# A Model-Driven Approach to Extract Views from an Architecture Description Language*

Cristóvão Oliveira
Department of Computer Science
University of Leicester
University Road
Leicester LE1 7RH, UK
co49@le.ac.uk

Michel Wermelinger
Computing Department
The Open University
Walton Hall
Milton Keynes MK7 6AA, UK
m.a.wermelinger@open.ac.uk

## Abstract

*A common approach to defining architectural views is to have independent heterogeneous representations that are tailored to each view's purpose, but this makes reconciling views into an overall architectural description harder. In this paper we put forward a complementary (not alternative) approach in which some views are derived from a given architecture description language (ADL) in a systematic way, by listing the design questions each view should answer. The approach is based on constructing the language's metamodel and extending it with the entities and associations needed to capture and explicitly relate the required views.*

## 1. Introduction

The need for multiple views, in order to have a richer description of a software system that helps capturing various design decisions and analyse their impact, has been recognized since the early days of Software Architecture research [23]. Common to many approaches, including the IEEE standard on architectural description [1], is the desire to allow for heterogeneous representations of mutually crosscutting perspectives that are relevant to different stakeholders. A recurrent problem is that different views might be inconsistent with each other, in the sense that, when put together, the views do not obey the constraints for a well-formed architectural description. Moreover, available notations to represent views, like the various UML diagrams, might not be ideally suited to describe architectures [18].

In this paper we put forward an approach that takes as starting point a language that is suitable for architectural description (i.e., an ADL) and extracts some views from it. The approach is model-driven in the sense that it is based on constructing a metamodel for the given language and then enriching it with views that aggregate the various architectural concepts embodied in the language. The approach is complementary to existing ones because the only views that can be derived are those encompassing concepts included in the given language. However, an ADL includes core concepts related to architectural design (like structure, behaviour, and interaction), and as such the set of views derived from the ADL is useful in spite of being restricted. A further restriction of this approach is that it is likely to only generate views that are relevant to those stakeholders that would use an ADL, most often designers or software architects. We say "likely" because it may happen that an ADL is sufficiently rich for views that are relevant for other stakeholders to be generated, like the implementation and deployment views. But on the other hand, the use of a single metamodel to explicitly represent the original language's and the derived views' concepts has the advantage of being a homogenous approach that explicitly relates the various views: this makes it easier to check for consistency between and within views and to combine views.

Although in principle any metamodeling approach can be envisaged, the concrete approach suggested in this paper is to use UML's (meta-)class diagrams and Object Constraint Language (OCL) to describe the architectural entities and their relationships. The reason for this choice, besides UML being a de facto standard metamodeling language through the Meta-Object Facility (MOF), is that it eases a future possible integration of this approach within a wider model-driven round-trip engineering framework, where one can envisage the views generated from the overall architectural description being carried over (through model-driven transformation) to corresponding views at various levels of detail (e.g., platform-dependent design and implemen-

*This research was supported by Fundacão para a Ciência e Tecnologia through the PhD Scholarship SFRH/BD/6241/2001 of Cristóvão Oliveira.

Proceedings of the Working IEEE/IFIP
Conference on Software Architecture (WICSA'07)
0-7695-2744-2/07 $20.00 © 2007 IEEE

tation). The approach consists of the following steps:

1. Construct the model for the given language, specifying consistency invariants in OCL.
2. List the questions that are possible, but not easy, to answer from an architectural specification, and determine the views that can provide immediate answers to those questions.
3. Extend the metamodel with extra classifiers and associations for each view, aggregating the architectural concepts relevant for each view and defining further consistency invariants for each view.
4. Combine views by adding new associations that enable further navigations between architectural entities.

Note that step 2 can be seen as a kind of "use cases" or "scenarios", not of the system being designed, but of the design process itself, i.e., of the ways the architectural specification is explored to answer specific design questions, like "is the system's functionality centralised?".

The following sections illustrate in more detail the above steps, by applying the approach to COMMUNITY, the language we have been using to support our research on the foundations of software architecture [8]. Section 2 introduces COMMUNITY and its metamodel (step 1), section 3 provides the rationale for the views (step 2), and sections 4 and 5 describe two derived views (steps 3 and 4). Due to space restrictions, and because the aim is just to illustrate the advocated approach, we only provide a simplified metamodel of COMMUNITY, we omit a third view we have derived, and we show only a few OCL constraints. Finally we provide a context for our work, relating it to other approaches to architectural views, and we present some conclusions.

## 2. CommUnity

COMMUNITY provides, like Darwin [17], Wright [3], or LEDA [4], 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 [7], 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.

In addition, COMMUNITY describes component behaviour with a parallel program design language that is similar to Unity [5] and IP [9] 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, and requires interactions between components to be made explicit. Moreover COMMUNITY 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.

