ARCHITECTURAL TECHNIQUES FOR EVOLVING CONTROL SYSTEMS

L.F.Andrade[1], J.L.Fiadeiro[2], A.Lopes[3], M.Wermelinger[4]
[1] *ATX Software S.A.*
Address: Alameda António Sérgio 7, 2795-023 Linda-a-Velha, Portugal
Phone: (+351) 210120500, Fax: (+351) 210120555, e-mail: landrade@atxsoftware.com
[2] *Department of Mathematics and Computer Science, University of Leicester*
Address: University Road, Leicester LE1 7RH, United Kingdom
Phone: (+44) 1162523907, Fax: (+44) 1162523915, e-mail: jose@fiadeiro.org
[3] *Department of Informatics, Faculty of Sciences, University of Lisbon,*
Address: Campo Grande, 1749-016 Lisboa, Portugal
Phone: (+351) 217500087, Fax: (+351) 217500084, e-mail: mal@di.fc.ul.pt
[4] *Department of Informatics, Faculty of Sciences and Technology, New University of Lisbon,*
Address: Quinta da Torre, 2829-516 Caparica, Portugal
Phone: (+351) 212948536, Fax: (+351) 212948541, e-mail: mw@di.fct.unl.pt

Abstract: We propose a layered architecture based on the separation of two concerns – computation and coordination – as a means of achieving higher levels of reconfigurability for control systems. This separation makes it possible for changes to be operated on the way system components are interconnected, which includes the way they are controlled, without intruding on the way they are deployed, enabling the overall system to evolve and adapt to changes detected on the environment without interrupting execution. Such features contribute to the new levels of flexibility and agility that are being required of control systems that operate critical infrastructures, like railway operation, in networked and distributed environments.

Keywords: architectures, connectors, coordination, dynamic reconfiguration, evolution

1. INTRODUCTION

The evolution of the internet and wireless communication is inducing an unprecedented and unpredictable variety and complexity on the roles that software can play in building control systems, leading to a growing dependency of vital functions of our modern society (telecommunications, financial services, transports, energy supplies, etc) on software-intensive systems often built over unreliable networks of heterogeneous, fragile platforms. As a result, one of the biggest challenges that the Software industry is facing today is to endow these systems with the levels of agility that are required to ensure that they can easily react and adapt to changes in the environment in which they operate or which they control, so that some agreed level of service is maintained. If the complexity of constructing Networked and Distributed Systems in general, and control systems in particular, was challenging enough before the advent of these new technologies, the ability to make them evolve is clearly growing beyond the reach of Formal Description Techniques as we know them today: there is a panoply of methods and techniques that can be used for *designing* evolvable systems, but hardly any formal support at the levels of abstraction at which the need for change is perceived and, hence, should be handled. This is especially true when one takes into account the need for control systems to react and operate in run-time, "just-in-time", without interrupting services that are often safety-critical.

Our purpose in this paper is to show that a step in the right direction can be taken by adopting what we have called "coordination technologies" (Andrade and Fiadeiro 2001) – a set of architectural principles and modelling primitives based on the separation between what in systems can account for the operational aspects (what we call "computations" in general) that are responsible for the behaviour that individual components ensure locally, e.g. the functionality of the services that they offer, and the mechanisms that control their behaviour and coordinate the interconnections through which global properties of systems can emerge. The idea is that, through this separation between Computation and Coordination, we will be able to support the externalisation, and definition as first-class citizens, of the rules according to which the joint behaviour of given components of a system are being controlled. The representation of such rules can then be evolved independently of the components that they coordinate, thus introducing new degrees of agility and flexibility with respect to changes taking place in the domain over which control is being superposed.

The major challenge that we face, and that justifies this paper, is to take into account the distribution/mobility aspects, so far ignored or neglected when the evolution of control systems is concerned. Yet, it is crucial that, when we reconfigure a system by replacing components or interconnecting them in different ways, we take into account the support that locations provide for the operational/computational aspects of the individual components, and the ability for the new interconnections to be effective over the communication network. Furthermore, it is essential that the system as a whole may self-adapt to changes occurring in the network topology, either to maintain agreed levels of quality of service, or to take advantage of new services that may become available.

