Architectural Primitives for Distribution and Mobility

António Lopes[1]
[1]Dept. Informatics
FCUL, Campo Grande
1749-016 Lisboa, Portugal
+351-217500604
mal@di.fc.ul.pt

José Luiz Fiadeiro[1,2]
[2]ATX Software SA
Al. António Sérgio 7, 1-C
2795-023 Linda-a-Velha, Portugal
+351-210120500
jose@fiadeiro.org

Michel Wermelinger[2,3]
[3]Dep. Informática
Univ.Nova Lisboa,
2829-516 Caparica, Portugal
+351-212948536
mw@di.fct.unl.pt

ABSTRACT
In this paper, we address the integration of a distribution dimension in an architectural approach to system development and evolution based on the separation between coordination and computation. This third dimension allows us to separate key concerns raised by mobility, thus contributing to our ability to handle the complexity that is inherent to systems required to operate in "Internet time and space".

Categories and Subject Descriptors
D.2.11 [**Software Engineering**]: Sofware Architectures – *Languages (description, interconnection, definition)* D.3.2 [**Programming Languages**]: Language Classifications – *Concurrent, distributed, and parallel languages.*

General Terms
Design, Languages, Theory.

Keywords
Software Architectures, Coordination, Mobility.

1. INTRODUCTION
Architecture-based approaches have been promoted as a means of controlling the complexity of system construction and evolution. Recently, a number of approaches show that, by adopting architectural principles based on the separation between computation and coordination, we can achieve higher levels of agility in systems that are required to operate in environments that are "business time critical", which includes those that make use of Web Services, B2B, P2P, or otherwise operate in what is known as "internet-time" [3]. Indeed, on the one hand, the clean separation that can be achieved between individual software components performing local computations that ensure basic services, and the connectors through which they are interconnected to make global properties of the system emerge from their interaction, allows systems to be evolved through the addition, deletion or substitution of connectors without interfering with the computations that are performed locally [17]. On the other hand, the same separation of concerns allows for components to be replaced, upgraded, monitored or regulated without interfering with the "logic" that dictates how their behaviour needs to be coordinated.

Recently, mobility has become an additional factor of complexity (see [20],[21] for surveys of this topic). In a mobile computing system, components may move across a network of locations, thus changing the environment in which computations need to be performed, which may require their adaptation or replacement. On the other hand, the properties of the network itself may change, which can make the connectors in place ineffective and require them to be replaced with ones that are compatible with the new topology of distribution, or give the opportunity for new coordination mechanisms to be introduced in order to optimise performance. That is to say, besides "time", "space" is an additional factor of complexity that needs to be addressed.

Our aim in this paper is to present the strategy that we are taking in order to add a "space"-dimension to our architectural approach. More precisely, our aim is to address distribution and mobility in the context of the separation between computation and coordination that we already mentioned, and provide semantic principles that support the externalisation of the mechanisms that are responsible for managing the distribution topology of systems. In particular, we aim to provide the definition and semantics of a primitive – distribution connector – that can fulfill a role similar to the one played by coordination connectors in externalising interconnections between components.

Having this goal in mind, the paper develops an extension to a program design language – CommUnity – that we proposed as a framework in which the principles underlying our architectural approach can be formalised and illustrated [7]. In section 2, we relate our approach to other work that can be found in the literature in order to further motivate and make clear our standpoint in what concerns "mobility". In section 3, we review the computational aspects of CommUnity and the model of interaction on which coordination mechanisms can be established. In section 4, we present the extensions that we propose for handling distribution and mobility. Finally, in section 5, we discuss the avenues that we are planning to follow to further develop our model in the scope of AGILE, a FET/IST funded project on "Architecture for Mobility".

2. RELATED WORK
Several different perspectives on mobile computing have been explored, giving rise to different ways of modelling distribution and mobility. A large number of models are process calculi, e.g. the Ambient Calculus [5] and different extensions of π-calculus [2][12][15]. There are also various

SIGSOFT 2002/FSE-10, Nov. 18-22, 2002, Charleston, SC, USA.

proposals of coordination languages and models that address distribution and mobility, for instance LLinda [16], Lime [19] and Mobis [14].

In this section we will focus our attention on formalisms that entail an explicit notion of space and adopt the same computational model as CommUnity. That is to say, formalisms where the functionalities provided by a component are described in terms of guarded multiple assignments. To the best of our knowledge, the only formalisms in this situation are Mobile Unity and Topological Actions Systems.

Although there is a large number of similarities between CommUnity and these two formalisms, even in the extensions we shall propose for CommUnity, they have perspectives on mobility that are very different from our own. Neither of them constitute a solution that addresses distribution and mobility in architectures that are structured according to the coordinating connectors.

Mobile Unity

This is an extension of Unity proposed in [21] that adopts programs as units of mobility. In Mobile Unity each program and its variables are co-located and move as a single unit. A program's location is represented by a special variable that can be effected as the other variables, possibly causing its movement. In contrast with Unity, in Mobile Unity the interactions between programs are described explicitly, decoupled from the description of the computational aspects. These interactions however are not used to describe purely the logical interactions between components (as in architectural connectors): they are rather a model of physical reality or a specification for services to be provided by the operating system or the middleware. For instance, it can be specified that the interactions between two components are the sharing of some variables limited, in a certain way, by the distance between them (transient sharing). This is, for instance, the way that wireless communication is modelled.

In the past, we have also carried through a similar extension to CommUnity [23]. However, this experience has shown that adding locations to programs in an ad-hoc way does not provide structural solutions to the main problem, which is the interference between computation, coordination and distribution.

Topological Actions Systems (TAS)

These are an extension of Action Systems [4][18] that address fine-grained mobility explicitly. Variables and actions (guarded commands) are regarded as resources, respectively data and code resources, and adopted as units of mobility. The space of mobility is assumed to be discrete (a set of positions) and its properties are modelled explicitly through a relation *cell* on positions s.t. *cell(p)* establishes the set of positions accessible from *p*. The execution of an action is then considered to be possible only if all the data resources accessed by the action are accessible from its location. In TAS, the interactions between the components of a system, that are achieved through sharing of memory, are described implicitly, relying on the use of the same names in different components.

