FESCA 2005 Preliminary Version

An Architectural Approach to Mobility - The Handover Case Study ⋆

.

Cristóvão Oliveira [1]

Department of Computer Science
University of Leicester
Leicester, UK

Michel Wermelinger [2]

Computing Department
The Open University
Milton Keynes, UK

José Luiz Fiadeiro [3]

Department of Computer Science
University of Leicester
Leicester, UK

Antónia Lopes [4]

Dep. de Informática
Univ. de Lisboa
Lisboa, Portugal

Abstract

COMMUNITY is a formal approach to software architecture. It has a precise, yet intuitive mathematical semantics based on category theory. It supports, at the methodological level, a clear separation between computation, coordination, and distribution (including mobility). It provides a simple state-based language, inspired by Unity, for describing component behaviour. Finally, it addresses composition as a first class concern and accounts for the emergence of global system properties from interconnections. This paper describes the approach and tool support by modelling essential aspects of the GSM handover protocol. We also sketch a framework that we are implementing for the distributed execution of such specifications, using Klava, a Java library for mobile agent systems based on tuple spaces by our project colleagues from Firenze.

Key words: Architecture, composition, (dynamic) configuration, connector, coordination, distribution, mobility, superposition.

1 Introduction

COMMUNITY provides, like Darwin [10], Wright [1], or LEDA [3], among others, a formal approach to software architecture. It has several advantages over other approaches, the main one being a precise mathematical semantics: architectures are not just depicted through lines and boxes; they are diagrams in the sense of category theory [5], involving explicit superposition and refinement relationships between architectural components. This graphical semantics (in both the mathematical and visual sense) mirrors closely and intuitively the design of the architecture. The categorical semantics has also allowed us to study connectors in a deep, rigorous, and language-independent way [6]. In particular, we have defined higher-order connectors and shown how they can be used to put together complex interactions (e.g., an encrypted compressed asynchronous communication) from simpler ones in a systematic way [9].

In addition, COMMUNITY describes component behaviour with a parallel program design language that is at a higher level of abstraction and is more convenient to use than the process calculi employed by most formal approaches to software architecture. The combination of this state-based language with the graph-based semantics of architecture allowed us to define run-time reconfiguration in an intuitive and formal way using typed graph transformation rules. The typing corresponds to a simple notion of architectural style that is preserved during reconfiguration [16].

Like most architectural approaches, COMMUNITY enforces a strict separation between computation and coordination. More recently, we have extended the approach with a small set of semantic concepts and syntactical constructs in order to include a distribution dimension that is kept separate from the other two [8]. The granularity of distribution is very fine in order to provide maximum flexibility: the execution of an action may be distributed over many different locations; at each location, it performs whatever computations are prescribed according to the available resources, and communicates with its environment through the communication channels that are in touch with the location. Changes to locations correspond to mobility of data or code. So far we have illustrated our approach with small examples. The goal of this paper is to show how it applies to larger examples taken

[⋆] This is an extended version of a short paper presented at WICSA'04) [14]

[1] Email: co49@mcs.le.ac.uk

[2] Email: M.A.Wermelinger@open.ac.uk

[3] Email: jose@fiadeiro.org

[4] Email: mal@di.fc.ul.pt

from real applications in which distribution and mobility are key concerns. For this purpose, we chose to model the GSM handover protocol. Due to space constraints, we only show a fragment; the complete details are given in [12].

In order to show how all the features of CommUnity can be used in practice, we have been developing in the last few years a workbench [13] as a proof of concept of the formal framework. The Workbench provides a graphical integrated development environment to write, run, and debug designs and to draw software architectures (configurations). This paper provides a more detailed description of the tool than previously available, in particular of the features recently added to cover the mobility dimension. These features are explored in the modelling of the GSM handover.

The structure of this paper is as follows. We start with the description of the handover protocol as we use it. In Section 3 we present an informal review of CommUnity. Formal definitions can be found in the papers cited above. Examples are taken from the case-study. Next, we describe the CommUnity Workbench and the architecture description language. In Section 5 we show how the handover protocol can be modelled with CommUnity and tested with the workbench. We finish the paper with some concluding remarks.

2 Handover Protocol

As an example of mobile network, we consider the Public Land Mobile Network that implements the world-wide adopted GSM network—a living and evolving wireless communications standard. Specifically, we concentrate on the handover process [15], which is what controls the dynamic topology of the network.

In order to illustrate the approach as clearly as possible, we concentrate on the following nodes of the GSM network:

- **Mobile Station (MS):** This node is made up of the Subscriber Identification Module (SIM) and the Mobile Equipment. The SIM is a separate physical entity containing all information regarding the subscription. The Mobile Equipment is the piece of hardware that enables radio communication with the system.
- **Base Station System:** This node is composed of some Base Transceiver Stations (BTS) and one Base Station Controller (BSC). The BTS is the radio equipment whose main task is the radio communication with the MS's. Each BTS covers a cell with transmitted radio wave. The BSC controls the communication and supervises the behaviour of all its underlying BTS's.
- **Mobile Services Switching Center (MSC):** This node sets up, supervises, and releases calls of some BSC's. It connects the calls within the GSM network and/or actuates as a gateway to the Public Switched Telephone Network or to some other networks.

The handover is a process required for ensuring the continuity of a call of a MS moving across a BTS cell boundary; without such a process the communication could be terminated because of the limited range of the BTS. As described in [15], the purpose of the handover procedure is to move data and control channels of the call from the BSC currently communicating with the MS (what we call the 'old' BSC) to a BSC in another cell (the 'new' BSC). When the MS is busy (during a call), the decision about which cell is the best for the dedicated connection is done by the BTS and the BSC. This procedure consists in evaluating the received radio signal measurements from the MS.

In the GSM network, there are three types of handover depending on the BTS's involved [17]:

- **Intra-BSC Handover:** In this case the handover process concerns two BTS's belonging to the same BSC.
- **Inter-BSC Handover:** The handover process concerns two BTS's but of two different BSC's controlled by the same MSC.
- **Inter-MSC Handover:** The handover process in this case is between two BTS's belonging to different BSC's controlled by different MSC's.