Because we are still in the early stages of this endeavour, the paper concentrates on the formal model that supports Description Techniques, leaving methodological and pragmatic issues for another opportunity. The proposed model is based on concepts and mechanisms that have been made available for Parallel Program Design, namely those that support the Unity language (Chandy and Misra 1988) such as the notion of superposition, and also on contributions from the area of "Coordination Languages and Models" (Gelernter and Carriero 1992), "Software Architectures" (Bass *et al* 1998), and Reconfigurable Distributed Systems" (Magee and Kramer 1996). All these different contributions are integrated over a common mathematical framework that builds on Goguen's categorical approach to General Systems Theory (Goguen 1973).

With respect to our previous work on architectures, we must point out that we are here addressing for the first time the distribution/mobility aspects as they apply to control-system development. In other papers (Wermelinger and Fiadeiro 1998), which include the example of the luggage transportation system as well, locations are dealt with not from first principles but as any other kind of data, similarly to Mobile Unity (Gruia-Catalin *et al* 1997). This prevents a proper separation of concerns and a corresponding architectural support. The extension with logical notions of location and network awareness are now being developed within the research project AGILE. A preliminary account of it can be found in (Lopes *et al* 2002). This is our second paper on the subject and the first application of architectural concerns to control-systems that addresses network-awareness.

2. DESIGN IN COMMUNITY

CommUnity, introduced in (Fiadeiro and Maibaum 1997), is a parallel program design language that is similar to Unity (Chandy and Misra 1988) 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 variable 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.

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 $<S,\square>$, where S is a set (of sorts) and $\square$ is a $S^{*}\square S$-indexed family of sets (of operations), to be given together with a collection $\square$ of first-order sentences specifying the functionality of the operations.

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 *LOC* included in the fixed data type specification over which individual component behaviour is 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. The only requirement that we make is that a special location –*⊥*– 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, as they arise for typical control systems.

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.

A CommUnity design (in fact, a restricted form of design that we have called "program" in other papers), is of the form:

```
design  P is
outloc  O'
inloc   I'
out     O
in      I
prv     V
do
   []       {g@l[D(g@l)]:G(g@l)→ R(g@l)
g∈sh(Γ)
                  | l∈Λ(g)}

   []     prv{g@l[D(g@l)]:G(g@l)→ R(g@l)
g∈prv(Γ)
                  | l∈Λ(g)}
```

The sets *I* and *O* correspond to the 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∪O∪V* and *local(X)* to denote the union *V∪O* of *local* channels. Each channel *v* is typed with a sort *sort(v)∈S.*

Locations in a component design can be declared as *input* or *output* in the same way as channels but are always typed with sort *LOC*. Input locations *(I')* 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 *(O')* 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.

Each local 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. Input channels are located at a special location *⊥*, which, intuitively, 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. We use *L* to denote the union *I'∪O'* and *outloc(L)* to denote *O'*.

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.

Each action name *g* is associated with a set *Λ*(*g*) of locations. This means that the execution of action *g* is distributed over the locations in

□*(g)* in the sense that its execution involves 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*□□*(g)*, as follows:

- *D(g@l)* is a subset of *local(X)*□*O'* consisting of the local channels into which executions of the action can place values and the location variables that it can change. This is what is sometimes called the *write frame* of *g@l*. For simplicity, we will often omit the explicit reference to the write frame when *D(g@l)* 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@l* such that *v*□*D(g@l)*. We denote by *F(g@l)* the *frame* of *g@l*, i.e., the channels that are in *D(g@l)* or used in *G(g@l)*.
- *G(g@l)* is the guard condition.
- *R(g@l)* is a conditional multiple assignment on the local channels declared in *D(g@l)*. When the write frame *D(g@l)* is empty, *R(g@l)* is denoted by *skip*.

