Available online at www.sciencedirect.com

SCIENCE DIRECT®

Electronic Notes in Theoretical Computer Science 97 (2004) 241–258

Electronic Notes in Theoretical Computer Science

www.elsevier.com/locate/entcs

Adding Mobility to Software Architectures

Antónia Lopes[1]

Department of Informatics
Faculty of Sciences, University of Lisbon
Campo Grande, 1749-016 Lisboa, Portugal

José Luiz Fiadeiro[2]

Department of Computer Science
University of Leicester
University Road, Leicester LE1 7RH, UK

Abstract

Mobility has become a new factor of complexity in the construction and evolution of software systems. In this paper, we show how architectural description techniques can be enriched to support the incremental and compositional construction of location-aware systems. In our approach, the process of integrating and managing mobility in architectural models of distributed systems is not intrusive on the options that are made at the level of the other two dimensions - computation and coordination. This means that a true separation of concerns between computation, coordination and distribution can be enforced at the level of architectural models.

Keywords: Software Architecture, mobility, refinement and connectors.

1 Introduction

The advent of Mobile Computing has opened completely new ways for software systems to be conceived and developed. Under this new style of computation, systems can be designed and programmed in more sophisticated ways by taking advantage of the fact that data, code and agents can change location during execution. For instance, remote execution of agents and local interactions are

[1] Email: mal@di.fc.ul.pt
[2] Email: jose@fiadeiro.org

doi:10.1016/j.entcs.2004.04.039

preferable to remote interaction when latency is high and the interaction is extensive; dynamic update of software by mobile agents is much simpler than the typical static installations of new versions as performed by users. Hence, as argued in [15] and [11], advantages over traditional forms of distribution can be obtained in terms of efficiency, as well as flexibility.

However, these advantages come at a price. While building distributed applications over static configurations has been proving to be challenging enough for existing software development methods and techniques, mobility introduces an additional factor of complexity due to the need to account for the changes that can occur, at run time, at the level of the topology over which components perform computations and interact with one another.

In recent years, architecture-based approaches have proved to contribute to the taming of the complexity of designing and constructing highly distributed systems. By enforcing a strict separation of concerns between computations, as performed by individual components, and the mechanisms that, through explicit connectors, coordinate the way components interact, architecture description languages support a gross modularisation of systems that can be progressively refined, in a compositional way, by adding detail to the way computations execute in chosen platforms and the communication protocols that support coordination. Compositionality means that refinements over one of the dimensions can be performed without interfering with the options made already on the other one. The same applies to evolution: connectors can be changed or replaced without interfering with the code that components execute locally to perform the computations required by system services, and the code can itself be evolved, e.g. optimised, without interfering with the connectors, e.g. with the communication protocol being used for interconnecting components.

In this paper, we report on work that we are currently pursuing within the IST-project AGILE – Architectures for Mobility – with the aim of conciliating the architectural approach with the need to account for distribution and mobility. More precisely, we investigate how the level of separation and compositionality that has been obtained for computation and coordination can be extended to distribution so that one can support the construction and evolution of location-aware architectural models by superposing explicit connectors that handle the mobility aspects while preserving the "static" properties that can be inferred from a location-transparent view of the architecture.

For this purpose, we capitalize on our previous work on the development of primitives through which mobility can be addressed explicitly in architectural models [7]. In particular, we make use of a design primitive – *distribution connectors* – that, for mobility, fulfills a role similar to the one played by architectural connectors [2] in externalising the interactions among components. In this setting, we show that location-transparent architectural models can be made location-aware through the superposition of distribution connectors over components and connectors. In this way, it becomes possible, for instance, to regulate the dependency of components and connectors on properties of locations and the network infrastructure that connects them, without interfering with the computation and coordination aspects that account for location-independent properties.

We present our approach over CommUnity, a language that we have been developing to support architecture description at its more basic level, i.e. having in mind to facilitate the analysis and formalisation of architectural semantic primitives and not the modelling of software architectures themselves. In this sense, CommUnity is a "prototype" language, stripped to the bare minimum that supports the externalisation of coordination from computations. CommUnity was recently extended in order to support the description of mobile systems [7]. In this paper, we use this extension to describe mobile architectures and to illustrate how it is possible to take advantage of a three-way separation of concerns among computation, coordination, and distribution.

2 Architectural Descriptions in CommUnity

Location-transparency is usually considered to be an important abstraction principle for the design of distributed systems. It assumes that the infrastructure masks physical and logical distribution of the system, and provides location-transparent communication and access to resources: components do not need to know where the components to which they are interconnected reside and execute their computations, nor how they themselves move across the distribution network.

Traditionally, architectural approaches to software design also adhere to this principle; essentially, they all share the view that system architectures are structured in terms of components and architectural connectors. Components are computation loci while connectors, superposed on certain components or groups of components, explicitly define the way these components interact.

In CommUnity, components are designed in terms of a set of channels V (declared as input, output or private) and a set of actions Γ (shared or private) that, together, constitute what we call signatures. More precisely,

- Input channels are used for reading data from the environment; the component has no control on the value that are made available in such channels. Output and private channels are controlled locally by the component. Output channels allow the environment to read data produced by the component.