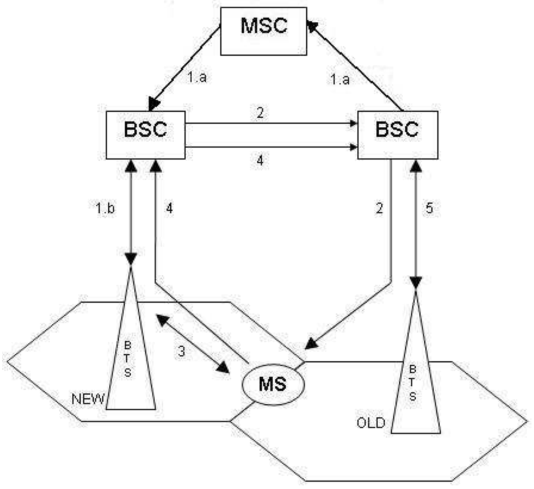


Fig. 1. Handover procedure

In this paper, we focus on the Inter-BSC handover process (see Figure 1) due to the fact that this type of handover is simpler than the third one but nevertheless involves all the steps required in a complete handover process. These steps can be described as follows:

- **Handover request** : The old BSC detects the necessity of the handover with the last received information from its BTS's, suspends the transmission of all messages except for the radio resource management sub-layer messages with the MS, and sends the message 'handover request' to the MSC. The MSC forwards this message to the new BSC (step 1.a).
- **Handover command** : The new BSC prepares its BTS for receiving the new MS (step 1.b). Then, the new BSC initiates the handover by transmitting the handover command message to the MS through the old BSC (step 2). This step permits the MS to locate the radio channel of the new BTS/BSC.

- **Handover bursts** : Upon receipt of the handover command message, the MS disconnects the old radio channels and initiates the establishment of lower layer connections in the new radio channels. In order to establish these connections the MS sends a handover burst message to the new BSC (step 3) and, when successful, the transmission suspended with the old BSC is re-established again between the MS and the new BSC through its BTS.
- **Handover complete** : Finally the MS sends the handover complete message to the old BSC through the new BSC (step 4). By receiving this message, the old BSC resumes all normal operations and releases the old radio channels on its BTS (step 5).

3 CommUnity

CommUnity [5] is a parallel program design language initially developed to show how programs fit into Goguen's categorical approach to General Systems Theory. The language and its framework have been extended to provide a formal platform for architectural design of open, reactive, and reconfigurable systems.

CommUnity designs are in the style of Unity programs [4], but they also combine elements from IP [7]. However, CommUnity has a richer coordination model and, even more important, it requires interaction between components to be made explicit. In this way, the coordination aspects of a system can be separated from the computational ones and externalized, making explicit the gross configuration of the system in terms of its components and the interactions between them. Each configuration can be transformed into a single, semantically equivalent, design that captures the behaviour of the whole system as it emerges from the components and their interconnections.

In order to model systems that are location-aware, a recent extension [8] of CommUnity was developed that adopts an explicit representation of the space within which movement takes place, but does not assume any specific notion of space. In this way, Computation, Coordination, and Distribution/Mobility are explicitly separated as architectural dimensions. The remaining of this section explains how this separation is achieved.

In CommUnity, every design is described in terms of a set of channels (declared as input, output or private) and a set of actions (shared or private).

- Input channels are used for reading data from the environment; the design has no control on the values received in such channels. Output and private channels are controlled locally by the design. Output channels allow the environment to read their values.
- Shared actions represent possible interactions between the design and the environment; private actions represent internal computations in the sense that their execution is uniquely under control of the design.

The computational aspects are described in CommUnity through the transformations operated by the actions over the channels. Each action is composed by a guard and a set of assignments over the private and output channels. A collection of data types is used for structuring the data that the channels transmit and for defining the operations that perform the computations required on the assignments. If an action has no assignments, we represent its body by the keyword *skip*. Figure 2 shows the design of the mobile station (cellular phone) for the handover case study. This design has, two input channels (*inMsg1, inMsg2*), two output channels (*outMsg1, outMsg2*), and two pairs of actions for communication (*send1* and *receive1*, *send2* and *receive2*). These channels and actions will be used to the communication with the current and the candidate BSCs. To reflect the fact that a sending should always precede a receipt we add a private channel (*sc*). This private channel is used in the guards of both actions (switching between true and false). The expression "X oneof T" assigns a random value of type "T" to channel "X".

In order to reflect the separation between computation and coordination, the definition of individual components of a system is completely separated from the definition of the interactions between these components. The model of interaction between designs is based on action synchronization and exchange of data through input and output channels. These are standard means of interconnecting software components. What distinguishes COMMUNITY from other parallel program design languages is the fact that such interactions between components have to be made explicit by providing the corresponding name bindings; no implicit interaction can be inferred from the use of the same name in different designs. We can witness three kinds of interaction between two components:

(i) A connection of an input channel of one design with an output channel of the other;

(ii) A connection of an input channel of one design with an input channel of the other;

(iii) Synchronization between one action from each design.

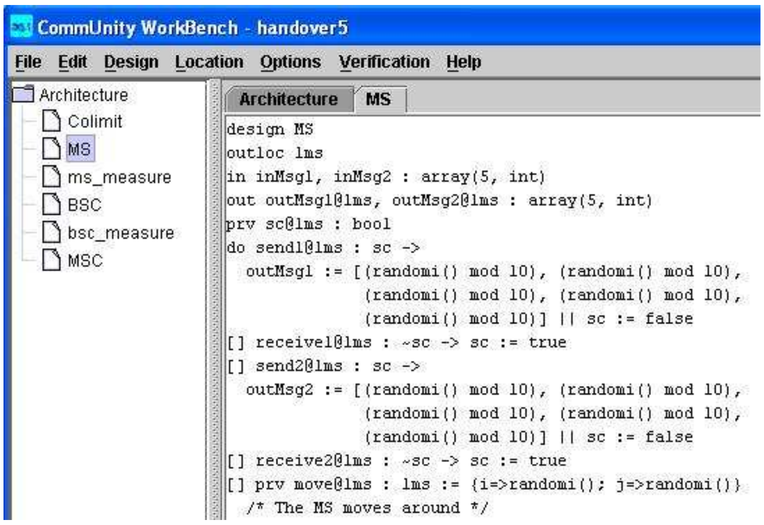
COMMUNITY was extended having in mind the need to provide support for the description of the distribution and mobility aspects of systems in a way that is completely separated from the computational and interaction aspects. In order to support this three-way separation of concerns, designs were extended to be location-aware by associating their "constituents"—private and output channels, and actions—with "containers" that can move to different positions. Hence the unit of mobility, that is the smallest constituent of a system that is allowed to move, is fine-grained.

More precisely, designs have a set of location variables over a special type `Loc` that describes the possible values that constitute the "space of mobility". The data sort `Loc` models the positions of the space in a way that is considered adequate for the particular application domain in which the system is or will be embedded. Each output channel, private channel, and action, is assigned to a set of location variables:

- Each local channel x is associated with a location variable l. We make this assignment explicit by writing $x@l$ in the declaration of x. The value of l indicates the current position of the space where the values of x are made available. A modification in the value of l entails the movement of x as well as of the other channels and actions located at l;

- Each action g is associated with a set of location variables 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 the parts of it in each of these locations.