This combination of implicit interactions with distributed resources gives rise to an approach oriented to context dependent services/resources. If an action *a* accesses a variable named *v*, it is necessary to make sure that *v* is available, i.e., there is at least a variable named *v* located at a position in the cell of *a*.

3. ARCHITECTURES IN COMMUNITY

CommUnity, introduced in [9], is a prototype parallel program design language that is similar to Unity [6] in its computational model but adopts a different coordination model. More concretely, whereas, in Unity, the interaction between a program and its environment relies on the sharing of memory, CommUnity relies on the sharing (synchronisation) of actions and exchange of data through input and output channels. Furthermore, CommUnity requires interactions between components to be made explicit whereas, in Unity, these are defined implicitly by relying on the use of the same variables names in different programs. As a consequence, CommUnity takes to an extreme the separation between "computation" and "coordination" in the sense that the definition of the individual components of a system is completely separated from the interconnections through which these components interact, making it an ideal vehicle for illustrating and formalising the approach that we wish to put forward.

3.1 The computational model

CommUnity is independent of the actual data types that can be used for modelling the exchange of data and, hence, we take them in the general form of a first-order algebraic specification. We assume a data signature $<S,\Omega>$, 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.

A CommUnity design is of the following form.

```
design  P is
out     O
in      I
prv     V
do      □ g∈sh(Γ)        g[D(g)]: L(g), U(g) → R(g)
        □ g∈prv(Γ)  prv  g[D(g)]: L(g), U(g) → R(g)
```

I and *O* are the sets of input and output channels of design *P*, respectively, and *V* is the set of channels that model internal communication. Input channels are used for reading data from the environment of the component. The component has no control on the values that are made available in such channels. Moreover, reading a value from an input channel does not "consume" it: the value remains available until the environment decides to replace it.

Output and private channels are controlled locally by the component, i.e. the values that, at any given moment, are available on these channels cannot be modified by the environment. Output channels allow the environment to read data produced by the component. Private channels support internal activity that does not involve the environment in any way. We use X to denote the union $I\cup O\cup V$ and *local(X)* to denote the union $V\cup O$ of *local* channels. Each channel v is typed with a sort $sort(v)\in S$.

Γ is the set of *action names*. The named actions can be declared either as *private* or *shared*. Private actions represent internal computations in the sense that their execution is uniquely

under the control of the component. Shared actions represent possible interactions between the component and the environment, meaning that their execution is also under the control of the environment. The significance of naming actions will become obvious below; the idea is to provide points of *rendez-vous* at which components can synchronise.

For every action name *g*:

- *D(g)* is a subset of *local(X)* consisting of the local channels into which executions of the action can place values. This is what is sometimes called the *write frame* of *g*. For simplicity, we will omit the explicit reference to the write frame when *R(g)* is a conditional multiple assignment (see below), in which case *D(g)* can be inferred from the assignments. 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 *frame* of *g*, i.e., the channels that are in *D(g)* or used in *L(g)*, *U(g)* or *R(g)*.
- *L(g)* and *U(g)* are two conditions such that $U(g) \supset L(g)$. These conditions establish an interval in which the enabling condition of any guarded command that implements *g* must lie. The condition *L(g)* is a lower bound in the sense that it is implied by the enabling condition. Therefore, its negation establishes a *blocking* condition. On the other hand, *U(g)* is an upper bound in the sense that it implies the enabling condition, therefore establishing a *progress* condition. Hence, 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 *X* 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 names account for references to the values that the channels take after the execution of the action. When *R(g)* is such that the primed version of each local channel in the write frame of *g* is 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*.

Channels and action names of a design *P*, together with the function that establishes the write frame of each action, constitute the *signature* of *P*, designated by *sig(P)*.

We present below an example of a CommUnity design.

```
design user is
out     p:ps+pdf
prv     s:0..2, w:Lowtex
do      work[w,s]: s=0,false → s'=1
[]      pr_ps: s=1,false → p:=ps(w) ‖ s:=2
[]      pr_pdf: s=1,false → p:=pdf(w) ‖ s:=2
[]      print: s=2 → s:=0
```

This design models a user that produces files in "Lowtex" format that it makes available, internally, in the private channel *w*. It can then convert them either to postscript or pdf formats, after which it makes them available (for printing) in the output channel *p*.

CommUnity supports higher-level component design. See for example the design *sender[t]* of a typical sender of messages. In this design, we are primarily concerned with the interaction between the sender and its environment, ignoring details of internal computations such as the production of messages. We only require that the component *sender* cannot produce another message before the previous one has been processed. After producing a message, the sender expects an acknowledgment to produce a new message. Such acknowledgment is modelled through the execution of action *send*.

```
design sender[t] is
out     o:t
prv     rd:bool
do      prod[o,rd]: ¬rd,false → rd'
[]      send[rd]: rd,false → ¬rd'
```

When, for every $g \in \Gamma$, *L(g)* and *U(g)* coincide, and the relation *R(g)* defines a conditional multiple assignment, the design is called a *program*. Notice that a program with a non-empty set of input channels is *open* in the sense that its execution is only meaningful in a configuration in which these inputs have been instantiated with output channels of other components. The behaviour of a closed program is as follows. At each execution step, one of the actions whose enabling condition holds of the current state is selected, and its assignments are executed atomically as a transaction. Furthermore, it is guaranteed that private actions that are infinitely often enabled are selected infinitely often (see [13] for a model-theoretic semantics of CommUnity).

As an example of a program, consider the following design.

```
design printer is
in      rf:ps+pdf
prv     busy:bool, pf:ps+pdf
do      rec: ¬busy→pf:=rf ‖ busy:=true
[] prv  prt:busy→busy:=false
```

This program models a printer that makes the files it downloads from the input channel *rf* available internally, after which it prints them. Weak fairness in the execution of private actions ensures that if there is a file to print, eventually it is printed.