- Private actions represent internal computations in the sense that their execution is uniquely under the control of the component whereas shared actions represent possible interactions between the component and the environment.

The computational aspects are described in CommUnity by associating with each action g an expression of the form

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

where

- *D(g)* consists of the local channels into which executions of the action can place values. We often omit this set because it can be inferred from the assignments in *R(g)*. Given a private or output channel v, we will also denote by $D(v)$ the set of actions g such that $v \in D(g)$. We denote by $F(g)$ the set of channels that are in *D(g)* or are used in *L(g)*, *U(g)* or *R(g)*.
- *L(g)* and *U(g)* are two conditions that establish the interval in which the enabling condition of any guarded command that implements g must lie. The enabling condition is fully determined only if *L(g)* and *U(g)* are equivalent, in which case we write only one condition.
- *R(g)* is a condition on V and $D(g)'$ where by $D(g)'$ we denote the set of primed local channels from the write frame of g. As usual, these primed channels account for references to the values that the channels take after the execution of the action. These conditions are usually a conjunction of implications of the form *pre* $\Rightarrow$ *pos* where *pre* does not involve primed channels. They correspond to pre/post-condition specifications in the sense of Hoare. When *R(g)* is such that the primed channels are fully determined, we obtain a conditional multiple assignment, in which case we use the notation that is normally found in programming languages. When the write frame *D(g)* is empty, *R(g)* is tautological, which we denote by *skip*.

Designs in CommUnity are defined over a collection of data types that are used for structuring the data that the channels transmit and define the operations that perform the computations that are required. Hence, the choice of data types determines, essentially, the nature of the elementary computations that can be performed locally by the components, which are abstracted as operations on data elements. For simplicity, we assume a fixed collection of data types, i.e. we shall not discuss the process of data refinement that needs to be involved when mapping designs and their interconnections to the platforms that support computations and coordination. In order to remain independent of any specific language for the definition of these data types, we take them in the form of a first-order algebraic specification. That is to say, we assume a data signature $\langle S, \Omega \rangle$, where S is a set (of sorts) and Ω is a $S^* \times S$-indexed family of sets (of operations), to be given together with a collection Φ of first-order sentences specifying the functionality of the operations.

In CommUnity, the separation between "computation" and "coordination" is taken to an extreme in the sense that the definition of the individual components of a system can be completely separated from the interconnections through which these components interact. The model of interaction between components is based on action synchronisation and exchange of data through input and output channels. These are standard means of interconnecting software components. What distinguishes CommUnity from other parallel program design languages is the fact that such interactions between components have to be made explicit by providing the corresponding name bindings; no implicit interaction can be inferred from the use of the same name in different signatures; such coincidences are treated as accidental and disambiguated whenever the system is looked at in its globality.

The following example illustrates how CommUnity supports this separation of concerns in terms of a very simple sender-receiver system. The sender produces, in one go, words of bits that are then transmitted one by one to the receiver through synchronous message passing.

An abstract architecture model of this system is depicted as follows:

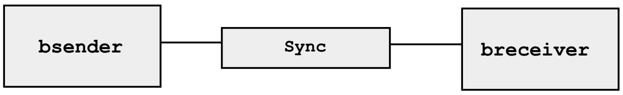


Fig. 1.

In CommUnity, the model consists of the (configuration) diagram presented in Figure 2 where

```
design bsender is
out    obit:bit
prv    word:array(N,bit), k:nat, rd:bool
do     new-w[ k,word] : k=N→ k'=0
[]     new-b: ¬rd∧k<N→ rd:=true‖obit:=word[ k] ‖k:=k+1
[]     send: rd→ rd:=false

design breceiver is
in     ibit:bit
out    w:array(N,bit), k:nat
prv    rd:bool, word:array(N,bit)
do     rec: k<N→ word[ k] :=ibit‖k:=k+1‖rd:=false
[] prv save-w: ¬rd∧k=N→ rd:=true‖w:=word
[]     new-w: rd∧k=N→ rd:=false‖k:=0
```

The design *bsender* models a component that repeatedly produces new words (action *new-w*) and transmits their bits one by one: it places a bit in the output channel *obit* and then waits for an acknowledgement that the bit has been read by the environment (action *send*).

The design *breceiver* models a component that receives bits through the input channel *ibit* and produces words of N-bits with them that makes available to the environment through the output channel *w*.

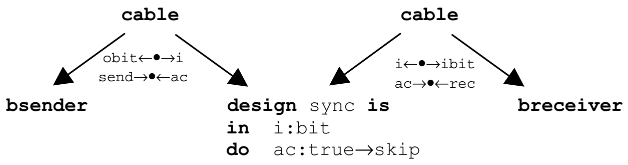


Fig. 2.