Location variables can be declared as input or output in the same way as channels. Input location variables are read from the environment and cannot be modified by the design and, hence, the movement of any constituent located at an input location variable is under the control of the environment. Output location variables can only be modified locally through assignments performed within actions and, hence, the movement of any constituent located at an output location variable is under control of the design. In Figure 2 there is an output location variable *lms*, and the movement of the local channels and the actions are under control of the design. The private action *move* changes the value of the location variable.



```
design MS
outloc lms
in inMsg1, inMsg2 : array(5, int)
out outMsg1@lms, outMsg2@lms : array(5, int)
prv sc@lms : bool
do send1@lms : sc ->
  outMsg1 := [(randomi() mod 10), (randomi() mod 10),
              (randomi() mod 10), (randomi() mod 10),
              (randomi() mod 10)] || sc := false
[] receive1@lms : ~sc -> sc := true
[] send2@lms : sc ->
  outMsg2 := [(randomi() mod 10), (randomi() mod 10),
              (randomi() mod 10), (randomi() mod 10),
              (randomi() mod 10)] || sc := false
[] receive2@lms : ~sc -> sc := true
[] prv move@lms : lms := {i=>randomi(); j=>randomi()}
  /* The MS moves around */
```

Fig. 2. A Distributed Design

Figure 3 shows the details of an interaction, where input channels and input locations are depicted by inward triangles, output channels and output locations by outward triangles, and actions by circles. Note that only input/output channels and shared actions appear, because private channels and private actions are not used in interactions. In the Figure we show the interaction between the MS and ms_measure designs. For instance this interaction consists only to the connection between two location variables (respectively *lms* and *lm*) and it means that the ms_measure is always co-located with the MS. In Section 5.4 this is explained in more details.

Due to the introduction of the distribution/mobility dimension two new binary interactions emerge:

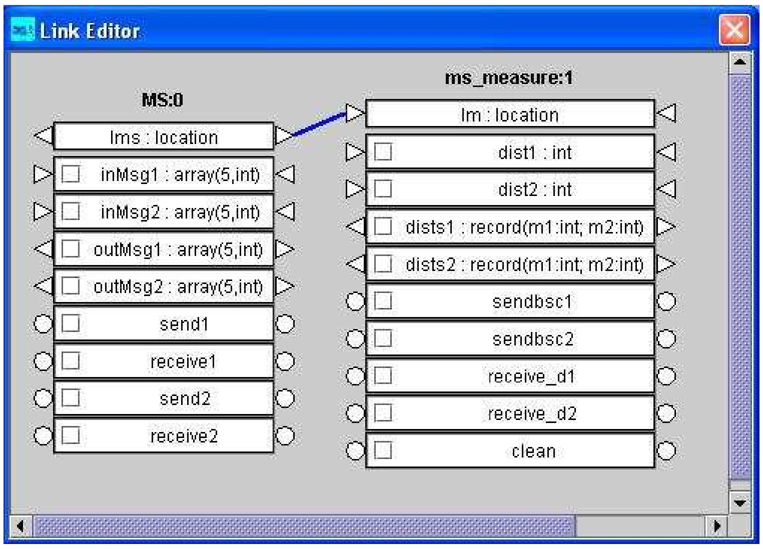


Fig. 3. Interactions

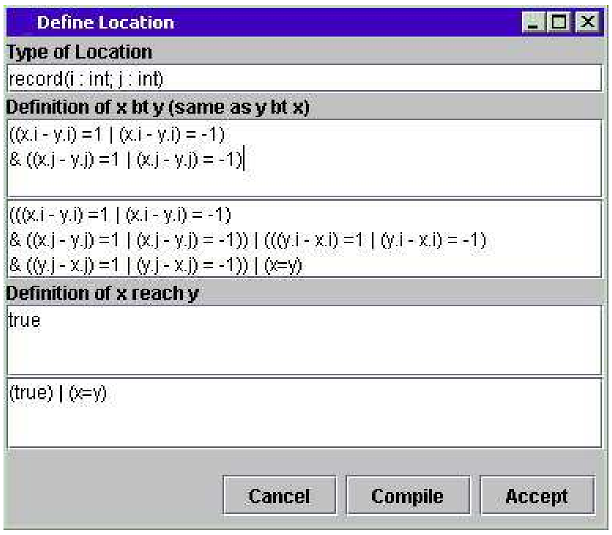
- A connection of an input location variable of one design with an output location variable of another;
- A connection of an input location variable of one design with an input location variable of another.

The relevant properties of the mobility space introduced upon are captured by two binary relations over the domain. First the relation `touch` defines that two positions in the space are "in touch" with each other. Coordination among components takes place only when all the locations of all the actions involved in a synchronization are "in touch". We assume that `touch` is reflexive and symmetric. The second relation `reach` means that one position is reachable from another, and is assumed to be reflexive. Movement to a new position is possible only when this position is reachable from the current one, i.e., a location variable can take a new value if this value is of a location reachable from the current position represented by the current value of the location variable. Figure 4 shows the definitions used for the case study: locations are represented by pairs of integers; two locations are in touch if they are adjacent; any location is reachable from any other. To define the relations we use mathematical expressions. For instance '|' is the disjunction and '&' the conjunction. Due to the relation 'in `touch`' being symmetric and reflexive the workbench rewrites the correspondent expression in the second part of the editor (Reflexivity for the relation 'Reach').

q a 2a parte é o fecho reflexivo (e simetrico) da definição do utilizador

The semantics of a system configuration can be obtained through the colimit of the system, which corresponds to computing the parallel composition of the processes and interactions involved. By internalizing all the interactions, the colimit delivers a design for the system as a whole. More generally, colimits in COMMUNITY are obtained as follows:

- Channels and locations involved in each i/o-communication established by the



```
record(i : int; j : int)
```

```
((x.i - y.i) =1 | (x.i - y.i) = -1)
& ((x.j - y.j) =1 | (x.j - y.j) = -1)
```

```
(((x.i - y.i) =1 | (x.i - y.i) = -1)
& ((x.j - y.j) =1 | (x.j - y.j) = -1)) | (((y.i - x.i) =1 | (y.i - x.i) = -1)
& ((y.j - x.j) =1 | (y.j - x.j) = -1)) | (x=y)
```

```
true
```

```
(true) | (x=y)
```

Fig. 4. Locations

configuration are amalgamated.

- Every set of actions that are synchronized is represented by a single action whose occurrence captures the joint execution of the actions in the set. The transformations performed by the joint action are distributed over the locations of the synchronized actions. Each located action is specified by the conjunction of the specifications of the local effects of each of the localized actions.