3.2 The Coordination Model

As mentioned before, the model of interaction adopted in CommUnity is based on action synchronisation and the interconnection of input channels of a component with output channels of other components. These are standard means of interconnecting software components. What distinguishes CommUnity from other parallel program design languages that adopt this discipline is the fact that such interactions between components have to be made explicit and external to the components by providing the corresponding name bindings in well identified connectors: names are local to designs, the use of the same name in different designs is treated as being purely accidental.

This externalisation of interactions is well supported by a mathematical semantics based on Goguen's categorical view of General Systems Theory [11]. According to that view, interconnections are expressed via the morphisms of a category of designs:

A morphism of designs $\sigma: P_1 \to P_2$ *consists of a total function* $\sigma_x: X_1 \to X_2$ *and a partial mapping* $\sigma_{ac}: \Gamma_2 \to \Gamma_1$ *s.t.:*

1. *for every* $x \in X_1$, $i \in I_1$, $o \in O_1$, $v \in V_1$

$sort_2(\sigma_x(x)) = sort_1(x)$

$\sigma_x(o) \in O_2$

$\sigma_x(i) \in O_2 \cup I_2$

$\sigma_x(v) \in V_2$

2. *for every* $g \in \Gamma_2$ *s.t.* $\sigma_{ac}(g)$ *is defined*

$if\ g \in sh(\Gamma_2)\ then\ \sigma_{ac}(g) \in sh(\Gamma_1)$

$if\ g \in prv(\Gamma_2)\ then\ \sigma_{ac}(g) \in prv(\Gamma_1)$

3. *for every* $g \in \Gamma_2$ *s.t.* $\sigma_{ac}(g)$ *is defined*

$\sigma_x(D_1(\sigma_{ac}(g))) \subseteq D_2(g)$

$\sigma_{ac}(D_2(\sigma_x(x))) \subseteq D_1(x)$ *for every* $x \in local(X_1)$

$\Phi \models (R_2(g) \supset \sigma(R_1(\sigma_{ac}(g))))$

$\Phi \models (L_2(g) \supset \sigma(L_1(\sigma_{ac}(g))))$

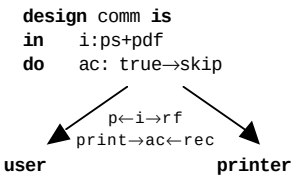
$\Phi \models (U_2(g) \supset \sigma(U_1(\sigma_{ac}(g))))$

A pair $<\sigma_x, \sigma_{ac}>$ *that satisfies 1 and 2 is called a signature morphism.*

This notion of morphism captures what in the literature on parallel program design is known as superposition [6],[10]. A morphism $\sigma: P_1 \rightarrow P_2$ identifies a way in which P_1 is "augmented" to become P_2 so that it can be considered as having been obtained from P_1 through the superposition of additional behaviour, namely the interconnection of one or more components. In other words, σ identifies P_1 as a component of P_2.

The map σ_x identifies for every channel of the component the corresponding channel of the system. The first group of constraints also establish that sorts of channels have to be preserved. Notice, however, that input channels of a component may become output channels of the system. This is because the result of interconnecting an input channel of a component with an output channel of another component in the system is an output channel of the system. Mechanisms for hiding communication, i.e. making it private, can be applied, but they are not the default in a configuration. In our opinion, hiding communication should result from an explicit design decision because it limits the ability of the system to accommodate new interactions.

The partial mapping σ_{ac} identifies the action of the component that is involved in each action of the system, if ever. The second group of constraints states that the type of actions is preserved. The last group of conditions on actions requires that change within a component is completely encapsulated in the structure of actions defined for the component and that the computations performed by the system reflect the interconnections established between its components. The two conditions on write frames imply that actions of the system in which a component is not involved cannot have local channels of the component in their write frame. The third condition reflects the fact that the effects of the actions of the components can only be preserved or made more deterministic in the system. The last two conditions allow the bounds that the design specifies for the enabling of the action to be strengthened but not weakened. Strengthening of these bounds 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.



In order to illustrate the way morphisms can be used for establishing interconnections, consider the diagram above. It defines a configuration in which files from a *user* component are sent to a *printer* component through synchronous communication. This form of communication is achieved by using a third design *comm* that, essentially, consists of an input channel to model the medium through which data is to be transmitted between the user and the printer (i/o interconnection of *p* and *rf*), and a shared action for the two components to synchronise in order to transmit the data (synchronisation of *print* and *rec*).

The connecting design – *comm* – just provides the required name bindings and has no computational contents. In fact, as we have shown in [7], in the configuration diagram of a system, only the signatures of the corresponding components need to be involved. CommUnity is what we have called a *coordinated formalism*, because it provides a complete separation between the computational and the coordination aspects of systems. Hence, in the rest of the paper configuration diagrams are defined at the level of signatures.

The semantics of configurations is given by a categorical construction: the colimit of the underlying diagrams. Taking the colimit of a diagram collapses the configuration into an object by internalising all the interconnections, thus delivering a design for the system as a whole. Colimits in CommUnity capture a generalised notion of parallel composition in which the designer makes explicit what interconnections are used between components:

- channels involved in each i/o-communication established by the configuration are amalgamated;
- every set $\{g_1,\ldots,g_n\}$ of actions that are synchronised through the interconnections is represented by a single action $g_1 \| \ldots \| g_n$ whose occurrence captures the joint execution of the actions in the set.
- the transformations performed by the joint action are specified by the conjunction of the specifications of the local effects of each of the synchronised actions, and the bounds on the enabling condition of joint actions are also obtained through the conjunction of the bounds specified by the components.