In order to define that the two components communicate through synchronous message passing, we have to make explicit the synchronisation of the actions *send* of *bsender* and *rec* of *breceiver* and the i/o interconnection of the channels *obit* and *ibit* used for the transmission of bits. The design *cable*, given below, models a component with no computational behaviour and whose role is to perform the bindings between channels and establish the rendez-vous required by the interconnection. The bindings and the rendez-vous themselves are expressed through the arrows.

```
design cable is
in     i:bit
do     a:true→skip
```

In fact, such arrows are mathematical objects that, together with designs, define the category **DSGN** (see below). Configuration diagrams in CommUnity are simply categorical diagrams in **DSGN**. Taking the colimit of a diagram collapses the configuration into an object of the category by internalising all the interconnections, thus delivering a design for the system as a whole (more details and motivations on the categorical approach can be found in [6]). Given a design P_i we denote

- the set of channels by X_i, the set of input, output and private channels by $in(X_i)$, $out(X_i)$ and $prv(X_i)$, respectively, and the union of output and private channels by $local(X_i)$;
- the set of actions by Γ_i and the set of shared and private actions by $sh(\Gamma_i)$ and $prv(\Gamma_i)$, respectively.

Definition 2.1 A morphism $\sigma : P_1 \to P_2$ consists of a total function $\sigma_{ch} : X_1 \to X_2$ and a partial mapping $\sigma_{ac} : \Gamma_2 \to \Gamma_1$ satisfying:

1. for every $x \in X_1$:

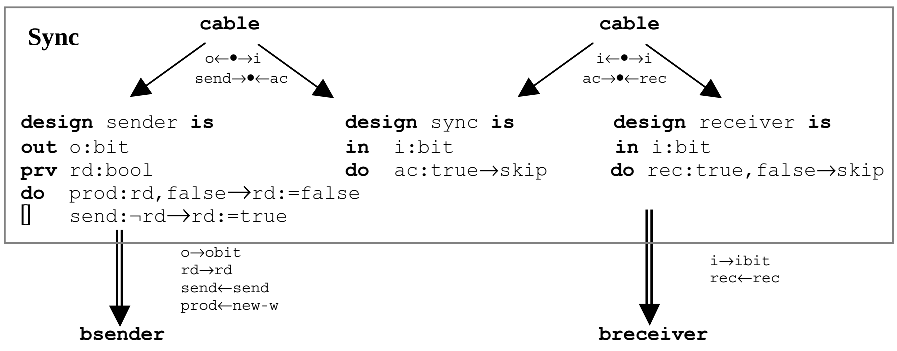
(a) $sort_2(\sigma_{ch}(x)) = sort_1(x)$
(b) if $x \in out(X_1)$ then $\sigma_{ch}(x) \in out(X_2)$
(c) if $x \in prv(X_1)$ then $\sigma_{ch}(x) \in prv(X_2)$
(d) if $x \in in(X_1)$ then $\sigma_{ch}(x) \in out(X_2) \cup in(X_2)$

2. for every $g \in \Gamma_2$ s.t. $\sigma_{ac}(g)$ is defined
(a) if $g \in sh(\Gamma_2)$ then $\sigma_{ac}(g) \in sh(\Gamma_1)$
(b) if $g \in prv(\Gamma_2)$ then $\sigma_{ac}(g) \in prv(\Gamma_1)$

3. for every $g \in \Gamma_2$ s.t. $\sigma_{ac}(g)$ is defined
(a) $\sigma_{ch}(D_1(\sigma_{ac}(g))) \subseteq D_2(g)$
(b) σ_{ac} is total on $D_2(\sigma_{ch}(x))$ and $\sigma_{ac}(D_2(\sigma_{ch}(x))) \subseteq D_1(x)$, for every $x \in local(X_1)$
(c) $\Phi \models (R_2(g) \Rightarrow \underline{\sigma}(R_1(\sigma_{ac}(g))))$
(d) $\Phi \models (L_2(g) \Rightarrow \underline{\sigma}(L_1(\sigma_{ac}(g))))$
(e) $\Phi \models (U_2(g) \Rightarrow \underline{\sigma}(U_1(\sigma_{ac}(g))))$

where $\models$ means validity in the first-order sense taken over the axiomatisation Φ of the underlying data types. Designs and morphisms constitute a category **DSGN**.

To conclude this overview on how CommUnity supports architectural design, it is important to mention that connectors can be represented as first-class entities through (configuration) diagrams. For instance, the connector that represents uni-directional synchronous transmission of messages can be modelled through the configuration diagram depicted in Figure 3. The connector is used in the construction of the sender-receiver system by establishing the instantiation of its roles *sender* and *receiver* with the components *bsender* and *breceiver*.



```
design sender is
out o:bit
prv rd:bool
do  prod:rd,false→rd:=false
[]  send:¬rd→rd:=true
```

```
design sync is
in  i:bit
do  ac:true→skip
```

```
design receiver is
in i:bit
do rec:true,false→skip
```

Fig. 3.