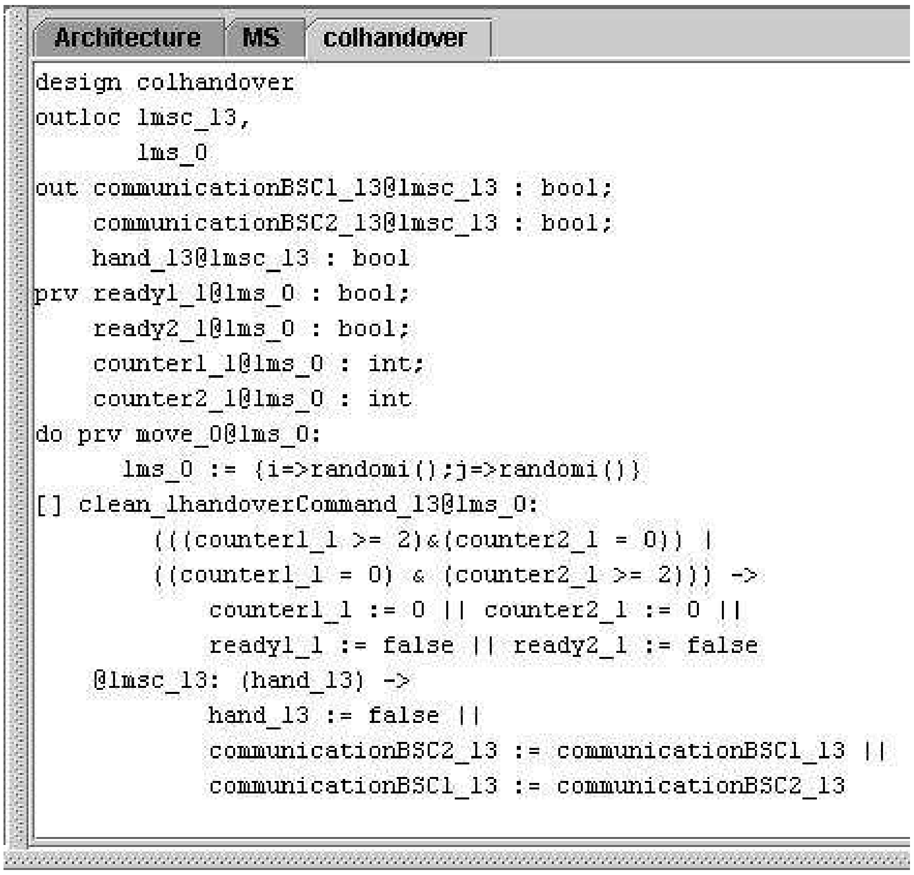
We show in Figure 5 part of the colimit generated for the case study (see Section 5). In particular, the figure shows an action *clean_1handoverCommand_13* distributed over two locations, given by location variables *lms_0* and *lmsc_13*.

The three following rules express the restrictions on the interactions that make an architecture a well-formed configuration and for which the existence of colimits can be ensured:

- An output channel or location of a component cannot be connected with output channels or locations of the same or other components (even indirectly);
- Private channels and private actions cannot be involved in the connections.
- Indirect synchronization of actions belonging to the same design is forbidden, because this entails their execution at the same time.

4 COMMUNITY Workbench

The COMMUNITY Workbench [13] implements directly over Java the constructions of coordination and distribution that give semantics to COMMUNITY as an architectural description language. Although COMMUNITY is independent of the actual data types used, the Workbench provides a fixed set of types: integer and real numbers, booleans, lists, arrays, records, and enumerations. The workbench also includes an export utility to save the whole architecture or just some connectors as a textual specification that can be easily read and understood without the



```
design colhandover
outloc lmsc_13,
       lms_0
out communicationBSC1_13@lmsc_13 : bool;
    communicationBSC2_13@lmsc_13 : bool;
    hand_13@lmsc_13 : bool
prv ready1_1@lms_0 : bool;
    ready2_1@lms_0 : bool;
    counter1_1@lms_0 : int;
    counter2_1@lms_0 : int
do prv move_0@lms_0:
      lms_0 := (i=>randomi();j=>randomi())
[] clean_1handoverCommand_13@lms_0:
        (((counter1_1 >= 2)&(counter2_1 = 0)) |
        ((counter1_1 = 0) & (counter2_1 >= 2))) ->
            counter1_1 := 0 || counter2_1 := 0 ||
            ready1_1 := false || ready2_1 := false
    @lmsc_13: (hand_13) ->
            hand_13 := false ||
            communicationBSC2_13 := communicationBSC1_13 ||
            communicationBSC1_13 := communicationBSC2_13
```

Fig. 5. Colimit

tool as will be explained in the next section. The workbench is available from the COMMUNITY web site (www.fiadeiro.org/jose/CommUnity).

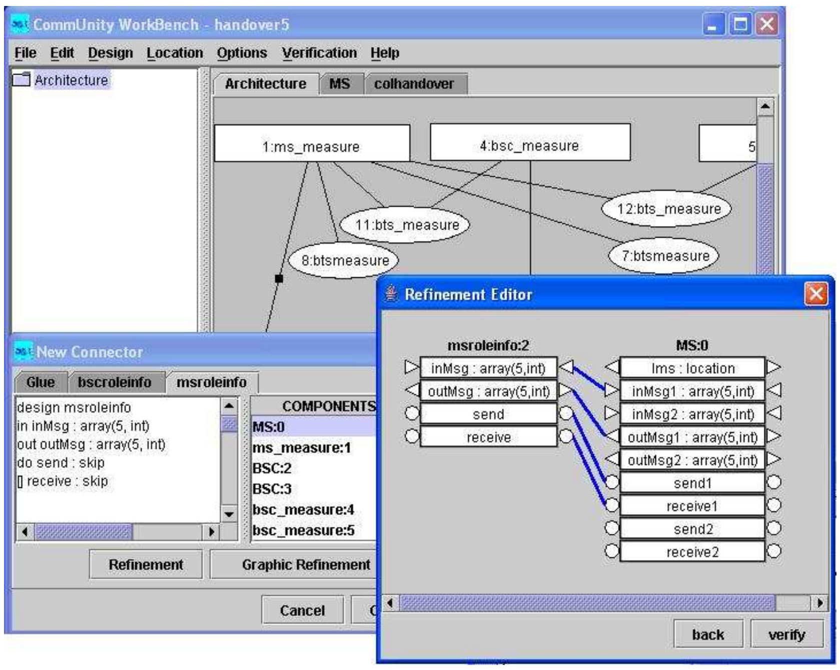
The current version of the tool (1.2) allows the user to:

(i) *Specify the* `Loc` *type and the relations* `touch` *and* `reach`*:* The location type is defined in terms of the pre-defined types (e.g., as a record of two integers to represent a 2 dimensional space). The relations are defined as boolean expressions with both parameters of location type.