For instance, the colimit of the previous diagram collapses the configuration into the following design.

```
design  user-printer is
out     p:ps+pdf
prv     s:0..2, w:Lowtex, busy:bool, pf:ps+pdf
do      work[w,s]: s=0,false → s'=1
[]      pr_ps: s=1,false → p:=ps(w) ‖ s:=2
[]      pr_pdf: s=1,false → p:=pdf(w) ‖ s:=2
[]      print|rec: s=2∧¬busy → s:=0 ‖ busy:=true ‖ pf:=p
[] prv  prt: busy → busy:=false
```

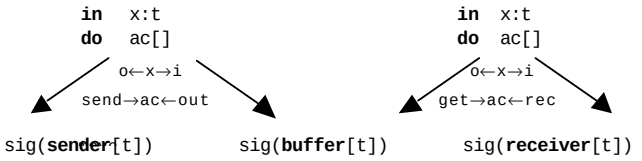
When the colimit of a configuration returns a closed program, it can also be used for providing an operational semantics for the system thus configured. The colimit can be seen as an abstraction of the actual distributed execution that is obtained by coordinating local executions according to the interconnections, rather than the program that is going to be executed as a monolithic unit. This point of view is specially important because it gives clues for the extension of CommUnity to location-aware systems.

As we have already mentioned, at each execution step one of the actions is chosen and executed if its guard is true, with the guarantee that private actions that are infinitely often enabled are selected infinitely often. Because actions of the system are synchronisation sets of actions of its components, the evaluation of the enabling condition of the chosen action can be performed in a distributed way by evaluating the enabling conditions of the component actions in the synchronisation set. According to the semantics that we have just given, the joint action will be executed iff all the local enabling conditions evaluate to *true*. The execution of the multiple assignment associated with the joint action can also be performed in a distributed way by executing each of the local assignments. What is important is that the atomicity of the execution is guaranteed, i.e. the next system step should only start when all local executions have completed, and the i/o-communications should be implemented so that every local input channel is instantiated with the correct value – that which holds of the local state before any execution starts (synchronicity). Finally notice that because, so far, we have been considering solely location-transparent systems, this distributed execution coincides with the execution of the program returned by the colimit as a monolithic unit.

We end this section by showing that using the mechanisms that we have just described for configuration design in CommUnity, it is possible to treat component interactions as first-class entities, namely by adopting architectural connectors [1] to support their design. According to [1], an architectural connector can be defined by a set of *roles*, that can be instantiated with specific components of the system under construction, and a *glue* specification that describes how the activities of the role instances are to be coordinated. For instance, asynchronous communication through a bounded channel can be represented by a connector *Async* with two roles – *sender* and *receiver* – and a glue that models a *buffer* with FIFO discipline. Their designs are given below.

```
design buffer[t]is                    design receiver[t] is
in     i:t                            in    i: t
out    o:t                            do    rec:true,false→skip
prv    rd: bool, b: list(t)
do     put:|b|<bound → b:=b.i
[] prv next:|b|>0∧¬rd →o:=head(b)‖b:=tail(b)‖rd:=true
[]     get:rd →rd:=false
```

The glue and roles of *async* are interconnected as follows.



The buffer prevents the *sender* from sending a new message when there is no space and prevents the *receiver* from reading a new message when there are no messages. The two roles – *sender* and *receiver* – define the behaviour required of the components to which the connector can be applied. For the *sender*, we require that no message be produced before the previous one has been processed. For the *receiver*, we simply require that it has an action that models the reception of a message.

The use of a connector in the construction of a particular system consists in the instantiation of its roles with specific designs. The instantiation of a role with a component is possible iff the component fulfils the requirements the role determines. Therefore, instantiation corresponds to a form of refinement that can also be captured by a notion of morphism (see [7] for details).

We will use *async(f:sender→A,g:receiver→B)* to denote the system that consists of components *A* and *B* communicating asynchronously, where *f* and *g* are signature morphisms that bind the actual parameters to the formal ones. In the case at hand, they identify, for instance, the output channel of *A* where the messages to be transmitted through the buffer are placed.

4. DISTRIBUTION AND MOBILITY

This architectural approach does not take into account the properties of the "physical" distribution topology of locations and communication links. It relies on the fact that

- the individual components can perform the computations that are required to ensure the functionalities specified for their services at the locations in which they are placed;
- the coordination mechanisms put in place through connectors can be made effective across the "wires" that link locations in the underlying communication network.

Hence, so far, we have been describing systems that are location-transparent or location-unaware. Our aim is to extend CommUnity in order to reflect distribution and mobility, namely we want to be able to describe systems with components that have a location-dependent behaviour and are able to move, and systems with location-dependent coordination patterns.

4.1 Syntax

We adopt an explicit representation of the space within which movement takes place, but we do not assume any specific notion of space. This is achieved by considering that "space" is constituted by the set of possible values of a special data type included in the fixed data type specification over which components are designed (see section 3.1). The corresponding data sort, named *loc*, models the positions of the space in a way that is considered to be adequate for the situation at hand. The only requirement that we make is a special location $-\perp-$ be distinguished: its role will be discussed further below. In this way, CommUnity can remain independent of any specific notion of space and, hence, be used for designing systems with different kinds of mobility. For instance, in physical mobility, the space is, typically, the surface of the earth, represented through a set of GPS coordinates. In some kinds of logical mobility, space is formed by IP addresses. Other notions of space can be modelled, namely multidimensional spaces, allowing us to accommodate richer perspectives on mobility such as the ones that result from combinations of logical and physical mobility, or logical mobility with security concerns.

In order to model systems that are location-aware, we make explicit how system "constituents" are mapped to the positions of the fixed space. Mobility is then associated to the change of positions. By constituents we mean output and private channels, actions, or any group of these. This means that the unit of mobility (the smallest constituent of a system that is allowed to move) is fine-grained and different from the unit of execution.