Each role of a connector defines the behaviour that is expected of each of the partners of the interaction being specified. The instantiation of a role with a component is possible if and only if the component fulfills the obligations the role determines. In CommUnity, instantiation of roles is also defined over a category of designs but using a different notion of morphism that captures behavioural refinement. This notion of refinement permits exchanging components and even connectors but not change the overall structure of the system. Details on these refinement morphisms between designs and between connectors can be found in [6] and [8], respectively.

3 Making Designs Location-aware

CommUnity was extended in [7] with features that support the description of systems with mobile components and location-aware patterns of computation or interaction. This extension is based on an explicit representation of the distribution topology: the intended "space of mobility" is constituted by the set of possible values of a special data type *Loc*, the properties of which should be part of the data type specification mentioned in the previous section. This type should model the positions of the space in a way that is considered to be adequate for the particular application domain in which the system is or will be embedded. The only requirement that we make is for a special location – $\perp$ – to be distinguished. The intuition is that $\perp$ is a special position of the space to locate entities that can communicate with any other entity in a location-transparent manner. The role of $\perp$ will be discussed further below.

In order to model systems that are location-aware, we enrich signatures with a set of location variables L_λ typed over *Loc*, and assign to each output and private channel, and each action, a set of location variables:

- Each local channel x is associated with a location l. We make this assignment explicit by simply writing $x@l$ in the declaration of x. The intuition is that the value of l indicates the current position of the space where the values of x are made available. A modification in the value of l entails the movement of x as well as of the other channels and actions located at l.
- Each action name g is associated with a set of locations $\Lambda(g)$ meaning that the execution of action g is distributed over those locations. In other words, the execution of g consists of the synchronous execution of a guarded command in each of these locations. Guarded commands, that may now include assignments involving the reading or writing of location variables, are associated with located actions, i.e. pairs $g@l$, for $l \in \Lambda(g)$.

Locations can be declared as input or output in the same way as channels. Input locations are read from the environment and cannot be modified by the component and, hence, the movement of any constituent located at an input location is under the control of the environment. Output locations can only be modified locally through assignments performed within actions and, hence, the movement of any constituent located at an output location is under the control of the component.

Each set L_λ has a distinguished output location – λ. This location has the particularity that its value is invariant and given by $\perp$. It is used when one wants to make no commitment as to the location of channels or actions. For instance, input channels are always located at λ because the values that they store are provided by the environment in a way that is location-transparent; their location is determined at configuration time when they are connected to output channels of other components.

Actions uniquely located at λ model activities for which no commitments wrt location-awareness have been made; the reference to λ in these cases is usually omitted. In later stages of the development process, the execution of such actions can be distributed over several locations, i.e. the guarded command associated with $g@\lambda$ can be split in several guarded commands associated with located actions of the form $g@l$, where l is a proper location. Whenever the command associated with $g@\lambda$ has been fully distributed over a given set of locations in the sense that all its guards and effects have been accounted for, the reference to $g@\lambda$ is usually omitted.

In order to illustrate the use of the new primitives, consider the sender-receiver system discussed in Section 2. Suppose that we want to make the receiver a mobile component: once a word defining a location is received, the component should move the execution of the computations to that location. The CommUnity design that models this kind of behaviour is as follows:

```
design mobile_breceiver is
outloc l
in     ibit:bit
out    w@l:array(N,bit), k@l:nat
prv    rd@l:bool, word@l:array(N,bit)
do     rec@l: k<N→ word[ k] :=ibit‖k:=k+1‖rd:=false
[] prv save-w@l: ¬rd∧k=N→ rd:=true‖w:=word
[]     new-w@l:rd∧k=N→rd:=false‖k:=0‖l:=if(loc?(w),loc(w),l)
```

where $loc? : array(N, bit) \to bool$ is an operation on bit arrays that indicates whether the corresponding word, given by $loc : array(N, bit) \to Loc$, is a location.

Suppose in addition that a decision has been made for the sender to be placed at some fixed location of which it is neither aware nor in control. The

CommUnity design that models this kind of behaviour is as follows:

```
design mobile_bsender is
outloc l
out    obit@l:bool
prv    word@l:array(N,bit), k@l:nat, rd@l:bool
do     new-w@l[ word,k] : k=N→ k'=0
[]     new-b@l: ¬rd∧k<N→ rd:=true‖obit:=word[ k] ‖k:=k+1
[]     send@l: rd→ rd:=false
```

The interconnection between these two mobile components can be established through a synchronisation connector as before (see Figure 4). The connector *sync* is the "same" as in Section 2 because, given that the synchronisation is not performed at any specific location but across the network, its action *ac* is uniquely located at λ and, as mentioned before, in such situations we tend to omit the reference to λ. Note that the fact that both designs declare an output location l does not imply any relationship between them as already mentioned; the two locations are not the same, they just happened to be given the same name. The fact that the connector does not bind them means that the two components are not being required to be co-located.