(ii) *Write* COMMUNITY *designs:* The user may write new designs or edit existing ones.

(iii) *Define graphically architectural connectors:* In connector diagrams the user may do the same he can do in the architecture diagram (see step IV below), except for adding and deleting connectors. In the diagram of a connector, each node is uniquely numbered and must be an instance of an existing design. Connectors must have a star topology, the glue being in the center.

(iv) *Define graphically the architecture, with connectors, and calculate its colimit:* In the architecture, the user may add and delete nodes, arcs and connectors, drag nodes around to change the diagram layout, double-click on an arc or select some nodes to invoke a graphic link editor to visualize and/or to set the

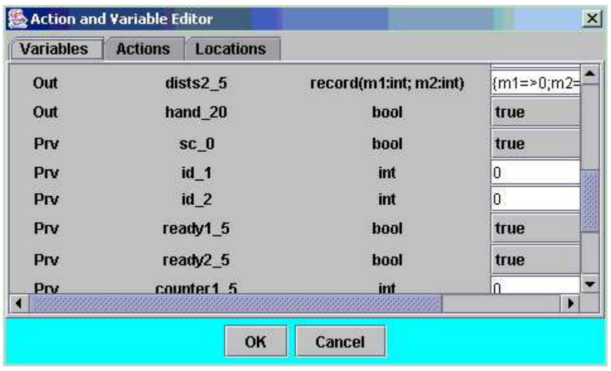


```
design msroleinfo
in inMsg : array(5, int)
out outMsg : array(5, int)
do send : skip
[] receive : skip
```

Fig. 6. Refinement in the Workbench

interactions between the selected nodes. The user can make direct connections between channels and locations as well as synchronizations between actions of different nodes. The editor presents to the designer all the actions, input and output channels of the selected nodes as was shown in Figure 3. The workbench detects invalid bindings—(in)direct sharing of output channels or location variables, or synchronization of actions of the same node. The user may add the connectors created in step III to the main architecture. This is done by explicitly defining which components refine which roles of the added connectors. Figure 6 shows how the instance of the MS design refines the msroleinfo role of the BTS connector (to be explained in Section 5), i.e., how the channels and actions of the role are mapped to those of the component. Notice that, whereas in the composition of components output channels are connected to input channels, in refinement output channels must be mapped to output channels. Components are represented by rectangles, connectors by oval shapes, and direct links by lines with small black squares.

(v) *Run a design (in particular the colimit of the configuration):* Before running a design, the user has to provide the initial values for all channels and location variables, and to choose which channels, location variables and actions should be traced. In Figure 7 we show the initialization window. By default, numbers are initialized with zero, booleans with false, and only output channels and location variables are traced. The trace gives great flexibility to test several scenarios. Shared actions can be explicitly selected for execution by the user or automatically by the tool, in fair or random mode. Private actions are always selected in a fair mode. There are also three choices for the value of any input channel that is not connected to an output channel: **1-**The same as



Out	dists2_5	record(m1:int; m2:int)	{m1=>0;m2=
Out	hand_20	bool	true
Prv	sc_0	bool	true
Prv	id_1	int	0
Prv	id_2	int	0
Prv	ready1_5	bool	true
Prv	ready2_5	bool	true
Prv	counter1_5	int	0

Fig. 7. Initialisation

in the previous execution step; **2-**A random value; **3-**Set by the user. During the trace the user can track the location variables: after each step, the current value of each location variable is shown, as well as the `touch` and `reach` relations between the location variables. We show in Figure 8 a possible state of the location variables when running the case-study. The black arrows show that the target location is reachable from the source location and the white lines show the locations are in touch.

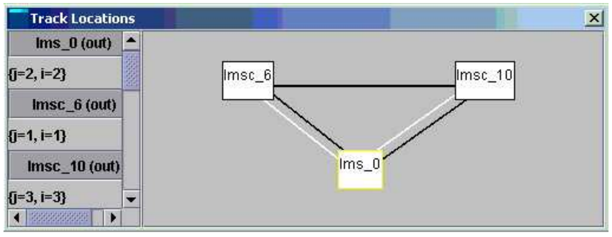


Fig. 8. Locations Track

CommUnity architectures and connectors described in the WB can be saved as textual specifications. The language that is used is like most Software Architecture approaches. We propose an Architecture Description Language to describe COMMUNITY architectures. This language is inspired mainly by Armani [11]. More precisely, the template of our architecture language is as shown in Figure 9. For illustration, the connectors presented in this paper are described using this notation (More precisely the subsection 5.3).

5 Handover in COMMUNITY

We will now produce the conceptual model, following the description of the handover protocol given in Section 2. The proposed development consists of five phases to be applied in the order shown in Figure 10:

(i) *Architectural entities* We operate a decomposition of the system in terms of designs and interactions (links and/or connectors). A black-box computational entity is represented as a single design, and a relationship between designs is

```
architecture name {
    loc = data type
    bt (x,y) = boolean expression over x and y
    reach (x,y) = boolean expression over x and y
    components
        design name_1
        design name_N
    connectors
        connector name {
            glue
                design name
            roles
                design roleName_1
                design roleName_K
            configuration
                instd_1, instd_L: designName
                attachments {
                    instd_I.idI - instd_J.idJ
                }
        }
    configuration
        instd_1, instd_M: designName
        instc_1, instc_P: connectorName
        attachments {
            instd_I.idI - instd_J.idJ
        }
        refinement instc_I.roleName to instd_J {
            idI to idJ
        }
}
```

Fig. 9. Architecture Specification Template

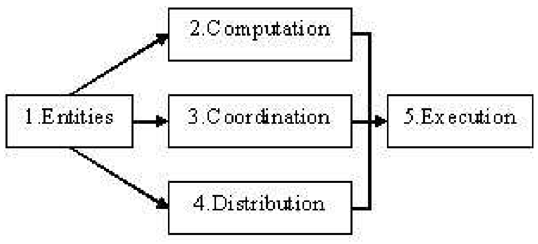


Fig. 10. Development

represented through an interaction.

(ii) *Computation* We describe in detail each identified design from the previous phase. That is, the private as well as the shared constituents (input/output/private channels and shared/private actions) are completely specified.

(iii) *Coordination* Each identified interaction has to be classified either as a simple link between two designs or a connector, which can includes some behaviour. In order to specify a connector the behaviour should be encapsulated in a glue, and a role has to be defined for each type of participant. In some cases it is necessary to go back to the computation phase based on intermediate results from this phase.