As an example, consider the typical airport luggage delivery system in which carts move along a track and stop at designated locations for handling luggage. We model locations in the track through the natural numbers modulo the length *L* of the circuit, i.e. *LOC* is nat_L. A CommUnity design that models the behaviour of a cart can be given as follows:

```
design  cart is
outloc  pos
in      next:LOC
prv     dest@pos: LOC
do
   move@pos: [pos≠dest □ pos:=inc(pos)]
[] handle@pos: [pos=dest □ dest:=next]
```

That is to say, a cart is able to move and handle luggage. It moves by incrementing its position (available on location *pos*), while it has not reached its destination. Handling luggage takes place only at the destination, which is available locally in *dest* and is recomputed when the cart reaches the destination for handling the luggage. The fact that *dest* is local to the cart means that the environment cannot change the destination of the cart until it reaches the pre-assigned one. There, the environment can control where the cart will go next because *next* is an input channel.

3. COORDINATION IN COMMUNITY

Consider now a typical situation of changes that need to be brought, in run-time, into control systems. Assume that we need to control/monitor the behaviour of the cart by observing how many times it visits a given location in the circuit. The typical way of doing so in languages like Unity would be to replace the program by an extension – a monitored version of it:

```
design  monitored_cart is
inloc   next, cpoint
outloc  pos
out     count@cpoint:nat
prv     dest@pos: LOC
do
c&move
  @pos: [pos≠dest □ pos:=inc(pos)]
  @cpoint:[pos=cpoint □ count:=count+1]
[] j&move@pos:
      pos≠dest□pos≠cpoint □ pos:=inc(pos)
[] handle@pos:
      pos=dest □ dest:=next
```

The extension (highlighted in **bold**) of the original design includes a new output channel that makes available the number of times the cart has gone through a given control point. The location of the control point is determined by the environment, which means that the passage of the cart can be monitored at different locations throughout its life. The *move* action is now split in two: *j&mov* accounts for movements that do not go through the control point, and *c&move* for those that do (thus incrementing *count*). The execution of action *c&move* is distributed: locally, i.e. at location *pos*, it increments its location unless it has reached its destination; when the location coincides with the control point, it further increments the counter.

This sort of extension is supported in languages for parallel program design like Unity through the notion of *superposition*: the program *monitored_cart* is obtained from *cart* by superposing additional behaviour accounting for the required monitoring activity. Extension by superposition follows certain rules that make sure that the original program is "protected" in a very precise sense. The notion of superposition that can be found in the literature (Back and sere 1996; Francez and Forman 1996; Katz 1993) does not address mobility explicitly. Therefore, what we propose below is an extension of the notion of superposition morphism that we have used in the past (Fiadeiro and Maibaum 1997) that takes the distribution aspects into account: given designs P_i with channels X_i, actions $□_i$ and locations L_i, a superposition 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 preserves the pointed element ($\bot$), satisfying:

1. for every $x \in X_1$, $l \in L_1$
   $sort_2(\sigma_{ch}(x))=sort_1(x)$
   if $x \in out(X_1)$, $\sigma_{ch}(x) \in out(X_2)$
   if $x \in in(X_1)$, $\sigma_{ch}(x) \in out(X_2) \cup in(X_2)$
   if $x \in prv(X_1)$, $\sigma_{ch}(x) \in prv(X_2)$
   if $l \in outloc(L_1)$, $\sigma_{lc}(l) \in outloc(L_2)$
   if $x \in local(X_1)$, $\sigma_{lc}(\Lambda_1(x)) = \Lambda_2(\sigma_{ch}(x))$
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 $\sigma_{ac}(g) \in prv(\Gamma_1)$
   $\sigma_l(\Lambda_1(\sigma_{ac}(g))) \subseteq \Lambda_2(g)$
3. for all $x \in X_1 \cup outloc(L_1)$ and $g@l_2 \in D_2(\sigma(x))$
   $\sigma_{ac}(g)$ is defined and $\sigma_{ac}(g)@l_1 \in D_1(x)$ for some $l_1 \in \sigma_{lc}^{-1}(l_2) \cap \Lambda_1(\sigma_{ac}(g))$
4. for every $g \in \Gamma_2$ s.t. $\sigma_{ac}(g)$ is defined and $l \in \sigma_{lc}^{-1}(\Lambda(g))$
   $\sigma_{ch}(D_1(\sigma_{ac}(g)@l)) \subseteq D_2(g@\sigma_{lc}(l))$
   $\models_\Phi (R_2(g@\sigma_{lc}(l)) \supset \sigma(R_1(\sigma_{ac}(g)@l)))$
   $\models_\Phi (G_2(g@\sigma_{lc}(l)) \supset \sigma(G_1(\sigma_{ac}(g)@l))$

