Coordination for Distributed Business Systems

L.Andrade[1], J.L.Fiadeiro[2], A.Lopes[3] and M.Wermelinger[4]

[1]ATX Software SA
Alameda António Sérgio 7, 2795-023 Linda-a-Velha, PORTUGAL
landrade@atxsoftware.com

[2]Department of Mathematics and Computer Science, University of Leicester
University Road, Leicester LE1 7RH, UK
jose@fiadeiro.org

[3]Department of Informatics, Faculty of Sciences, University of Lisbon
Campo Grande, 1749-016 Lisboa, PORTUGAL
mal@di.fc.ul.pt

[4]Dep. of Informatics, Faculty of Sciences and Technology, New University of Lisbon
Quinta da Torre, 2829-516 Caparica, PORTUGAL
mw@di.fct.unl.pt

Abstract. We motivate, discuss and present extensions to architecture modelling techniques through which business systems can support services and applications that are location and network aware. These extensions provide a complete separation between three concerns: Computation, which accounts for the way service functionalities are provided in system components, Coordination, which accounts for the interactions that need to be put in place to enforce business rules, and Distribution, which accounts for the need to reconfigure the way services are provided in reaction to changes in the locations where functionalities are computed and the communication infrastructure over which coordination takes place.

1 Introduction

The advent of the Net and the promise of ever more stable wireless technologies have completely changed the way information systems are required to support business. For instance, the user friendliness of PDAs and the speed, relative low cost, and reliability of communication protocols (e.g. GPRS) is creating natural expectations on companies such as banks, which are now looking for ways of putting in place channels dedicated to B2C interactions for customers using handheld devices. It is clear that there will be an increasing demand for more sophisticated applications that can take advantage of the user friendliness of PDAs, e.g. for integrating services from different corporations such as consolidated account statements from different banks, management of personal payment plans, *inter alia*.

The challenges that the ability of being on-line anywhere and at any time is raising on software development are, therefore, immense. For a start, we have definitely shifted from a product to a service-based economy for which systems can no longer

be conceived in terms of static client-server relationships between components. Systems have to be prepared to support business interactions that occur over a network of distributed locations in which the different parties can be mobile. Mobility means, in particular, that business services need to be location-aware. For instance, the services that we can expect from our bank account will be different depending on whether we are sitting in front of our bank manager, using the Internet at home or the office, using an ATM on the street, or our PDA while sitting in a traffic jam. Mobility also means that systems need to be network-aware: the interaction with my account will have to be updated when I change from using the PDA in the car to my desktop computer when I arrive in my office.

The increased levels of complexity involved in the design, development and deployment of this new generation of systems, services and applications is no longer of an algorithmic nature but results, instead, from the need to account for intricate networks of interconnected components from which the global behaviour of the system emerges. Each individual component is, usually, sufficiently simple and self-contained in terms of its computational nature. It is the ability to interconnect these components dynamically, according the the properties of the distribution topology over which they are operating, that enables business systems to take advantage of the opportunities that are being offered by the new networking technologies. In other words, in the age of Ubiquitous Computing, the challenge is not to program "clever" algorithms but "clever" interactions. This is the challenge that we have proposed ourselves to address and on which we wish to report in this paper: to investigate techniques through which business systems can be made location and network aware.

2 Coordination and Architectures

The approach that we decided to follow comes from the simple realisation that we can only focus on and configure interactions once these are made first-class citizens, i.e. available explicitly in system models, not hidden in the code that programs the behaviour of individual components. Although this observation seems to be too obvious to deserve much attention, one just has to remember that object-oriented programming is a good example of a class of languages in which interactions are, precisely, buried in the code that implements the components through feature calls and other means of association [11]!

This ability to separate interactions from computations has been the subject of study, for the past 10 years, of a research area called "Coordination Languages and Models" [7]. Languages like Linda, Gamma and Manifold, to name just a few, have shown how this principle of separation can be made effective from a computational point of view and how it leads to more flexible composition mechanisms. These same principle of separation can be found at the core of Software Architectures [2]: it leads to the explicit representation, as first-class citizens, of architectural connectors as entities that can be dynamically superposed over computational components to coordinate the way they interact.