More concretely, CommUnity designs are extended with "locations", and each of their constituents is assigned a location. On the one hand, each output and private channel x of a design is associated with a location l. We make this assignment explicit by writing $x@l$. At every given state, the value of l indicates the position of the space where the values of x are made available. Each location l may assume different values at different times, making x a potentially mobile entity. On the other hand, each action name g is associated with a set $\Lambda(g)$ of locations. This means that the execution of action g is distributed over the locations in $\Lambda(g)$ in the sense that its execution involves the synchronous execution of a guarded command in each of these locations. Hence, guarded commands are not anymore associated with actions but with located actions, i.e. pairs $g@l$, for $l \in \Lambda(g)$.

Locations in a component design 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. Hence, if l is an input location, the movement of any constituent located at l is under the control of the environment. Output locations can only be modified locally but can be read by the environment. Hence, if l is an output location, the movement of any constituent located at l is under the control of the component.

As an example, consider the design *mobuser*, a location-aware version of the *user* presented before.

```
design mobuser is
inloc   l
out     p@l:ps+pdf
prv     s@l:0..2, w@l:Lowtex
do      work@l[w,s]: s=0,false → s'=1
[]      pr_ps@l: s=1,false → p:=ps(w) ‖ s:=2
[]      pr_pdf@l: s=1,false → p:=pdf(w) ‖ s:=2
[]      print@l: s=2 → s:=0
```

This design models a centralised system, in the sense that all its constituents are located at the same position, that does not control its own movement. This is captured by the fact that every action and local channel is assigned the same input location l. For instance, the value of l can be determined by a human that carries a laptop where Lowtex is running.

A different example is *followme*, the design of a system that partially controls its own movement but that is still centralised.

```
design  followme is
inloc   lf
outloc  l
do      action@l[]: l=lf→ skip
[] prv  move@l[l]: ¬(l=lf) → l'=lf
```

The system decides itself when to move but the choice of the destination is still made by the environment (through the input location *lf)*. The design also includes an action that is enabled for execution only when the component is in the location provided by the environment.

```
design  mobuser-fprinter is
inloc   lu
outloc  lp
out     p@lu:ps+pdf
prv     s@lu:0..2,w@lu:Lowtex,busy@lp:bool,pf@lp:ps+pdf
do      work[w,s]@lu: s=0,false → s'=1
[]      pr_ps@lu: s=1,false → p:=ps(w) ‖ s:=2
[]      pr_pdf@lu: s=1,false → p:=pdf(w) ‖ s:=2
[]      print|rec @lu: s=2 → s:=0
                  @lp: ¬busy → pf:=p ‖ busy:=true
[] prv  prt@lp: busy → busy:=false
```

Finally we present an example of a design distributed over two, potentially different, locations. The design *mobuser-fprinter* is also a location-aware version of a design presented before *–user-printer*. It models a system in which the mobile user communicates synchronously with a printer located potentially at a different location. Notice that, whereas the location of the mobile user is under the control of the environment, the location of the printer is locally controlled. More precisely, because none of the actions of *mobuser-fprinter* changes the values of *lp* ($D(lp)=\emptyset$) we may conclude that the location of the printer, once it is set at configuration time, will remain unchanged. That is to say, the printer is a non-mobile component of this system.

In order to keep the possibility of designing location-unaware systems, the set of the locations L of a design P includes by default a special element designated by λ_L. It is an output location that, however, cannot be modified by P. This is enforced by the condition $D(\lambda_L)=\emptyset$. By default, every action and channel is considered to be located at λ_L. Trivially, every standard CommUnity design defines a canonical distributed design: the one that has all its constituents located at λ_L.

In order not to overload designs, we omit $g@\lambda_L$ whenever it is an empty action, i.e. when

$$D(g@\lambda_L)=\emptyset \text{ and } L(g@\lambda_L)= U(g@\lambda_L)=R(g@\lambda_L)=true.$$

We will see that any action located uniquely at λ_L can access any required entity in a location-transparent manner.

4.2 Semantics

Before proceeding towards more complex examples that illustrate the expressive power of our approach, it is essential to identify what is the semantical counterpart of the proposed syntatic extension of CommUnity designs.

Let us analyse the *mobuser-printer* example a little further. Recalling the colimit semantics given in section 3.2, the joint action *print|rec* will be executed iff its two local guards evaluate to *true*. The distribution dimension added to *user-printer* in order to obtain *mobuser-printer* determines that the expression *s=2* is evaluated at the location given by *lu* and *¬busy* is evaluated at the location given by *lp*. The execution of the multiple assignment associated with *print|rec* is also performed in a distributed way by executing each of the local assignments. The next system step should only start when the two local executions have completed. Given that the locations variables *lu* and *lp* may have different values, the question of knowing whether this synchronisation is possible arises.

As explained before, in what concerns space, we have only assumed that it is constituted by the set of possible values of the data type *loc*. For the purpose of this example, we might have decided, for instance, to consider *loc* as being an interval in $nat \times nat$ modelling some contiguous space in a building. Notice, however, that this does not give us any information about the structure of the space (whether it is an open space or there are walls). It also does not express in which conditions the communication is viable. Those conditions typically depend on the communication medium. For instance, the conditions for a viable communication are not the same for infrared communication and radio links.

We consider that, once we fix an algebra $\mathcal{U}$ for the data types, namely a domain $\mathcal{U}_{loc}$ for *loc*, the relevant properties of the mobility space are captured by two binary relations over $\mathcal{U}_{loc}$:

- A relation *bt* s.t. $n\ bt\ m$ means that *n* and *m* are positions in the space "in touch" with each other. Coordination among components takes place only when they are "in touch" with each other.
- A relation *reach* s.t. $n\ reach\ m$ means that position *n* is reachable from *m*. Movement of a component to a new position is possible only when this position "is reachable" from the current one.