Like most architectural approaches, COMMUNITY enforces a strict separation between computation and coordination, whereby interactions between components are specified just through their interfaces. The approach has been extended with a small set of semantic concepts and syntactical constructs in order to include a distribution dimension [15]. COMMUNITY requires the designer to provide an explicit representation of the space within which movement takes place, and as such the formalism is independent of any specific notion of space. The GSM-Handover protocol [22] has been modeled with COMMUNITY to illustrate the distribution extension. Fig. 1 shows the class diagram of a slightly simplified metamodel of COMMUNITY; the metamodel is completed with OCL constraints.

In addition to the language and mathematical semantics, we have been developing a workbench [21] that serves as a proof of concept of the formal framework. The workbench provides a graphical integrated development environment to write, run, debug components and draw configurations of components and connectors. We are currently starting the extension of the workbench to provide support for the three views we defined for COMMUNITY (Section 3).

### 2.1. Computation

A computational unit of a COMMUNITY system is given by a so called *design*, which is defined in terms of input and output location variables, input, output and private channels and a set of private and shared actions. Location variables hold the positions where data (channels) are stored and where code (actions) is executed. Input channels are used for reading data from the environment; the component has no control on the values that are made available in such channels. Output and private channels are controlled locally by the component. Output channels allow the environment to read data produced by the component. Private actions represent internal computations in the sense that their execution is uniquely under the control of the component whereas shared actions represent possible interactions between the component and the environment.

Channels are typed, and although the COMMUNITY semantics is independent of any type definitions, we have implemented a small set of types in the workbench (integers, booleans, lists, etc.). However, the metamodel remains generic, requiring a type for each channel (class *Type* in Fig. 1) and a type to represent locations ($LocationType$).

The computational dimension resides in the bodies of actions: each action has one or more bodies, each body being