where $\models_\Phi$ means validity in the first-order sense taken over the axiomatisation of the underlying data types (which includes the location space).

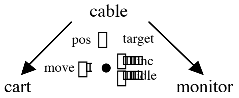
The functions account for the extension. Notice that input channels may become output channels through superposition in the sense that the extension may account for the part of the environment from which values were being read. The morphisms on actions are partial and contravariant to account for the fact that, on the one hand, superposition may unfold actions of the original program (as in the example) and, on the other hand, new actions may be added. The restrictions are the usual ones for superposition: the guards of the actions can be strengthened and the assignments can be made more deterministic. New actions are also restricted to leave the old channels out of their scope (encapsulation).

Superposition morphisms support a discipline of evolution in which system components are extended according to the rules above, which blends well with the traditional stepwise refinement method initiated by (Dijkstra 1976) and adapted to reactive system design by (Back and Sere 1996). However, such a form of evolution does not correspond to the approach that we advocated in the introduction because it requires changes to be carried out on the way components have been implemented, e.g. by changing the guards of given actions or extending the assignments that they perform. The way we want to support the evolution of the system is by leaving the original component unchanged and interconnecting to it an explicit controller that ensures the required monitoring activity. The required monitor can be designed as follows:

```
design  monitor is
inloc   base, target
out     count@base:nat
do  inc@base:
    [base=target → count:=count+1]
[]  idle@target:
    [base≠target → skip]
```

This is a component that makes available a channel *count* at a location *base* controlled by the environment – meaning that the environment can change the location of the monitor – in which it places a value that is incremented by action *inc* which is enabled when *base* coincides with the location *target*. When *target* does not coincide with *base*, the monitor is idle.

In order for this component to monitor the behaviour of the cart as intended, we need to interconnect it to *cart*. The required interconnection is expressed through the following diagram



where

```
design  cable is
inloc   a
do      g: [→ ]
```

The “cable” that interconnects the cart and the monitor is itself a design but one that does not involve any computation. It provides a pure coordination function by establishing the means for the two components to communicate. In the case of the example above, we need a "cable" with two connections: one for the monitor to read the location of the cart, and another for the move action of the cart to synchronise with those of the monitor.

Given that *cable* models the medium through which data is to be transmitted, it is location-unaware in the sense that its action is not “performed” at any particular location (there is nothing to perform...). This is what the special location $\bot$ is used for: whenever no location is assigned to an action or a channel, it is implicitly assumed that this location is $\bot$ with the meaning that their location is not meaningful.

The interconnection itself is established through the arrows. The left arrow maps *a* to *pos* and *move* to *g*, and the right arrow maps *a* to *pos* and both *inc* and *skip* to *g*. Because there are no computations involved in the cable, the arrows are trivial superposition morphisms. Hence, the diagram is a categorical entity i.e. a mathematical object which, in this case, expresses a given system configuration. Such a configuration diagram establishes input/output relations, as in the case of *pos*, as well as synchronisation sets as in the cases of *{move,inc}* and *{move,idle}*. This semantics of the configuration is the one that corresponds to taking the colimit of the diagram. The colimit returns a program whose behaviour corresponds to the joint behaviour of the components thus interconnected. In the case above, it is easy to prove that the colimit returns, up to isomorphism (i.e. renaming of the channels and actions) the program *monitored_cart*.