The semantics of the new primitives can be summarised as follows. We distinguish a distribution space that consists of the set of possible values of the given data sort *Loc*, among which we distinguish $\perp$. Two binary relations capture the relevant properties of this space:

- A relation *bt* s.t. $n\ bt\ m$ means that n and m are positions in the space "in touch" with each other. Interactions among components can only take place when they are located in positions that are "in touch" with one another. Because the special location variable λ intends to be a position to locate entities that can communicate with any other entity in a location-transparent manner, we require that the value of λ is always set at configuration time as being $\perp$ and, furthermore, $\perp\ bt\ m$, for every position m.
- A relation *reach* s.t. $n\ reach\ m$ means that position n is reachable from m. Permission to move a component or a group of components is conceded when the new position "is reachable" from the current one.

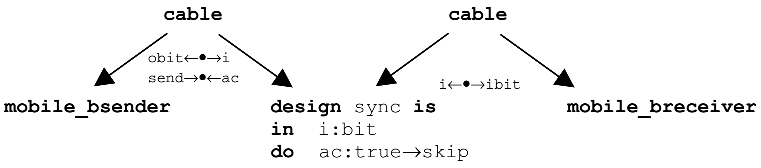


Fig. 4.

In general, the topology of locations is dynamic and, hence, the operational semantics for a program is given in terms of an infinite sequence of relations

$(bt_i, reach_i)_{i \in N}$. The conditions under which a distributed action g can be executed at time i are the following:

- for every $l_1, l_2 \in \Lambda(g), [l_1]^i \; bt_i [l_2]^i$: the execution of g involves the synchronisation of its local actions and, hence, their locations have to be in touch;
- for every $l \in \Lambda(g)$, $g@l$ can be executed, *i.e.*,
  i. for every $x \in F(g@l)$, $[l]^i bt_i [\Lambda(x)]^i$: the execution of $g@l$ requires that every channel in the frame of $g@l$ can be accessed and, hence, l has to be in touch with their locations;
  ii. for every location $l_1 \in D(g@l)$ and $m \in [R(g)]^i(l_1)$, $m \; reach_i \; [l_1]^i$: if a location l_1 can be effected by the execution of $g@l$, every possible new value of l_1 must be a location reachable from the current one.
  iii. the enabling condition of $g@l$ evaluates to true;

  where $[e]^i$ denotes the value of the expression e at time i.

Hence, the synchronisation connector of Figure 4 does not require a specific location for a rendez-vous to take place: it works on the assumption that the locations involved are "in touch".

As before, the configurations that we used above are categorical diagrams. The underlying category in this case is **MDSG** defined below.

Given a mobile design P_i we denote the set of location variables by L_i, the set of input and output locations by, respectively, $inloc(L_i)$ and $outloc(L_i)$.

Definition 3.1 A morphism $\sigma : P_1 \to P_2$ consists of a total function $\sigma_{ch} : X_1 \to X_2$, a partial mapping $\sigma_{ac} : \Gamma_2 \to \Gamma_1$ and a total function $\sigma_{lc} : L_1 \to L_2$ that maps the designated location (λ) of P_1 to that of P_2, and, further to the properties required in definition 2.1, satisfies:

1. for every $x \in X_1$and $l \in L_1$:
   (e) if $l \in outloc(L_1)$ then $\sigma_{lc}(l) \in outloc(L_2)$
   (f) $\sigma_{lc}(\Lambda_1(x)) \subseteq \Lambda_2(\sigma_{ch}(x))$
2. for every $g \in \Gamma_2$ s.t. $\sigma_{ac}(g)$ is defined
   (c) $\sigma_{lc}(\Lambda_1(\sigma_{ac}(g))) \subseteq \Lambda_2(g)$
3. for every $g \in \Gamma_2$ s.t. $\sigma_{ac}(g)$ is defined and $l \in \sigma_{lc}^{-1}(\Lambda(g))$ the same conditions as before but applied to $g@\sigma_{lc}(l)$ and $\sigma_{ac}(g)@l$

The extended designs and morphisms constitute a category **MDSG**.

4 Supporting Incremental Development

As already motivated, our aim is to support the construction of architectural models in an incremental way through, one the one hand, the enforcement of a strict separation of concerns through which computation, coordination and distribution (mobility) can be evolved independently of each other in a non-intrusive way and, on the other hand, a precise notion of refinement through which increments can be made in principled ways. Because, in previous publications, we have addressed these issues for the separation between computation and coordination, we are going to concentrate on the way the distribution aspects need to be handled.

The example that we used in the previous section shows how the extension of CommUnity provides for a finer-grain modelling of architectural design. However, the way we developed the new architecture that takes into account the requirements on mobility is not the one we wish to see supported. Instead of rewriting the components and connectors to take into account locations and the way they are updated, we would like to superpose specific connectors that handle the distribution and mobility aspects. We want to represent them explicitly in the architecture so that they can be refined or evolved independently of the architectural elements (components and connectors) that handle the location-transparent aspects of computation and coordination.

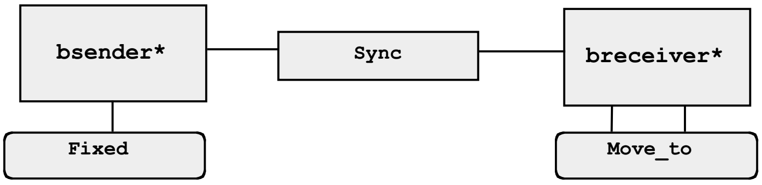


Fig. 5.