(iv) *Distribution* This phase starts by checking the possible topological distribution for each design and glue resulting from the previous phases. In case of required distribution, the corresponding design or glue has to be extended/adapted with appropriate location variables and actions dealing with them (e.g., to express mobility). Due to the new requirements some existent actions can also be distributed over locations. Finally, to handle the topological distribution of the real system, new connectors dealing only with locations can be created from scratch.

(v) *Execution* The architecture is executed with different initial values in order to test it. In the current version of the workbench, the execution of the specified architecture is centralized, in the sense that it is simulated by the execution of the generated colimit. In the future we will make available a distributed execution mode: the location values will be mapped to physical hosts and the physical mobility will be real (for more details see Section 5.5).

In the following we describe the handover process using COMMUNITY and the workbench, applying the development steps described above. For this description we assume a small system with only one MS, two BTS/BSC's and one MSC, as it is sufficient for reflecting all features of a typical handover. Moreover, in our system specification we assume that the MS is continuously communicating with a BSC, because if the MS is idle no handover would be required. The handover is triggered by the current BSC, and it initiates the process. In our model the relation `touch` plays an important role. That is, the MS moves around and it can only receive the measurements from both BTS's when it is in `touch` with them. Then each time the MS receives both measurements, it sends them to the current BSC, which verifies if it controls the BTS closest to the MS. If this is not the case it begins the handover process. Although the handover process can fail in a real system, we assume no failure for this model because our goal is to model the architecture, focuses on the separation of dimensions. The externalization of the measurements' process and the handover to connectors shows to be a first good step, because for example it permits us to detect where the model must explicitly evolve to include scalability. The evolution of the model to include failure is a possibility for future work.

5.1 Architecture entities

By analyzing the case study, we discover two different kinds of computation for each entity—namely communication and measurements, and we propose therefore to distinguish them in different elements in the architecture. Afterwards we named as layers the two different kinds of computation.

- **MS:** We describe the MS with two designs. The first one, denoted MS, includes the computation related to the communication. The second one, denoted ms_measure, encapsulates the computation reflecting the reception and sending of any measurements.
- **BSC:** Similarly, we capture the BSC using two designs, denoted by BSC and bsc_measure.
- **MSC:** We propose just one design, named MSC, to deal with the communication.
- **BTS:** Three connectors are associated with this entity, that we refer to as BTS, bts_measure, and btsmeasure. The distinction between the last two connectors will be described in the subsection 5.3.
- **Handover:** In order to externalize the handover process, we define this entity as an interaction represented by a connector called Handover.

Figure 11 depicts the organization of the different entities of the Handover architecture. The elements of the architecture are disposed in order to visualize the two layers referred before. Above are all the components and connectors related to the treatment of the measurements.

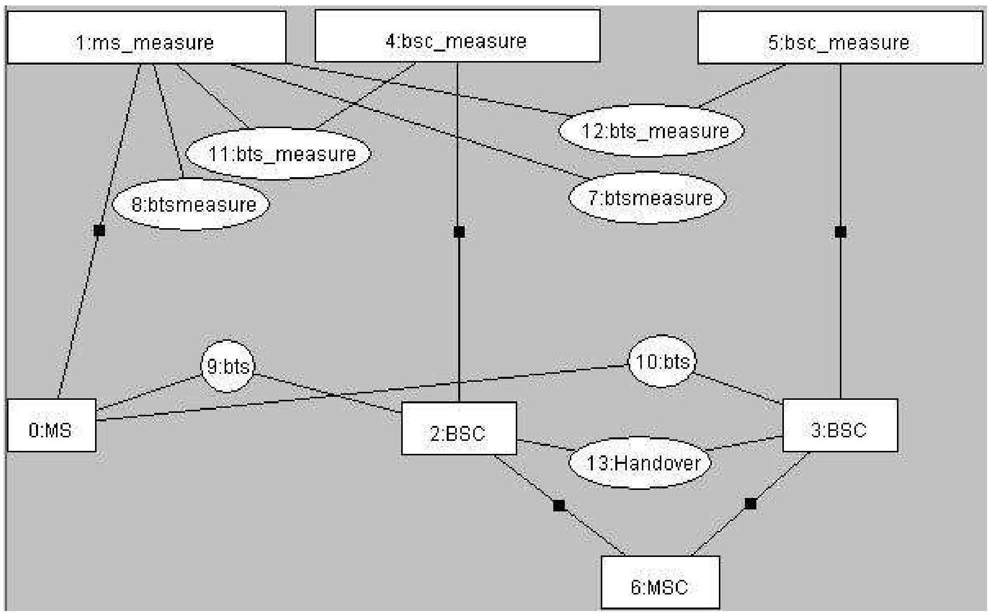


Fig. 11. The architecture in the Workbench

5.2 *Computation*

As stated before, this phase details each of the previously identified computational entities, the designs. We only show and discuss the details of the ms_measure and the bsc_measure designs. To describe each design, we refer to the syntax of COMMUNITY (see Section 3). The design MS is already explained in section 3 and shown in Figure 2.

ms_measure:

The ms_measure receives the measurements from the two BTS's through the connectors btsmeasure's, then sends them to the current BSC (respective bsc_measure). For this purpose, it uses one input channel (*dist1* and *dist2*) for each btsmeasure, and one output channel (*dist1dist2*) to make the measurements available. Like the MS design, for each of the two considered BSCs we have one action for sending the measurements (*sendbsc1* and *sendbsc2*) and another one for receiving them (*receive_d1* and *receive_d2*). Finally, for controlling the sequence of its actions, some private channels are required (the *ready* and *counter* channels). For executing the action *sendbsc1* or *sendbsc2* the private channels *ready1* and *ready2* must have the value true. This only happens if the ms_measure has received both measurements executing the actions *receive_d1* and *receive_d2*. But the ms_measure must avoid to receive a measurement twice, so the *counter*'s are used for that purpose. If action *receive_d1* (and the same for *receive_d2*) is executed twice before the other one, then action *clean* is executed and the *counter* and *ready* channels are re-initialized. This leads to the COMMUNITY design shown in Figure 12.

bsc_measure:

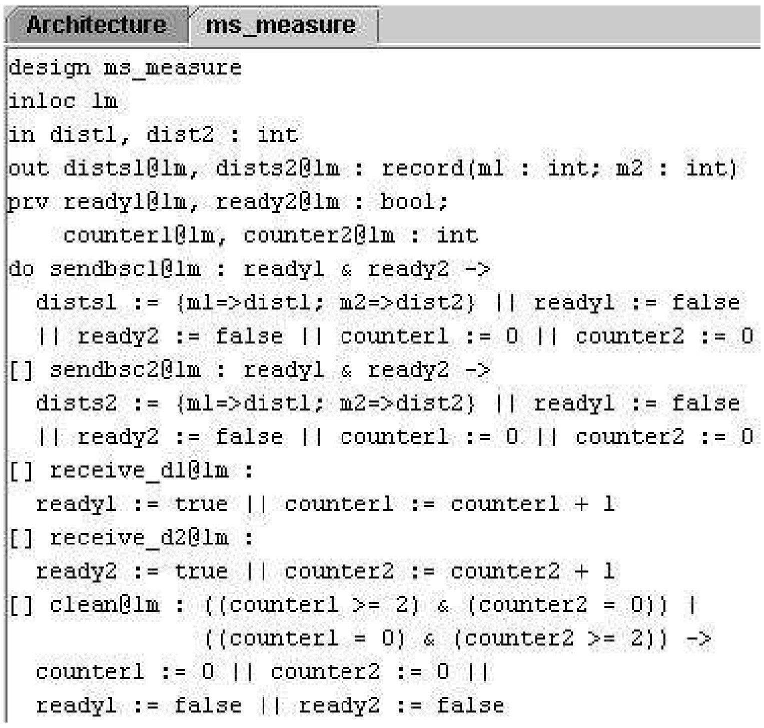
The bsc_measure receives the measurements from its corresponding BTS (trough the respective connector bts_measure), then verifies if the best measurement corresponds to its BTS. The result of the verification is send to the component BSC. For this purpose, it uses one input channel (*distdist2*) and one action (*receive_distdist2*) to receive the measurements. The result is made available with the output channel *newId*. As for the ms_measure this design results from the abstraction of the treatment of the measurements and the communication into different components. And finally the bsc_measure has an boolean input channel *communication* that is used as guard to the action which is used to receive the measurements. There is no sense to receive the measurements if this BSC is not the current one used on the communication.

5.3 Coordination

This phase details each of the interactions identified in Section 5.1. That is, for each connector, its glue, roles, and attachments are completely defined. We show and discuss the details of the BTS entity. The separation of the two layers is represented by different connectors.

BTS:

This is a binary connector relating the MS instance to a BSC instance. The glue part of this connector is a design with two input channels (*inMsg* and *outMsg*) for



```
design ms_measure
inloc lm
in dist1, dist2 : int
out dists1@lm, dists2@lm : record(m1 : int; m2 : int)
prv ready1@lm, ready2@lm : bool;
    counter1@lm, counter2@lm : int
do sendbsc1@lm : ready1 & ready2 ->
  dists1 := {m1=>dist1; m2=>dist2} || ready1 := false
  || ready2 := false || counter1 := 0 || counter2 := 0
[] sendbsc2@lm : ready1 & ready2 ->
  dists2 := {m1=>dist1; m2=>dist2} || ready1 := false
  || ready2 := false || counter1 := 0 || counter2 := 0
[] receive_d1@lm :
  ready1 := true || counter1 := counter1 + 1
[] receive_d2@lm :
  ready2 := true || counter2 := counter2 + 1
[] clean@lm : ((counter1 >= 2) & (counter2 = 0)) |
              ((counter1 = 0) & (counter2 >= 2)) ->
  counter1 := 0 || counter2 := 0 ||
  ready1 := false || ready2 := false
```

Fig. 12. The MS_Measure in the Workbench

the incoming messages and outgoing messages during the communication, and two actions for transferring the messages (*transferIn*, *transferOut*). The roles of this connector are very simple designs, each having two channels and two actions to be refined with component instances of the architecture (see below). After declaring one instance of the glue (btsI) and one instance for each role (msroleinfoI, bscroleinfoI), the attachment part defines the interactions of the glue channels and actions with those in both roles. For example, the output channel *outMsg* (resp. *inMsg*) has to be linked with the msroleinfoI instance through *inMsg* (resp. *outMsg*) and with the bscroleinfoI through *outMsg* (resp. *inMsg*). The same linking is to be applied between the transfer, send, and receive actions.

```
connector BTS {
    glue
        design bts
        in inMsg, outMsg : array(5, int)
        do transferIn : skip
        [] transferOut : skip
    roles
        design msroleinfo
        in inMsg : array(5, int)
        out outMsg : array(5, int)
        do send : skip
        [] receive : skip
        design bscroleinfo
        in inMsg, outMsg : array(5, int)
        do transferIn : skip
        [] transferOut : skip
    configuration
        btsI: bts
        msroleinfoI: msroleinfo
        bscroleinfoI: bscroleinfo
    attachments {
        btsI.outMsg - msroleinfoI.inMsg
        btsI.inMsg - msroleinfoI.outMsg
        btsI.transferOut - msroleinfoI.send
        btsI.transferIn - msroleinfoI.receive
        btsI.inMsg - bscroleinfoI.inMsg
        btsI.outMsg - bscroleinfoI.outMsg
        btsI.transferIn - bscroleinfoI.transferIn
        btsI.transferOut - bscroleinfoI.transferOut
    }
}
```

In order to use this connector with the MS and BSC designs, the role msroleinfoI and the role bscroleinfoI are refined with a MS design instance and a BSC design instance, respectively. Part of the refinement is shown in Figure 6.

```
refinement bts.msroleinfoI to MS {
    inMsg to inMsg1
    outMsg to outMsg1
    send to send1
    receive to receive1
}
refinement bts.bscroleinfoI to BSC {
    inMsg to inMsg
    outMsg to outMsg
    transferIn to transferIn
    transferOut to transferOut
}
```

Handover:

This connector externalizes all the handover process. It has three actions related to the handover process. The actions *handoverReqBSC1* and *handoverReqBSC2* initiate the handover. The action *handoverCommand* can be selected after the execution of one of the previous actions, because the output channel *hand* used in the guard remains true. This action is the handover core of our model, because it makes an exchange of values between the two BSC's through the two output channels *communicationBSC1* and *communicationBSC2*.

5.4 Distribution

As the topological distribution is obviously crucial in this application, any attempt to abstract it away would not be realistic. Following our development steps, on the basis of the resulting specification, we propose to enrich it with the required distribution features. We consider as illustration two designs and two connectors, namely the 'Mobile' MS, and the 'Location-aware' ms_measure, bts_measure and btsmeasure. We recall that Figure 4 shows the definition of type `Loc` and its binary relations adopted for this case study.