Our previous work [1] has shown how such an architectural approach ensures that interconnections can evolve to reflect changes in the business rules that determine how business entities should interact, or to integrate new business components, without interfering with the rest of the system. It demonstrated how individual components can evolve, for instance to take advantage of new technologies or computational solutions for their functional behaviour, without requiring the other components of the system to be changed or the global configuration of the system to be updated.

The architectural, coordination-based approach that we have developed has been characterised, in a canonical way, through a "prototype" Architecture Definition Language – CommUnity – defined precisely having in mind the formalisation of the separation of concerns that allows for interactions to be modelled as first-class entities through connectors. CommUnity, introduced in [5], was conceived as a parallel program design language, similar to Unity [4] in its computational model, but adopting 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 they interact

The following example illustrates how CommUnity supports this separation. Consider a banking application in which we have to address the way customers can handle their accounts. A simple, abstract model of a bank account can be given by the following CommUnity program:

```
program  account is
out      number, balance: int
in       amount: int
do       deposit[balance]: →  balance:=balance+amount
       □ withdrawal[balance]: →  balance:=balance-amount
```

What we call a CommUnity program (in fact, a special case of what we call *designs* in other papers) is a structure of the following form:

```
program  P is
out      O
in       I
prv      V
do     □,g sh(Γ)          g[D(g)]: G(g) →  R(g)
       □,g prv(Γ) prv     g[D(g)]: G(g) →  R(g)
```

I and *O* are the sets of input and output channels of *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, for instance the amount with which a customer may wish to make a deposit or a withdrawal. The component has no control on the values that are made available in such channels (in the example, it is the customer who decides the value available in *amount*). 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 without involving the actions declared in the program. Output channels allow the environment to read data that the component may calculate from its current state, for instance the current balance of the account. Private channels support internal activity that does not involve the environment in any way.

Γ is a 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.

Guarded commands are the means through which the computational aspects can be described and are associated with actions as follows:

$D(g)$ consists of the local channels into which executions of the action can place values. This is what is sometimes called the *write frame* of g. Given a private or output channel v, we will also denote by $D(v)$ the set of actions g such that $v \in D(g)$. In the example, both actions can only interfere with the value made available in the output channel *balance*. When the write frame can be deduced from the assignments performed by the action, we normally omit it.

$G(g)$ is the enabling condition (guard) of g. We normally omit it when it is tautological.

$R(g)$ is a multiple assignment on $D(g)$. When the write frame $D(g)$ is empty, $R(g)$ is denoted by *skip*.

In contrast to what might be expected in the example, the guard of the command associated with the withdrawal action is true. The idea is that the bank may wish to provide different policies on withdrawals, for instance depending on the type of client, by restricting the guard in different ways. For instance, a strict policy of withdrawals could be programmed as follows:

```
program  standard_account is
out     number, balance: int
in      amount: int
do      deposit: balance:=balance+amount
      ▯ withdrawal: balance•amount → balance:=balance-amount
```

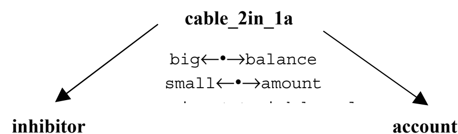
This program is obtained from the previous one by superposing what can be seen as a business rule, in this case a guard restricting withdrawals to states in which the balance is greater than the amount requested by the customer. This kind of superposition can be related to typical uses of inheritance in object-oriented program and illustrates why such forms of evolution interfere with the way the components in place are

programmed: they require the code that implements the withdrawal to be changed, and these changes to be propagated to the components that use the service. That is to say, each time a new business service needs to be offered, its deployment may require the overall system to be redesigned. This hindrance of "traditional" development methods, like object-oriented ones, has been analysed and illustrated in more detail in [1,9].