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 $\perp_{\mathcal{U}}$ and, furthermore, $\perp_{\mathcal{U}}\ bt\ m$, for every $m \in \mathcal{U}_{loc}$. Moreover, in our setting, it seems reasonable to require that *bt* be symmetric and reflexive, and that *reach* be reflexive. In this way, components can always communicate when they are co-located (at the same location/position). Notice that, when we say communication, we mean the primitive forms of communication between components in CommUnity – action synchronisation and i/o-communications through channels. Because we are considering that *bt* captures the conditions under which these two forms of communication are viable and action synchronisation is intrinsically symmetric, we have required that *bt* be symmetric. A more fined-grained model could be obtained by modelling separately, with two different relations, the conditions under which action synchronisation and i/o communication are possible.

To make things harder, the relations *bt* and *reach* may also vary over time. For instance, in the case of infrared communication, an obstacle may appear at any time between the communicating entities making the communication impossible.

The operational semantics for an extended (distributed) CommUnity program can only be given in terms of a infinite sequence of relations $(bt_i, reach_i)_{i \in N}$. At each execution step, one of the actions that can be executed is chosen and executed. The conditions under which a distributed action *g* can be executed at time *i* are the following, where $[e]^i$ denotes the value of the expression *e* at time *i*.

1. *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 their local actions and, hence, their locations have to be in touch

2. *for every* $l \in \Lambda(g)$, *g@l can be executed, i.e.,*

a. *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 read or written and, hence, *l* has to be in touch with the locations of these channels

b. *for every* $l_1 \in D(g@l) \cap L$ *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*, then every possible new value of l_1 must be a position reachable from the current one

c. *the local guard L(g@l), which is equal to U(g@l) in a program, evaluates to true*

In light of these conditions, it is interesting to analyse again some of the previous examples. For instance, in the example of *mobuser-fprinter*, after a file has been put in *p* for being printed, the communication of this file to the printer has to wait until the mobile user is located at a position that is in touch with the printer's. Hence, the job may be pending for ever. In the example of *followme*, it is interesting to notice that the action *move*, that moves the entities located at *l* to a new position given by *lf*, can only be executed if the current value of *l* is a position in touch with the value of *lf* and, furthermore, *lf* is reachable from *l*.

4.3 Coordination

The generalisation of the notion of morphism presented in section 3.2 to distributed designs is straightforward. Essentially, we have to extend signature morphisms with a mapping between the sets of locations and transpose the conditions over the actions to located actions. More precisely,

A morphism $\sigma: P_1 \to P_2$ *consists of a signature morphism* $\sigma: sig(P_1) \to sig(P_2)$ *and a total function* $\sigma_l: L_1 \to L_2$ *that preserves the pointed element (λ), satisfying:*

1. *for every* $l \in outloc(L_1)$, $\sigma_l(l) \in outloc(L_2)$
2. *for every* $x \in local(X_1)$ *and* $g \in \Gamma_2$ *s.t.* $\sigma_{ac}(g)$ *is defined,*

$$\sigma_l(\Lambda_1(x)) \subseteq \Lambda_2(\sigma_l(x))$$

$$\sigma_l(\Lambda_1(\sigma_{ac}(g))) \subseteq \Lambda_2(g)$$

3. *for every* $g \in \Gamma_2$ *s.t.* $\sigma_{ac}(g)$ *is def. and* $l \in \sigma_l^{-1}(\Lambda(g))$

$$\sigma_{var}(D_1(\sigma_{ac}(g)@l)) \subseteq D_2(g@\sigma_l(l))$$

$$\sigma_{ac}(D_2(\sigma_x(x)@\sigma_l(l))) \subseteq D_1(x@l) \text{ for } x \in local(X_1)$$

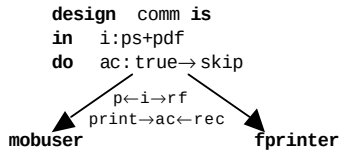
$$\Phi \models (R_2(g@\sigma_l(l)) \supset \sigma(R_1(\sigma_{ac}(g)@l)))$$

$$\Phi \models (L_2(g@\sigma_l(l)) \supset \sigma(L_1(\sigma_{ac}(g)@l))$$

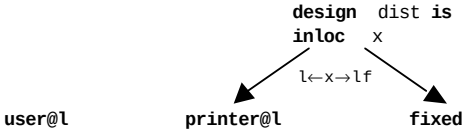
$$\Phi \models (U_2(g@\sigma_l(l)) \supset \sigma(U_1(\sigma_{ac}(g)@l))))$$

For instance, the diagram below, similar to the one presented in section 3.2, establishes the synchronous communication of files from the mobile user to the fixed printer. Notice that the design *comm* used in this diagram is the canonical distributed design defined by the standard one. Given that *comm* models the medium through which data is to be transmitted, it is obviously location-unaware. Using the analogy with hardware, *comm* together with the two morphisms act as an

interconnection "cable" between the two components, and cables are, in fact, location-unaware (wireless connections have this property as well).



Not surprinsingly, the design *mobuser-fprinter* already presented is a colimit of this diagram. More interesting is the fact that the distribution dimension added to this system – the fact that the user is a mobile entity that does not control its own movement whereas the printer is fixed – can be described separately from the two other dimensions: computation and coordination. The diagram above



where

- *fixed* is a design that consists of an output location named *lf*;
- *P@l* denotes the distributed design we obtain by adding an input location variable *l* to *P* and by locating every constituent of *P* in *l*;

specifies that the components are located in independent locations (recall that the use of *l* twice is treated as being purely accidental) and the user is mobile and does not control its movement (*l* is an input location). Moreover, the binding of location *l* of the printer with the location of *fixed* – a location with a constant value – defines that the printer is fixed in a certain position.

It is important to notice that interconnections between distributed designs can also be established at the level of their signatures, which were extended accordingly. Signatures are extended in order to include the locations of the component and of its channels and actions.

4.4 Externalisation of Distribution

As mentioned in the introduction, our aim is to provide an externalisation of the mechanisms that are responsible for managing the distribution topology of systems, in particular, to provide semantic primitives through which the distribution/mobility dimension can be explicitly represented in system architectures.