It is not possible to provide herein the full explanation of how colimits in the category of programs and superposition morphism work. See (Fiadeiro and Lopes 1999) instead. In the case of interconnections such as the one above, and besides establishing these input/output relations and synchronisation sets as illustrated, the colimit assigns to each synchronisation set an action whose guard is the conjunction of the guards of the actions in the set, and whose assignments are given, for every output channel, by the union of the assignments performed locally by the actions in the set. This operation captures what in IP (Francez and Forman 1996) is actually called superposition – a generalised parallel composition operator based on a *rendez-vous* synchronisation mechanism. In this case, cables can be seen to provide the places in which the *rendez-vous* take place.

It is this pattern of extension that we wish to support evolution as motivated in the introduction: components are not extended in the sense that we rewrite whatever features we want to change (even if subject to certain rules), but superposed with explicit controllers that regulate their behaviour without intruding in the way they are deployed. In fact, the component will not even know that is being controlled in the sense that the coordination is performed externally. This is particularly important for networked and distributed systems in general, and control systems in particular, because we may want to assign different locations to the new components that are being superposed, have their locations controlled by the environment as in the case of the monitored cart, or have them distributed across different locations.

This degree of flexibility can be achieved because, in languages like CommUnity, the mechanisms through which interactions between system components are coordinated can be completely externalised from the code that implements them (in the case of software components) or, more generally, the mechanisms that are responsible for their local behaviour, and modelled explicitly as first-class citizens.

CommUnity was developed precisely to illustrate how this separation can be supported by Formal Description Techniques (Fiadeiro and Lopes 1999). It is not itself a language that we can use for the day-to-day analysis and design activities. For this purpose, we have developed a set of modelling primitives that build on the separation between Computation and Coordination, which can be used to extend languages like the UML (Booch *et al* 1998).

The centrepiece of this modelling kernel is the notion of *coordination contract* (Andrade and Fiadeiro 1999). A coordination contract makes available the expressive power of a *connector* in the terminology of Software Architectures (Allen and Garlan 1997). It consists of a prescription of *coordination effects* (the *glue* of the connector) that are superposed on a collection of partners (system components) when the occurrence of given *triggers* is detected in the system. For instance, the monitor that we discussed above is an example of a coordination contract: as a connector, it has a single role (formal parameter). This role identifies the abstract properties that the components that can be monitored have to exhibit: basically that, in their public interfaces, they make available a trigger that allows for the monitor to detect the passage of the cart through the control point. The specification of such interfaces is essential for modelling control systems because it abstracts away from the nature of the components that can be controlled (software, mechanical, human, and so on), making connectors/controllers reusable pieces of design that need not change if the nature of the components being controlled changes but still comply with the interface. The glue of the monitor as a connector consists of the counting process that we described above. This glue exhibits a public interface through which the current value of the counter is made available.

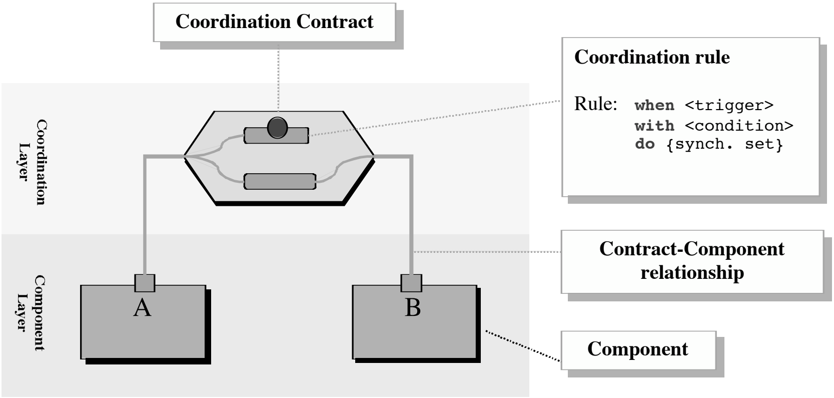


Figure 1: Controllers are superposed over core components as coordination contracts