More precisely, making use of an additional design primitive – *distribution connector* – we would like to enrich the configuration diagram given in Figure 1 with an explicit representation of the mechanisms that handle this distribution/mobility policy. This mobile sender-receiver system can be structured as depicted in Figure 5 where

- *bsender** and *breceiver** are location-aware "extensions" of *bsender* and *breceiver*, respectively. These extensions provide a new input location variable l that is assigned to every action and local channel.
- *Move_to* is a distribution connector with two roles – *dest_provider* and *subject_of_move* – and a glue defining the movement of *subject_of_move* to the location provided by *dest_provider*. In the example, both roles are played by *breceiver**.
- *Fixed* is a distribution connector with one role – *subject* – and a glue defining that subject is a non-mobile component.

The CommUnity model of this architecture is given by the configuration diagram represented in Figure 6. This configuration establishes that the location of *bsender** is controlled by the design fixed which has no actions to change it; hence, as required, once the value of this location is set at configuration time, it remains unchanged. On the other hand, it defines the synchronisation of the action *new-w* of *breceiver** with action *move* of *move_to*; this enforces, as required, the migration of *bsender** to the location defined by the word it has just received, if the word defines a location.

The main difference between this architectural model and the one illustrated in the previous section is in the explicit representation of the distribution and mobility aspects through the two distribution connectors *Fixed* and *Move_to*; in the previous section, these aspects were embedded in the extensions of the components.

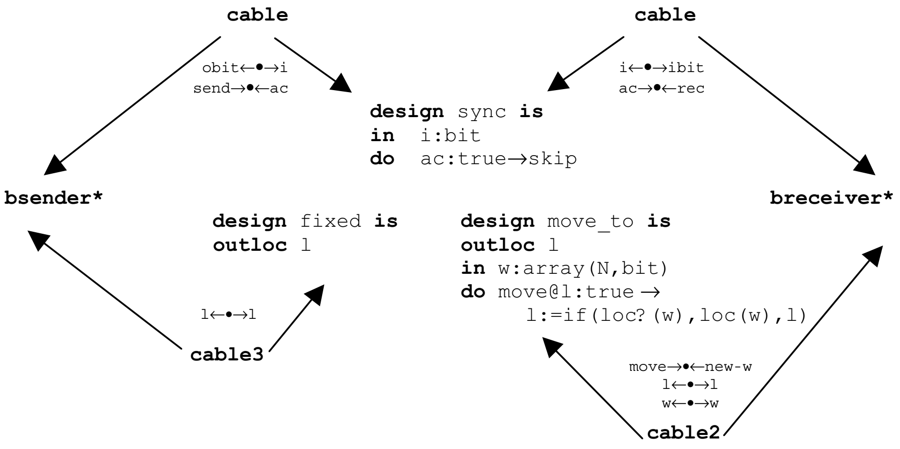


Fig. 6.

Distribution connectors can be developed in the same way as coordination connectors; the only difference is one of purpose, the former handling the way locations are interconnected and updated, the latter interconnecting communication channels and synchronising actions. Hence, in the rest of the paper, we concentrate on the ingredients that support incremental development as illustrated: the way "canonical" embeddings can make components and connectors location-aware.

We start by pointing out that every location-unaware design P defines implicitly a canonical location-transparent one – $\mathbf{mob}(P)$ – that has no location variables apart from λ which is where every channel and action is located. This is the minimal possible embedding in that it makes no decisions related to the distribution aspects of the system. Indeed, the very purpose of distinguishing the special location variable λ invariantly located at the special position $\perp$, is to allow for commitments wrt location-awareness to be postponed. In other words, this lifting preserves the location-transparency of the original design.

This lifting with location-awareness extends to morphisms: every morphism $\sigma : P_1 \to P_2$ in **DSGN** gives rise to a morphism $mob(\sigma) : \mathbf{mob}(P_1) \to \mathbf{mob}(P_2)$ in **MDSG** in a unique way because, as explained in the previous section, morphisms have to preserve the designated location λ. These two mappings on designs and morphisms define what in Category Theory is called an embedding $\mathbf{mob} : \mathbf{DSGN} \to \mathbf{MDSG}$, a functor that is injective on morphisms.

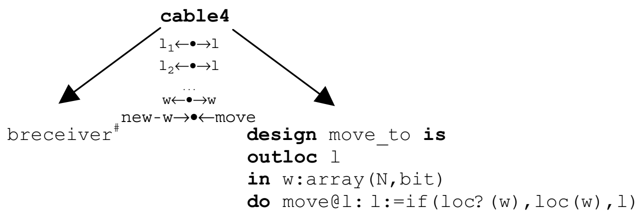
The functor **mob** automatically carries out the first step in the process of making a system architecture *sys* location aware: it lifts *sys* – a categorical diagram in **DSGN** – to its location-transparent version $\mathbf{mob}(sys)$ – a categorical diagram in **MDSG**. The second step consists in the extension of the components of *sys* in order to make them location-aware in ways that are already geared towards specific policies.