In this section, we will proceed with the use of CommUnity distributed designs to model location-aware systems but now focusing on the externalisation of their distribution dimension. The approach we will adopt is to describe separately what, in the definition of a system, is responsible for its computational aspects, what is concerned with coordinating the interaction between its different components, and what is concerned with the distribution of its constituents.

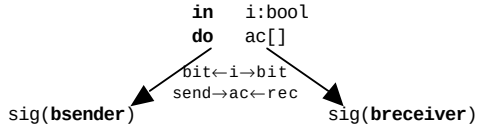
The first example, introduced in [21], consists of a system of two components – a sender and a receiver of bits. The sender exists at some fixed location in space and is neither aware of nor in control of its own location. It produces, in one go, words of bits that are then transmitted one by one. In contrast to the sender, the receiver is mobile and controls its own location. Upon receipt of a word which is a location, it may choose to move to that location. The communication between the two components is synchronous.

Computation

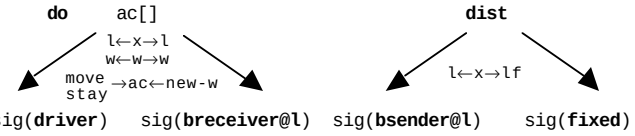
```
design  bsender is
out     bit:bool
prv     word:array(N,bool), k:nat, rd:bool
do      new-w[word,k]: k=N → k'=0
[]      new-b: ¬rd∧k<N→ rd:=true‖bit:=word[k]‖k:=k+1
[]      send: rd → rd:=false

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

Coordination



Distribution/Mobility



where

```
design  driver is
outloc  l
in      w@l:array(N,bool)
prv     b@l:bool
do      move@l: loc(w) → l:=location(w)
[]      stay@l: true → skip
```

We assume that *loc(_)* is an operation on bit arrays that indicates whether the corresponding word, given by *location(_)*, is a location.

In the interconnection of *bsender* with the *driver*, both actions *move* and *stay* are synchronised with the action *new-w* of *bsender*. This gives rise to a design with the action *rec* as defined in *breceiver* and the action *new-w* split in two:

```
      new-w|move@l: k=N∧loc(w) → k:=0‖l:=location(w)
[]    new-w|stay@l: k=N → k:=0
```

In this way, once the last bit of a word has been received, the receiver may chose to move or not to that location. The choice of moving is only available if the transmitted word is a location that is reachable from the current location.

In order to define the synchronous communication between the *bsender* and *breceiver* in our system, we opt to present explicitly a configuration diagram. An alternative would have been to define it implicitly by using the connector *sync[bool]*. This is possible because its roles – *sender and receiver* – are refined by *bsender* and *breceiver*, respectively.

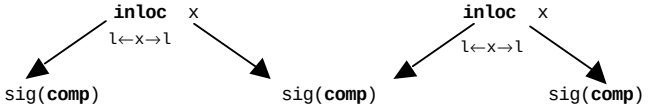
Given the structural similarity of the coordination and distribution descriptions – configuration diagrams involving the signatures of the components of the system – it is evident that it is also possible and desirable to give to the patterns of

distribution/mobility of components a first-class status and provide primitives for the description of these patterns. Our proposal is that these patterns be defined, like interactions, by a set of roles and a glue specification. Each role describes what is expected of each involved part, i.e., it determines the obligations that they have to fulfil to become instances of the roles. The glue describes the pattern of mobility and distribution the role instances experience. These primitives are called distribution connectors.

In this example, we can easily identify two patterns, both with one unique role. On the one hand, we have a pattern of non-mobility of a given component. This pattern can be captured by a distribution connector with role *comp*, a design consisting of an input location, which describes that the pattern is only applicable to components that do not control their own movement. The glue is the design *fixed*, which defines the non-mobility property.

On the other hand, we have the following pattern of mobility: whenever the value of the input variable *w* defines a possible location reachable from the current one, a non-deterministic choice exists between to move to that location or to remain still. This pattern can be captured by a distribution connector also with the role *comp* but with a different glue – the design *drive*.

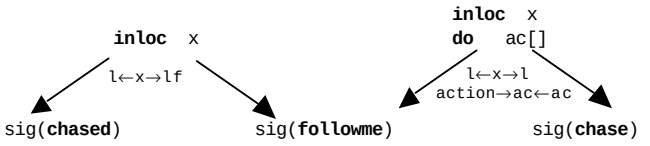
Another very simple pattern is the one that establishes that two components are always co-located. This pattern is captured by a distribution connector *co-loc* with two roles and a glue, all modelled by the design *comp* that consists simply on an input location *l*.



A more complex and interesting pattern is given by the distribution connector *fm*, defining a pattern of mobility involving two components that have to be instances of, respectively, *chased* and *chase*:

```
design  chased is          design  chase is
outloc  l                  inloc   l
                           do      ac@l:true,false → skip
```

The glue of *fm* is the design *followme* presented previously and is interconnected to the roles as follows.



In the mobility pattern thus defined, the instance of *chase* is moved to the location of the instance of *chased* whenever they are not co-located. Furthermore, in this situation *chase*'s action *ac* is prevented from occurring. We may use this pattern, for instance, to describe yet another variant of a sender-receiver system. This time, the sender and the receiver communicate asynchronously. The sender does not control its own location, whereas the receiver, whenever it is not co-located with the sender, interrupts the reception of messages and tries to move to the sender's position.