Space is too scarce for a proper definition and explanation of the language, methodology and pragmatics associated with the coordination primitives that we have developed. In the next and final section, we will discuss the implications that the adoption of an architecture based on the proposed separation of concerns can have for the evolvability of control systems.

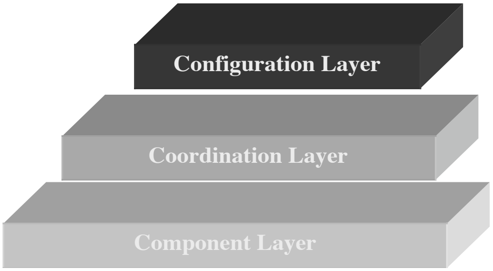
4. A LAYERED ARCHITECTURE

The approach to control system development that we have been advocating is based on a layered architecture that we have been building based on a methodological and technological separation between the behaviour that can be ensured locally by system components (e.g. computations performed by software services) and the mechanisms that coordinate their interaction. We have been applying this architectural approach to different business domains, and extending it now to the development of control and embedded systems. Components in this sense correspond to "core", "stable" entities (even if mobile) in the sense that evolution should not be intrusive on their implementation but, instead, make different uses of the services that they provide or adapt them to new circumstances through the superposition of coordination mechanisms (controllers). These are modelled explicitly and outside the components in the form of what we have named "coordination contracts", leading to a two-tiered architecture for distributed and networked systems in which basic components and coordination contracts reside in different layers as depicted in *Figure 1*.

Coordination contracts ensure that given global properties of the system will emerge from the local behaviour of the components and the interactions established between them through the contracts. They are the means through which we propose that evolution be conceived in the sense that, in order to be adapted to new requirements, the system should be reconfigured by removing existing contracts and plugging in new contracts from which different behaviour can emerge.

Having mechanisms for evolving systems is not the same as prescribing when and how these mechanisms should be applied. Evolution is a process that needs to be subject to rules that aim at enforcing given global properties for systems that should be evolution invariant. For this purpose, we have developed a modelling primitive – coordination contexts – through which the reconfiguration capabilities of the system can be automated, both in terms of ad-hoc services that can be invoked by authorised users, and programmed reconfigurations that allow systems to react to well identified triggers and, hence, adapt themselves to changes brought about on their state or configuration.

For instance, in order to avoid collisions between carts moving along the same track, we may superpose binary coordination contracts on pairs of carts to synchronise their movements: a cart that is just behind another one should not move until the front one has moved. The glue of the contract can be designed as follows:



Ad-hoc and programmed reconfiguration operations that support evolution, including auto-adaptation.

Coordination units (controllers) that can be plugged and removed dynamically without interfering with the component layer.

Core components, possibly of different nature, independent of each other.

Figure 2: Architectural separation of three key concerns for evolution

```
design   subsume is
do       sync: [true □  skip]
    []   free: [true □  skip]
```

The idea is to synchronise the move action of the cart behind with *sync* and that of the cart in front with both *sync* and *free* so that the cart in front can move freely but the cart behind just moves synchronised with the one in front.

Because this coordination mechanism needs to be in place when there is a risk of collision, it makes no sense to have it superposed on permanence. What we would like to do is to program the configuration of the system in a way that the contract is superposed whenever the risk of collision arises. This is what coordination contexts are for: they consist of sets of trigger/reaction rules that can react to changes occurring in the state of the system by reconfiguring it, such as:

```
when□front.pos+1<behind.pos
do    superpose subsume(
         sync□ {front.move,behind.move},
         free□ front.move)
```

This also shows how we can build self-adaptable systems.