CommUnity supports instead the externalisation and explicit representation of this business rule as a regulator that can be connected to the account to restrict the way it can be used without interfering with its implementation. The regulator can be designed as follows:

```
program  inhibitor is
in       big, small: int
do       restrict: big•small → skip
```

This program models a very simple component that is able to block an action (restrict) whenever the value that it reads in *big* is smaller than the value it reads in *small*. The idea is to use this inhibitor to model the business rule above by interconnecting it with the account in a way that it reads *big* from *balance* and *small* from *amount*, and synchronises *restrict* with *withdrawal*. This interconnection can be specified by the following diagram:



where

```
program  cable_2in_1a is
in       a, b: int
do       c:
```

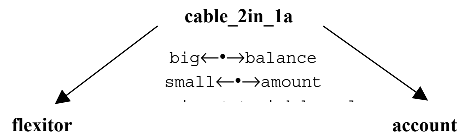
models a component with no computational behaviour and whose role is to perform the bindings between channels and establish the rendez-vous required by the interconnection. The bindings and the rendez-vous themselves are expressed through the arrows. Such diagrams are formal, mathematical objects as shown in [6]. Their semantics is given by the program that represents the joint behaviour of the inhibitor and the account interconnected as specified, which, as proved in [6], is precisely the standard_account.

Diagrams like the one above represent system configurations in which business rules are explicitly represented. Hence, they can be evolved by simply replacing the coordination mechanisms in place. For instance, one can upgrade the standard account to a more flexible account in which an overdraft is made available by replacing the inhibitor by the following regulator:

```
program  flexitor is
in       big, small, over: int
do       restrict: big+over•small → skip
```

For maximum flexibility, the overdraft limit itself is modelled as an input channel meaning that its value can be changed dynamically. Because the coordination on the

use of withdrawal has been separated from its computational aspects, the change of business rule can be performed by replacing *inhibitor* by *flexitor*, which leaves *account* unchanged:



We are aware that this is an oversimplistic example, but its sole purpose is to illustrate what an architectural approach centred on coordination can offer in terms of flexibility in terms of incorporating, dynamically, new business rules, or revising the ones already in place. Again, we refer to some of our previous papers, e.g. [1,9], for a more detailed discussion. What we want to address in this paper are the limitations of the approach in handling the challenges that "ubiquity" raises for business systems.

Indeed, this architectural approach offers only a "logical" view of change. It 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, and that the coordination mechanisms put in place through connectors can be made effective across the "wires" that link components in the underlying communication network.

The effects of mobility on coordination are only now being recognised as an additional factor of complexity, one for which current architectural concepts and techniques are not prepared for. As components move across a network, the ability of locations to deliver the expected services and of the "wires" to support the required interactions will change as well, which may make the connectors in place ineffective and require that they be replaced with ones that are compatible with the new topology of distribution. For instance, deposits and withdrawals are not available when the customer is using internet-banking. Withdrawals at an ATM are limited to a certain daily amount if the balance is not accessible through the network.

Hence, our recent research has focused on making architectures "location and network-aware" [10]. In the remainder of the paper, we will give an outline of the extensions that we are developing to meet this goal.

3 Making CommUnity network aware

The extension that we are proposing for CommUnity to support the design of the distribution and mobility dimension of systems adopts an explicit representation of the space within which movement takes place, but no specific notion of space is assumed. This is achieved by considering that "space" is constituted by the set of possible values of a special data type *Loc* included in a fixed data type specification over which components are designed. The data sort *Loc* models 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. Together with the definition of operations on locations, this provides a description mechanism that is expressive enough to establish location hierarchies or taxonomies, which is crucial in the context of mobile and ubiquitous systems. The only requirement that we make is for a special location –$\perp$– to 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" (output and private channels, actions, or any group of these) are mapped to the positions of the space statically determined by *Loc*. This is achieved by associating each "constituent" of a system with a location variable. Mobility is then associated with the change of value of location variables.