Mobile MS:

This design has already been shown in Figure 2. All its constituents, channels and actions move together around in a random way.

ms_measure:

Due to the fact that ms_measure is always co-located with MS, this design has an input location variable (*lm*) that is connected to the output location variable of the MS, as shown in Figure 3. Moreover, all its channels and actions are at location *lm*, in the same way as for the MS.

btsmeasure:

This unary connector sends to ms_measure the measurement based on the location of its glue and the location it receives from the ms_measure (which is the same as the MS location), but only when their location variables are "in `touch` ". The glue of the connector is composed of: an output location variable holding its location, an input location variable for reading the current location of the ms_measure when in `touch`, and an action with an adequate output channel for sending the computed distance to the ms_measure.

bts_measure and Handover:

The connector bts_measure has an input location variable in its glue, that must be initialized with the value of the output location variable of the glue of the last connector, because they are related to the same BTS entity. When the MS is "in touch" with the last connector, it must be in touch with the related one. The handover connector has one output location variable and all the constituents are located in the respective location variable.

5.5 Execution

In the current version of the workbench, the execution phase is centralized, in the sense that it is simulated by the execution of the generated colimit. Figure 7 shows the editor where the channels and locations can be initialized before the user can test the desired scenario. In the same editor he can choose which channels he wants to see evolving during the execution steps. While the colimit's execution is traced, the state of the system after each execution step can be analyzed with the current values of the channels.

We now present a summary of the architecture for distributed execution that we are currently implementing in the Workbench, based on the Klava package [2]. This library for mobile agent systems based on tuple spaces contains some facilities for specifying hosts, and logical and physical locations. The communication between hosts is encapsulated in a sub-layer, and because of that, we can abstract from the communication implementation layer. The Klava package has support for nodes that are located on the hosts and support for execution of processes on the nodes.

The specification of the architecture is distributed over a set of hosts, where we give an important role to one host, called the global host, which has the centralized knowledge of the architecture in COMMUNITY.

We define a mapping from the location values of the location types to the existent hosts. The `touch` and `reach` relations are implemented directly in Java over the location type.

The global host stores the connections (shared channels and synchronized actions) of the COMMUNITY architecture, the values of the location variables, and all channels that are not located at any location variable. The global host also stores the input channels that are not connected to any output channel. Moreover, it has a process, "globalInterpreter", runnning. This process knows all the COMMUNITY architecture and uses a fair algorithm for deciding which actions it will try to execute at each step.

We assume a fixed number of hosts. Each host stores all channels and actions located in location variables whose values map to that host.

All hosts (including the global one) have a running process denoted "localInterpreter". This process is in a loop, waiting for messages from the "globalInterpreter" which ask for values of local channels, for the verification of guards of the local actions, or for the execution of local actions. We are implementing the necessary sequential steps for the communication between the global and the other hosts, and we use the facilities of Klava of mobility of tuples between hosts for moving the channels and actions during the execution of the system.

6 Concluding Remarks

In this paper, we have illustrated how COMMUNITY supports architectural modelling of applications that need to be location aware. The semantic and syntactic primitives of COMMUNITY were shown to be expressive enough to model mobile systems as we know them in reality. The Workbench proved useful in detecting some errors in our initial, hand-written models. In particular, the trace facilities and the graphical depiction of the co-located channels and actions, and of the current "in touch" relationships, were found to be quite useful to animate the model and to find deadlocks. In summary, we feel that COMMUNITY and the Workbench are useful to anyone wishing to define and animate executable architectural models of distributed and mobile systems at a higher level of abstraction than other formal approaches based on process calculi.

Currently we can only simulate the execution of the generated colimit. But soon we expect to test, with the case-study model, the distributed execution implementation, in order to make the approach scalable.

Acknowledgements

This research was partially supported through the IST-2001-32747 Project AGILE - Architectures for Mobility. The work of Cristóvão Oliveira was supported by Fundação para a Ciência e Tecnologia through the PhD Scholarship SFRH/BD/6241/2001. We thank José Aires, Mário Costa, Brian Plüss, and Frederico Palma for having implemented several parts of the Workbench.

References

[1] R. Allen and D. Garlan. A formal basis for architectural connection. *ACM TOSEM*, 6(3):213–249, July 1997.

[2] L. Bettini, R. D. Nicola, and R. Pugliese. Klava: a Java framework for distributed and mobile applications. *Software–Practice and Experience*, 2002.

[3] C. Canal, E. Pimentel, and J. M. Troya. Specification and refinement of dynamic software architectures. In *Software Architecture*, pages 107–125. Kluwer Academic Publishers, 1999.

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

[5] J. L. Fiadeiro. *Categories for Software Engineering*. Springer-Verlag, 2004.

[6] J. L. Fiadeiro, A. Lopes, and M. Wermelinger. A mathematical semantics for architectural connectors. *Generic Programming*, LNCS Springer(2793):190–234, 2003.

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

[8] A. Lopes, J. Fiadeiro, and M. Wermelinger. Architecural primitives for distribution and mobility. In *Proc. 10th Symposium on the Foundations of Software Engineering*, pages 41–50. ACM Press, 2002.

[9] A. Lopes, M. Wermelinger, and J. Fiadeiro. Higher-order architectural connectors. *ACM Trans. on Software Eng. Methodology*, 12(1):64–104, Jan. 2003.

[10] J. Magee, N. Dulay, S. Eisenbach, and J. Kramer. Specifying distributed software architectures. In *Proceedings of the Fifth European Software Engineering Conference*, Barcelona, Sept. 1995.

[11] R. T. Monroe. Capturing software architecture design expertise with Armani. Technical Report CMU-CS-98-163, School of Computer Science, Carnegie Mellon University, Oct. 1998.

[12] C. Oliveira. GSM system - handover. Appendix I of Deliverable 4.2 of the AGILE project, 2003.

[13] C. Oliveira and M. Wermelinger. The community workbench. In *ICSE '04: Proceedings of the 26th International Conference on Software Engineering*, pages 709–710. IEEE Computer Society, 2004.

[14] C. Oliveira, M. Wermelinger, J. Fiadeiro, and A. Lopes. An architectural approach to mobility - the handover case study. In *Proc. of the 4th Working IEEE/IFIP Conference on Software Architecture*, pages 305–308. IEEE Computer Society Press, 2004.

[15] F. Orava and J. Parrow. An algebraic verification of a mobile network. In *Protocol Specification, Testing, and Verification X*, pages 275–291, Uppsala University, Department of Computer Systems, North-Holland, 1990.

[16] M. Wermelinger and J. L. Fiadeiro. A graph transformation approach to run-time software architecture reconfiguration. *Science of Computer Programming*, 44:133–155, 2002.

[17] *GSM System Overview*, volume rev. no. 100. APIS Technical Training AB, Sweden, apis training & seminars edition, 1998. training@apis.se.