Coordination contexts reside on a third-layer of the architecture (see *Figure 2*) above the component and coordination layers: it uses components and contracts for reconfiguring the system but components and contracts are developed without knowing in which contexts they will be used.

5. CONCLUDING REMARKS

In this paper, we have outlined how a layered architecture that we have been building for supporting the development of information systems that can evolve in their application domain (Andrade and Fiadeiro 2001), can be extended to address the higher levels of reconfigurability that are being required of control systems, mainly those which operate critical infrastructures like railway operation. Essentially, the proposed architecture is based on the explicit identification, as first-class citizens, of the mechanisms that model interaction rules and other aspects of the application domain that require that the behaviour of its components be coordinated in order to achieve certain effects.

Given that, in many application domains, changes are required because of the need to introduce new or replace existing coordination mechanisms, we advocate that corresponding changes in running systems should be supported through the dynamic reconfiguration of the coordination mechanisms, without having to change the way the components being coordinated are implemented. In this way, a true "plug-and-play" approach to control-system development can be supported.

The significance of this approach for control systems stems not only from the general requirements that the Net and Wireless Communications are putting on the ability for systems to be agile in reaction to change, but also from the fact that Distribution and Mobility are two aspects that cannot be ignored in this equation: changes on systems that are distributed across communication networks need to take into account the properties of the locations in which components perform their computations and the properties of the media through which their interconnections can be effectively coordinated. This is why we are working on abstract characterisations of “Distribution contracts” and on ways through which they can be made explicit at given architectural levels as a means of understanding, controlling and managing mobility across distribution topologies. Preliminary work in this direction can be found in (Lopes *et al* 2002).

ACKNOWLEDGEMENT

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

REFERENCES

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

Andrade, L., and J.Fiadeiro (1999). "Interconnecting Objects via Contracts", in *UML'99 – Beyond the Standard*, R.France and B.Rumpe (eds), LNCS 1723, Springer Verlag 1999, 566-583.

Andrade, L., and J.Fiadeiro (2001). “Coordination Technologies for Managing Information System Evolution", in Proc. CAISE’01, K.Dittrich, A.Geppert and M.Norrie (eds), LNCS 2068, Springer-Verlag, 374-387.

Back, R. and K.Sere (1996). “Superposition Refinement of Reactive Systems”, *Formal Aspects of Computing* 8(3), 324-346.

Bass, L., P.Clements and R.Kasman (1998). *Software Architecture in Practice*, Addison Wesley.

Booch, G., J.Rumbaugh and I.Jacobson (1998). *The Unified Modeling Language User Guide*, Addison-Wesley.

Chandy, K. and J.Misra (1988). *Parallel Program Design – A Foundation*, Addison-Wesley.

Dijkstra, E. (1976). *A Discipline of Programming*, Prentice-Hall International.

Fiadeiro, J. and T.Maibaum (1997). "Categorical Semantics of Parallel Program Design", *Science of Computer Programming* 28, 111-138.

Fiadeiro, J. and A.Lopes (1999). "Algebraic Semantics of Coordination, or what is in a signature?", in *AMAST'98*, A.Haeberer (ed), LNCS 1548, Springer-Verlag.

Francez, N. and I.Forman (1996). *Interacting Processes*, Addison-Wesley.

Gelernter, D. and N.Carriero (1992). "Coordination Languages and their Significance", *Communications ACM* 35, 2, 97-107.

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

Gruia-Catalin, R., P.J.McCann and J.Y.Plun (1997). “Mobile UNITY: reasoning and specification in mobile computing”, *ACM TOSEM*, 6(3),250-282.

Katz, S. (1993). "A Superimposition Control Construct for Distributed Systems", ACM TOPLAS 15(2), 337-356.

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

Magee, J. and J.Kramer (1996). "Dynamic Structure in Software Architectures", in 4th Symp. on Foundations of Software Engineering, ACM Press, 3-14.

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