Location Variables. Location variables (locations, for short) are a new syntactic category, all typed with sort *Loc*. Like channels, they can be declared as *input* or *output*. The movement of any constituent located at an input location is under the control of the environment. Output locations can only be modified locally but can be read by the environment. Hence, the movement of any constituent located at an output location is under the control of the component.

Channels. Each output and private channel x of a program is now associated with a location l. We make this assignment explicit by writing $x@l$. 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. Input channels are located at a special location λ whose value is invariant and given by $\perp$. The intuition is that this location variable is a non-commitment to any particular location. The idea is that input channels will be assigned a location when connected with a specific output channel of some other component of the system.

Actions. Each action name g is now associated with a set $\Lambda(g)$ of locations, 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 are associated with located actions, i.e. pairs $g@l$, for $l \in \Lambda(g)$. Their execution can change the value of output locations, thus accounting for self-inflicted mobility.

It is important to notice that, by using the special location λ, we keep the possibility of designing location-unaware systems. Every "standard" CommUnity program defines a canonical distributed program in which every action and channel is considered to be located at λ.

Variations in the context of execution of a mobile system are not limited to the locations of its components and respective hosts. It is important that other observables, e.g. network bandwidth, battery power or the communication range, can be used at the programming level. These can be handled by data type constructs, for instance *inrange:Loc→bool* for modelling observations of whether given positions of the space are in the communication range of a location.

Semantics. The distribution space consists of the set of possible values of the given data sort *Loc*. 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 *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.

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 local guard G(g@l) evaluates to true*

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

In order to illustrate the extension, consider the bank account again. According to the motivation we gave in the introduction, it makes sense to assign a specific location to each account in order to reflect the fact that customers may invoke services from locations different from their bank's:

```
program  account is
outloc branch
out     number@branch, balance@branch: int
in      amount: int
do      deposit@branch: balance:=balance+amount
      [] withdrawal@branch: balance:=balance-amount
```

Because the location *branch* of the account is declared as an output variable, it cannot be modified by the environment. On the other hand, none of the actions of

account changes the value of *branch*. This means that the location of an account, once it is set at configuration time, remains unchanged. That is to say, *account* is a located but non-mobile component.

It is now up to the bank to establish different rules for customers to use the services of their accounts depending on their location. We might do this by extending *account* as follows:

```
program  mobile_account is
outloc branch
inloc  cust
out    number@branch, balance@branch: int
in     amount, lim_atm: int
do     deposit
          @branch: balance:=balance+amount
          @cust: branch=cust ∨ atm(cust) → skip
     ▯ withdrawal
          @branch: balance:=balance-amount
          @cust: branch=cust ∨ (atm(cust)∧amount•lim_atm)) → skip
```

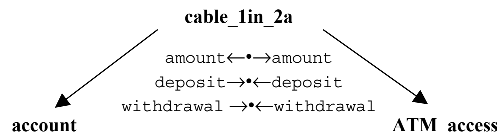
That is to say, we are making the account aware of the location of the customer through the input location *cust*. As a consequence, each of its actions is now distributed over both locations. The projection of each action on the location of the customer is now guarded according to a given business rule. For instance, when the customer is using an ATM, withdrawals cannot exceed a given limit.

Like in the previous section, this extension is intrusive on *account*. In fact, each time a new type of location is made available, the *account* needs to be redesigned to reflect the rules that determine how its services can be used from locations of that type. Even worse, rules that are related to locations can now become mixed together with the other business rules!