The options here can be many. For instance, we may group all actions and channels on the same location as in the case of our running example: *bsender* and *breceiver* were extended to *bsender** and *breceiver** with one input location to which every action and local channel was assigned. This form of extension implies that the component can only be moved as a whole; in other words, it forces the unit of mobility in the system to coincide with the unit of computation.

At the other extreme, we can extend $\mathbf{mob}(P)$ with a different input location for each action and local channel. In this way, we can give means for them to be controlled independently. In the case of our running example, this would lead to the following designs:

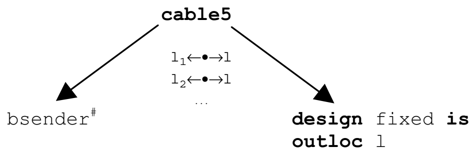
```
design bsender# is
inloc l1,l2,l3,l4,l5,l6,l7
out   obit@l1:bit
prv   word@l2:array(N,bit), k@l3:nat, rd@l4:bool
do    new-w@l5[ word,k] : k=N→ k'=0
[]    new-b@l6: ¬rd∧k<N→ rd:=true‖obit:=word[ k] ‖k:=k+1
[]    send@l7: rd → rd:=false

design breceiver# is
inloc l1,l2,l3,l4,l5,l6,l7
in    ibit:bit
out   w@l1:array(N,bit), k@l2:nat
prv   rd@l3:bool, word@l4:array(N,bit)
do    rec@l5: k<N→ word[ k] :=ibit‖k:=k+1‖rd:=false
[] prv prod@l6: ¬rd∧k=N→ rd:=true‖w:=word
[]    new-w@l7: rd∧k=N→ rd:=false‖k:=0
```

The pattern of mobility that we have to impose on *breceiver* in order to fulfill the system requirements stated in Section 2 can now be achieved through a more sophisticated cable that co-locates all the channels and actions as depicted in Figure 7. In a similar way, in order to define that the *bsender* is a fixed component, we just have to add the interconnection depicted in Figure 8.



```
design move_to is
outloc l
in w:array(N,bit)
do move@l: l:=if(loc?(w),loc(w),l)
```

Fig. 7.

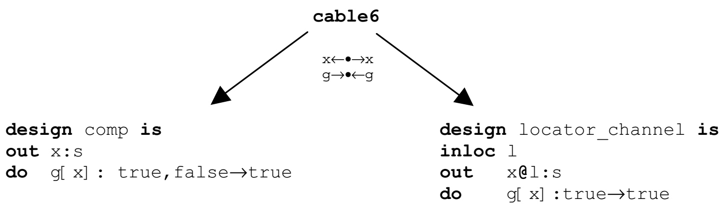


```
design fixed is
outloc l
```

Fig. 8.

In summary, there can be many ways in which individual designs can be made location-aware. What is important is that each extension P' : **MDSG** of a design P : **DSGN** comes together with an inclusion morphism *inc* : $\mathbf{mob}(P) \rightarrow P'$. These morphisms can be used to extend any configuration diagram *sys* over **DSGN** by replacing any interconnection morphisms $C \rightarrow P$ by the composition $\mathbf{mob}(C) \rightarrow \mathbf{mob}(P) \rightarrow P'$.

Notice that these different forms of extension of a location-transparent design $\mathbf{mob}(P)$ can be achieved through the superposition of what we can call *location connectors* whose purpose is to locate individual channels or actions. For instance, an output channel can be located through the location connector described in Figure 9. We only have to instantiate the role *comp* of the connector with the specific design by identifying the output channel that one wants to locate and identify the actions that operate in this channel.



```
design comp is
out x:s
do  g[ x] : true,false→true
```

```
design locator_channel is
inloc l
out   x@l:s
do    g[ x] :true→true
```

Fig. 9.

The extensions that we have used for the sender and the receiver can be obtained through the superposition of one or more such connectors. These *location connectors* are standard solutions that can be used across different components. From a methodological point of view, they should be made available, together with distribution connectors, in libraries that assist designers in the process of making a system architectural model location-aware. We are now working on extending the algebra of connectors that we started defining in [14] for systematising this process of publishing, finding, and reusing connectors.

5 Concluding Remarks

In this paper, we have addressed the integration of mobility in the set of aspects that systems architectures should be able to deal with. Our proposal supports an incremental and compositional approach based on a strict separation of concerns between computation, coordination and distribution. We showed how a new class of connectors can be defined that enforces patterns and policies related to the locations in which components perform computations and the network topology that supports coordination. Such distribution connectors can be superposed over location-transparent models of components and connectors as a means of addressing the mobility-based aspects that reflect the properties of the operational and communication infrastructure. In our approach, this process of tailoring the architecture of a system to specific deployment contexts can be achieved without having to redesign the other two dimensions – computation and coordination. The idea of having mobility explicitly represented in architectural models, which in this paper was explored in order to support incremental design, has other advantages. It allows taking advantage of new technologies or computational solutions without requiring the other components of the system to be changed or the global configuration of the system to be updated. The explicit representation of mobility aspects also makes it easier to evolve systems at run time because they can be evolved independently of the components or the interactions that they affect. Changes occurring in the communication infrastructure or in the topology of the network may require the adoption of different mobile-based paradigms, which can be obtained simply through the substitution of the distribution connectors in place in the system.

