. This kind of morphism, that we have defined with special concern regarding the substitutability principle [15] (an essential property associated with sound component extension), allows us to complement the refinement and (regulative) superposition morphisms of CommUnity, and obtain a suitable formal framework to characterise certain *aspects*, in the sense of aspect oriented software development. We have argued that some useful aspects require extensions on the components, as well as in the connectors, and therefore the introduced extension morphisms are necessary. Also, having the possibility of extending components provides us a way of balancing the distribution of augmented behaviour in the

connectors and the components, which would otherwise be put exclusively on the connector side (typically by means of higher-order connectors).

We illustrated the need for extension morphisms by means of a simple case study based on the communication of two components via an unreliable channel. We then augmented the behaviour of this original system with a fault tolerance aspect for making the communication reliable, which required the extension of components, as well as the use of higher-order connectors. This small case study also allowed us to illustrate the relationships and combined use of extension, superposition and refinement morphisms.

As we mentioned before, This problem has also arisen in the context of object oriented design and programming, attempting to define various forms of inheritance, resulting in the proposals attempting to characterise the concept of substitutability [15,21]. We believe that this proposal provides a more solid foundation for substitutivity, one that is better structured and more amenable to analysis. The definition of extension in CommUnity that we introduced has been partly motivated by the definitions and proof obligations used to define the structuring mechanisms in B [1,4], that justifies the notion of substitutivity and provides a structuring principle for augmenting components by breaking the encapsulation of the component.

References

1. J.-R. Abrial, *The B-Book, Assigning Programs to Meanings*, Cambridge University Press, 1996.
2. N. Aguirre, T. Maibaum and P. Alencar, *Abstract Design with Aspects*, submitted, 2005.
3. K. Chandy, J. Misra, *Parallel Program Design - A Foundation*, Addison-Wesley, 1988.
4. T. Dimitrakos, J. Bicarregui, B. Matthews and T. Maibaum, *Compositional Structuring in the B-Method: A Logical Viewpoint of the Static Context*, in Proceedings of the International Conference of B and Z Users ZB2000, York, United Kingdom, LNCS, Springer-Verlag, 2000.
5. J. Fiadeiro, *Categories for Software Engineering*, Springer, 2004.
6. J. Fiadeiro and T. Maibaum, *Categorical Semantics of Parallel Program Design*, in Science of Computer Programming 28(2-3), Elsevier, 1997.
7. R. Filman, T. Elrad, S. Clarke and M. Aksit, *Aspect-Oriented Software Development*, Addison-Wesley, 2004.
8. N. Francez and I. Forman, *Interacting Processes*, Addison-Wesley, 1996.
9. D. Garlan and R. Monroe and D. Wile, *ACME: An Architecture Description Interchange Language*, in Proceedings of CASCON'97, Toronto, Ontario, 1997.
10. J. Goguen, *Categorical Foundations for General System Theory*, in F.Pichler and R.Trappl (eds), Adavaces in Cybernetics anda Systems Research, Transcripta Books, pages 121-130, 1973.
11. M. Katara and S. Katz, *Architectural Views of Aspects*, in Proceedings of International Conference on Aspect-Oriented Software Design AOSD 2003, 2003.
12. S. Katz, *A Superimposition Control Construct for Distributed Systems*, ACM Transactions on Programming Languages and Systems, 15:337-356, 1993.

13. S. Katz and J. Gil, *Aspects and Superimpositions*, ECOOP Workshop on Aspect Oriented Programming, 1999.
14. G. Kiczales, *An overview of AspectJ*, in Proceedings of the European Conference on Object-Oriented Programming ECOOP 2001, Lecture Notes in Computer Science, Budapest, Hungary, Springer-Verlag, 2001.
15. B. Liskov and J. Wing, *A Behavioral Notion of Subtyping*, ACM Transactions on Programming Languages and Systems, Vol 16, No 6, ACM Press, November 1994.
16. A. Lopes, M. Wermelinger and J. Fiadeiro, *Higher-Order Architectural Connectors*, ACM Transactions on Software Engineering and Methodology, vol. 12 n. 1, 2003.
17. A. Lopes and J. Fiadeiro, *Superposition: Composition vs. Refinement of Non-Deterministic, Action-Based Systems*, in Formal Aspects of Computing, Vol. 16, N. 1, Springer-Verlag, 2004.
18. D. Luckham, J. Kenney, L. Augustin, J. Vera, D. Bryan and W. Mann, *Specification and Analysis of System Architecture Using Rapide*, IEEE Transactions on Software Engineering, Special Issue on Software Architecture, 21(4), 1995.
19. J. Magee, N. Dulay, S. Eisenbach and J. Kramer, *Specifying Distributed Software Architectures*, in Proceedings of the 5th European Software Engineering Conference (ESEC '95), Sitges, Spain, Lecture Notes in Computer Science, Springer-Verlag, 1995.
20. N. Medvidovic, P. Oreizy, J. Robbins and R. Taylor, *Using Object-Oriented Typing to Support Architectural Design in the C2 Style*, in Proceedings of the ACM SIGSOFT '96 Fourth Symposium on the Foundations of Software Engineering, ACM SIGSOFT, San Francisco, CA, 1996.
21. B. Meyer, *Object-Oriented Software Construction*, Second Edition, Prentice Hall, 2000.
22. M. Sihman and S. Katz, *Superimpositions and Aspect-Oriented Programming*, BCS Computer Journal, vol. 46, 2003.
23. M. Wermelinger, A. Lopes and J. Fiadeiro, *A Graph Based Architectural (Re)configuration Language*, in ESEC/FSE'01, V.Gruhn (ed), ACM Press, 2001.
24. M. Wermelinger, C. Oliveira, *The CommUnity Workbench*, In Proc. of the 24th Intl. Conf. on Software Engineering, page 713. ACM Press, 2002.