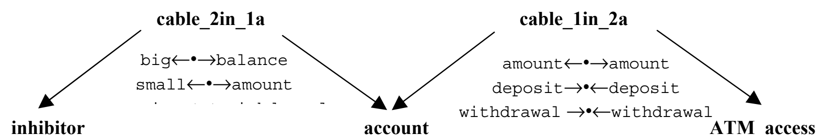
Again, CommUnity allows for the mobility aspects to be separated from the coordination and computational ones. For each type of location, we can define a distribution connector that enforces the business rules that apply to that location. For instance, the distribution connector that controls access through an ATM could be designed as follows:

```
program  ATM_access is
inloc  cust: Loc
in     amount, lim_atm: int
do     deposit@cust: atm(cust) → skip
     ▯ withdrawal@cust: atm(cust)∧amount•lim_atm → skip
```

Notice that the actions of the connector are performed in the location of the customer, meaning that the controller is co-located with the customer. The superposition of this business rule on the account can be made through the following configuration diagram:



Other rules can now be superposed and evolved independently of the location:



We have thus achieved a complete separation of three different concerns – computation, coordination and distribution, leading to architectures in which each of the concerns can evolve independently of the others.

4 Concluding remarks

In this paper, we presented extensions to a prototype architecture description language – CommUnity – to illustrate how interactions can be made network-aware and distribution concerns can be separated from calculations and location-independent coordination mechanisms. Location awareness at this architectural modelling level should be distinguished from the other ways of addressing system mobility that can be found in the literature, which take place in the context of programming languages or process calculi (e.g., Cardelli's Ambient Calculus [3]). We believe that, for application domains in which location-awareness is intrinsic, distribution and mobility are aspects that should be addressed as first-class citizens from the very early levels of modelling and be represented explicitly in the business architecture. Postponing their treatment to properties of the programming and middleware platforms precludes a level of separation between concerns that can support the levels of flexibility that systems are required to exhibit to operate in volatile and turbulent business environments.

This is the first step that we have taken to extend architectural approaches in order to meet the challenges that the ability of being on-line anywhere and at any time is raising on software development. In [12], locations have been addressed not from first principles but as any other kind of data, similarly to Mobile Unity [8]. Again, this prevents a proper separation of concerns and a corresponding architectural support to business policies that are location-dependent. Restrictions on the length of the paper prevent us from illustrating our approach in greater detail (case studies can be found in [http://boogie.pst.informatik.uni-muenchen.de/Agile]) as well as its mathematical semantics (which can be found in [10]).

Research is now progressing in order to provide a language and semantics for addressing the need to program operations that reconfigure the system in reaction to changes in the distribution topology, i.e. to make systems aware and self-adaptable to mobility. For instance, in the case of the banking application that we used, the replacement of an ATM_access connector by an Internet_access one should be automatically performed whenever the customer, through its PDA, moves away from the ATM and starts using the internet though its wireless connection. We hope to be able to report on this new important facet in the near future.

Acknowledgements

This work has been partially supported by the project IST-2001-32747 AGILE – Architectures for Mobility.

References

[1] L.F.Andrade and J.L.Fiadeiro, "Agility through Coordination", *Information Systems* 27, 2002, 411-424.

[2] L. Bass, P.Clements and R.Kasman, *Software Architecture in Practice*, Addison Wesley 1998.

[3] L.Cardelli and A.Gordon, "Mobile Ambients", in *Foundations of Software Science and Computational Structure*s, LNCS 1378, Springer-Verlag 1998, 140-155.

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

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

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

[7] D.Gelernter and N.Carriero, "Coordination Languages and their Significance", *Communications ACM* 35, 2, pp. 97-107, 1992.

[8] R.Gruia-Catalin, P.J.McCann and J.Y.Plun, "Mobile UNITY: reasoning and specification in mobile computing", *ACM TOSEM*, 6(3), 250-282.

[9] G.Koutsoukos, T.Kotridis, L.Andrade, J.L.Fiadeiro, J.Gouveia and M.Wermelinger, "Coordination technologies for business strategy support: a case study in stock-trading", in R.Corchuelo, A,Ruiz and M.Toro (eds), *Advances in Business Solutions*, Catedral Publicaciones 2002, 45-56.

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

[11] M.Shaw, "Procedure Calls are the Assembly Language of Software Interconnection: Connectors Deserve First-Class Status", in D.A. Lamb (Ed.), *Studies of Software Design*, LNCS 1078, Springer-Verlag 1996.

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