Most of the existing approaches to architecture refinement focus on the formalisation of refinement across different architectural views in different languages (the so called *vertical refinement*) and in particular transformations of architectural models into implementation code, e.g.,[10,5,1]. In the last case, refinement involves the addition of details on how computations and interaction protocols may be realised with specific programming constructs. However, to the best of our knowledge, architecture models and architectural styles based on mobility paradigms have never been addressed in this context.

In contrast, several approaches to the formalisation of mobile architectures and architectural styles based on components and architectural connectors can be found in the literature (e.g., [3,12,4]). In all these approaches, the mobility dimension is not taken as a separate and first-class concern. As far as the objectives are concerned, the approach most closely related to ours is [9]. It proposes an architecture-based approach to the modelling and implementation infrastructure of code mobility by exploring C2's connectors and message passing. In this work, groups of components and connectors are governed by an *Admin* Component and a *TopBorder* Connector. These special components and connectors are responsible for the mobility of components from one group to another, including the disconnection of a migrating component in the origin group and its reconnection in the destination group (the links that have to be established there are sent together with the component). In this way, the approach does not support the decoupling of computation, interaction, mobility and dynamic reconfiguration that we achieved in our framework.

Acknowledgements

This work was partially supported through the IST-2001-32747 Project AGILE – Architectures for Mobility. We would like to thank our colleagues in this project for much useful feedback, in particular Luís Andrade and Michel Wermelinger.

References

[1] Aldrich, J., C. Chambers and D. Notkin, "ArchJava: Connecting Software Architecture to Implementation", Proc. of the International Conference on Software Engineering, ACM Press, 2002.

[2] Allen, R., and D. Garlan, "A Formal Basis for Architectural Connectors", *ACM TOSEM*, **6**(3), 213–249, 1997.

[3] Ciancarini, P., and C.Mascolo, "A Catalog of Architectural Styles for Mobility", UBLCS Technical Report Series, Department of Computer Science, Univ. Bologna, 98-2, 1998.

[4] Cortellessa, V., and V. Grassi, "A Performance Based Methodology to Early Evaluate the Effectiveness of Mobile Software Architectures", Workshop progetto SALADIN: Software Architectures and Languages to Coordinate Distributed and Mobile Components Venezia, Italy, Feb. 14-16, 2001.

[5] Egyed, A., and N. Medvidovic, "Consistent Architectural Refinement and Evolution using UML", Proc. of the 1st Workshop on Describing Software Architecture with UML, Toronto, Canada, 83-87, 2001.

[6] Fiadeiro, J.L., A. Lopes and M. Wermelinger, "A Mathematical Semantics for Architectural Connectors", *Generic Programming*, R.Backhouse and J.Gibbons (eds), LNCS 2793, 190–234, Springer-Verlag, 2003.

[7] Lopes, A., J.L. Fiadeiro and M. Wermelinger, "Architectural Primitives for Distribution and Mobility", Proc. of SIGSOFT 2002/FSE-10, 41–50, ACM Press, 2002.

[8] Lopes, A., M. Wermelinger and J.L. Fiadeiro, "Higher-order Architectural Connectors", *ACM TOSEM*, in print.

[9] Medvidovic, N., and M. Rakic, "Exploiting Software Architecture Implementation Infrastructure in Facilitating Component Mobility", Proc. of Software Engineering and Mobility Workshop, Toronto, Canada, May 2001.

[10] Moriconi, M., X. Qian and R.A. Riemenschneider, "Correct Architecture Refinement", *IEEE Transactions on Software Engineering* **21**(4), 356–372, 1995.

[11] Picco, G.,"Mobile Agents: An Introduction",*Journal of Microprocessors and Microsystems* **25**(2),65–74, 2001.

[12] Picco, G., G.-C. Roman and P. McCann, "Expressing Code Mobility in Mobile Unity", Proc. of 6th ESEC, M.Jazayeri and H.Schauer (eds), LNCS 1301, 500–518, Springer-Verlag, 1998.

[13] Roman, G.-C., A.L. Murphy and G.P. Picco, "Coordination and Mobility", *Coordination of Internet designs: Models, Techniques, and Applications*, A.Omicini et al (eds), 253–273, Springer-Verlag, 2001.

[14] Wermelinger, M., and J.L. Fiadeiro,"Towards an Algebra of Architectural Connectors: a Case Study on Synchronization for Mobility", Proc. of ESEC/FSE'99, LNCS 1687, 393–409, Springer-Verlag, 1999.

[15] J.White, "Mobile Agents", *Software Agents*, J. Bradshaw (Ed.), 437–472, AAAI Press, 1997.