```
Computation
     sender, receiver
Coordination
     async(id:sender→sender,id:receiver→receiver)
Distribution/Mobility
     fm(incl:chased→sender@l, η_s:chase→receiver@l)
     co-loc(incl:comp→buffer@l,incl:comp→sender@l)
```

where *incl* denotes the inclusion and η_s identifies that action *ac* of *chase* is *rec* of *receiver*.

This example raises an interesting question: should the glues of coordination connectors be location-transparent? In the case at hand, it was specified that the glue of *async* is co-located with the sender.

In contrast with the coordination connector *sync*, the glue of *async* has a computational part, namely it mantains a buffer with pending messages, and hence should be a location-aware design. In situations in which the glue, although it is defined as a design, does not perform any computation, but simply provides a pure coordination function just like an ideal, neutral "cable", the glue will be typically location-transparent.

5. CONCLUSIONS

In this paper, we extended the architectural approach that we developed around the separation between computation and coordination with a third dimension: distribution. This extended approach will allow us to address the complexity of systems that are required to operate not only in environments that are "business-time" critical, i.e. are able to reconfigure themselves dynamically very quickly, but also "space" critical in the sense that they need to be able to address mobility of components across locations and react to changes in the communication networks.

For that purpose, we introduced a notion of "distribution connector" that, together with the more standard "coordination connectors" will enforce a strict separation of concerns that

- ensures that the configuration of the system as a whole can evolve in a compositional, non-intrusive way.
- means that all the mechanisms that are needed to ensure the levels of coordination required for global system properties to emerge from individual computations, even simple message calling, have to be superposed through explicit coordination connectors.
- makes sure that all the mechanisms that are needed to ensure the levels of distribution and mobility required for global system properties to emerge from the individual components, even simple permanent co-location, have to be superposed through explicit distribution connectors.

A mathematical semantics based on categories of CommUnity designs was proposed for this extension. We would now like to use this semantics to explore more complex topological properties of location spaces – the ones we used in the paper were very naive, just enough to illustrate the intended approach.

Other avenues that we intend to explore immediately are related to reconfiguration and verification. This is an effort that we are pursuing within a research consortium –AGILE – funded by the EU under the FET/IST Global Computing initiative, that also involves the Universities of Munich (LMU), Pisa and Florence, and the CNR-IEI of Pisa.

6. ACKNOWLEDGMENTS

This work is supported by EU under the consortium AGILE, IST-2001-32747 Global Computing Initiative and Fundação para a Ciência e Tecnologia and FEDER through project POSI/32717/00 (FAST-Formal Approaches to Software Architecture).

7. REFERENCE

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

[2] R.Amadio,"An asynchronous model of locality, failure, and process mobility", in D.Garlan and D.Métayer (eds), *Coordination'97:Coordination Languages and Models*, LNCS 1282, Springer-Verlag, 1997.

[3] L.Andrade and J.Fiadeiro, "Coordination Technologies for Managing Information System Evolution", in K.Dittrich, A.Geppert and M.Norrie (eds), CAiSE'01, LNCS 2068, 374-387, Springer-Verlag 2001.

[4] R.Back and R.Kurki-Suonio, "Distributed Co-operation with Action Systems", in *ACM Transactions on Programming Languages and Systems*, 10(4), 513-554, 1988.

[5] L.Cardelli and A.Gordon, "Mobile Ambients", in Nivat (ed), *FoSSACs'98*, LNCS 1378, 140-155, Springer-Verlag, 1998.

[6] K.Chandy and J.Misra, *Parallel Program Design - A Foundation*, Addison-Wesley 1988.

[7] J.L.Fiadeiro, A.Lopes and M.Wermelinger, "A Mathematical Semantics for Architectural Connectors", submitted, available at www.fiadeiro.org/jose/papers.

[8] J.L.Fiadeiro and A.Lopes, "Semantics of Architectural Connectors", in *TAPSOFT'97*, LNCS 1214, Springer-Verlag 1997, 505-519.

[9] J.L.Fiadeiro and T.Maibaum, "Categorical Semantics of Parallel Program Design", *Science of Computer Programming* 28, 1997, 111-138.

[10] N.Francez and I.Forman, *Interacting Processes*, Addison-Wesley 1996.

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

[12] M.Hennessy and J.Riely,"A typed language for distributed mobile processes", in *Proc. ACM Principles of Prog. Lang.* ACM, 1998.

[13] A.Lopes and J. L. Fiadeiro, "Using explicit state to describe architectures", in E. Astesiano (ed), *FASE'99*, LNCS 1577, 144–160, Springer-Verlag, 1999.

[14] C.Mascolo, "MobiS: A specification language for mobile systems", Coordination'99: Coordination L*anguages and Models*, LNCS 1594, 37–52, Springer-Verlag 1999.

[15] R.Nicola, G.L.Ferrari and R.Pugliese, "Klaim: a Kernel Language for Agents Interaction and Mobility", *IEEE Trans. on Software Engineering*, 24 (5), 315-330, 1998

[16] R.Nicola, G.L.Ferrari and R.Pugliese, "Coordinating Mobile Agents via Blackboards Acess Rights", in D.Garlan and D.Métayer (eds), *Coordination'97: Coordination Languages and Models*, LNCS 1282, 220-237, Springer-Verlag, 1997.

[17] P.Oreizy and R.N.Taylor, "On the Role of Software Architectures in Runtime System Reconfiguration", IEE Proceedings-Software, 145 (5), 137-145, 1998

[18] L.Petre, K.Sere and M.Waldén, "A Topological Approach to Distributed Computing", in *Proc. of WDS'99 Workshop on Distributed Systems*, Electronical Notes in Theoretical Computer Science, 8, Elsevier, 1999.

[19] G.P.Picco, A.L.Murphy and G.-C.Roman, "Lime: Linda meets Mobility", in Proceedings of the 21st International Conference on Software Engineering, May 1999, 368-377.

[20] G.-C.Roman, G.P.Picco, A.L.Murphy, "Software Engineering for Mobility: A Roadmap," in A. Finkelstein (ed), *Future of Software Engineering, 22nd International Conference on Software Engineering*, 241-258, June 2000

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

[22] G.-C.Roman, P.J.McCann and J.Y.Plun, "Mobile UNITY: reasoning and specification in mobile computing", *ACM TOSEM*, 6(3),250-282, 1997.

[23] M.Wermelinger and J.Fiadeiro, "Connectors for Mobile Programs", *IEEE Transactions on Software Engineering* 24(5), 331-341, 1998