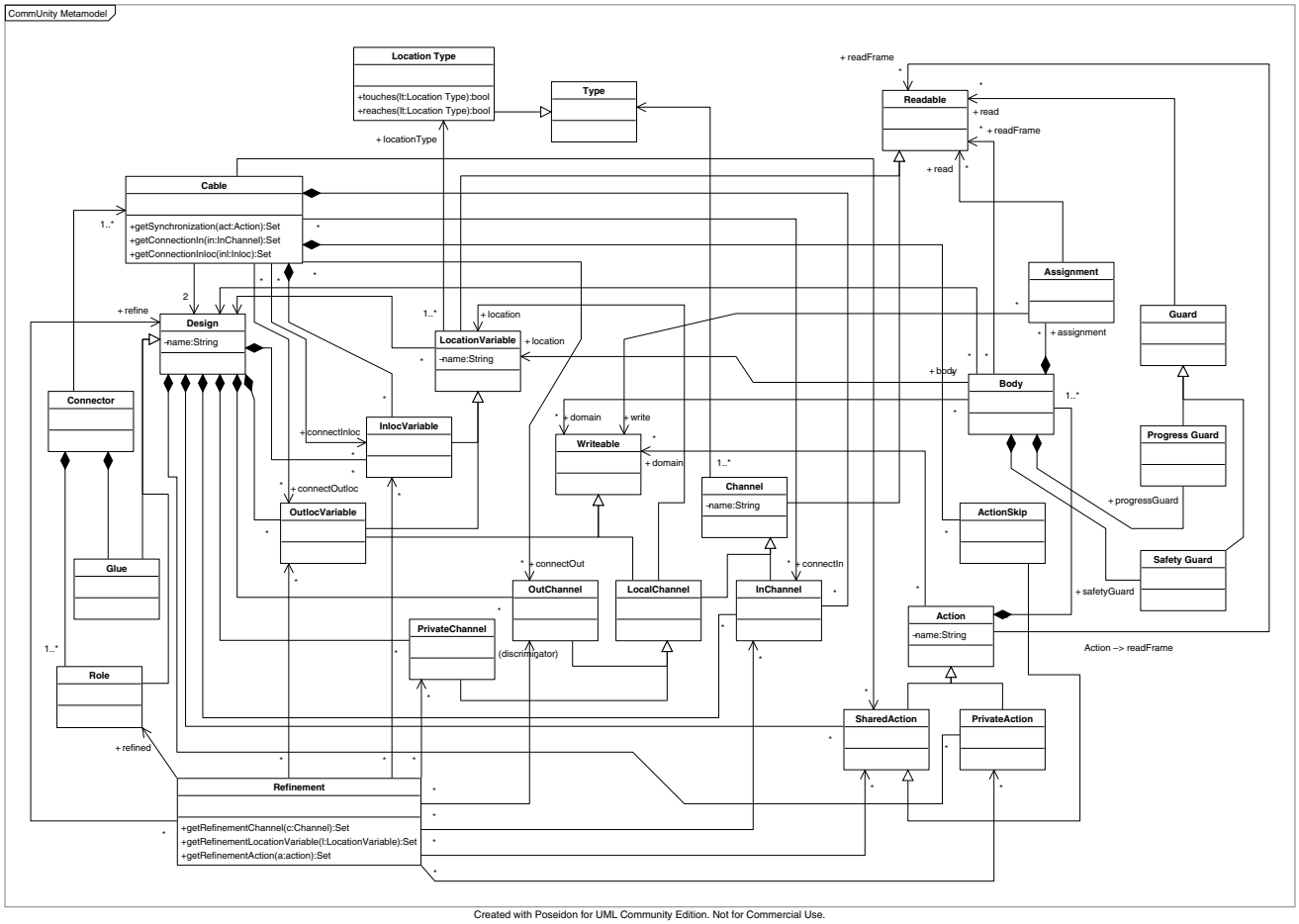


**Figure 1.** COMMUNITY **metamodel**

a guarded set of assignments. Multiple bodies allow for the distributed execution of an action (see next subsection). Each body has two guards (shown in Fig. 1), but for the purposes of this paper it is not relevant to distinguish them, and hence our example assumes they are always represented by the same boolean expression. At each step, one of the actions is non-deterministically selected and its assignments are executed in parallel if the guards of all its bodies evaluate to true. For a more precise definition of the operational semantics see [8]. The class *ActionSkip* in Fig. 1 represents an action, without any bodies, to be used for binding with other components' actions.

In order to illustrate the approach we use an example adapted from [16]: consider an airport luggage delivery system in which carts move along a track and stop at designated locations for handling luggage. The design *located_cart* models a cart able to move and handle luggage along the track; the cart is also monitored to count the number of times it handles luggage. The cart does not control its movement.

The cart (shown in Fig. 2) moves while it is not busy handling luggage and while its current position is different from its destination. However, the *move* action doesn't change the current position because it is an input location under control of the environment. Handling luggage takes place only when the cart docks at the destination station, which is available locally in *dest* and is recomputed when the cart undocks from the station. Note that the cart keeps track of the luggage stations it passed through in the private channel *path*. 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 location variable with the value of the new location to where the cart will move.

## 2.2. Distribution

Distribution is represented by the location variables in the designs. Each local channel "x" is associated with a lo-

```
design located_cart
inloc pos, pox, next
outloc dest
prv busy@pos : bool; path@pos : list(int)
do move@pos : ∼busy & (pos /= dest) → skip
[] dock@pos : ∼busy → busy := true
     @dest : pos = dest→ path := path : <dest>
[] undock@pos : busy & (pos = dest) →
        busy := false || dest := next
```

**Figure 2. Design located_cart**

cation variable "l". We make this explicit by simply writing "x@l" in the declaration of "x". The intuition is that the value of variable "l" indicates the current position of "x". Note that the location of input channels is unknown, because they are provided by the environment. Each action body is also associated with a location variable; hence, the execution of an action is distributed over the locations of its bodies. In other words, the execution of the action consists of the synchronous execution of a guarded command in each of those locations. The granularity of distribution is very fine (i.e. at the level of individual channels and commands) in order to provide maximum flexibility in the specification of the distribution of data and code. A modification in the value of a location variable "l" implies the movement of all channels and action bodies located at "l".

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

In Fig. 1 actions and channels are associated to their locations. This part of the metamodel also shows that the domain of an action is the set of the local channels and output location variables assigned to in the action's bodies. For the example, almost all the cart's data and code are located at the cart's current position (*pos* in Fig. 2).

For illustration the execution of the action *dock* is distributed: locally, i.e. at location *pos*, it tests if the private channel *busy* is false; at the position indicated by *dest*, it further stores the current destination in *path*. The unused location variable *pox* is a mistake that will be detected in the distribution view.

The cart moves because its position (available in variable *pos*) is incremented by the environment, more precisely by the *step_controller*, while the cart has not reached its destination (available in *dest*). How this is achieved is explained in the next subsection. The *step_controller* design is:

```
design step_controller
outloc theirs
do control@theirs : true → theirs := theirs + 1
```

Although no specific notion of space is assumed (i.e., the type used to represent locations is user-defined), there is always a special location, called $\lambda$, that represents a "central" location. The metamodel hence assumes that every local channel and each action body has a location, but in the concrete syntax of COMMUNITY we simply write "x" instead of "x@$\lambda$". The relevant topological properties of the space are captured by two binary relations over locations. The relation `touch` defines the pairs of positions that 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, symmetric and that all locations are in touch with $\lambda$. The second relation `reach` defines when one position is reachable from another, and is assumed to be reflexive. Movement to a new position is possible only when that position is reachable from the current one. Reachability from or to $\lambda$ is always valid.

## 2.3. Coordination

The coordination dimension is made explicit by connecting output and input channels and location variables, and by synchronizing non-private actions. Bindings have to be explicit (i.e., cannot be based on implicit naming) because all names in COMMUNITY are local, i.e., their scope is the design in which they occur. This makes it easier to reuse designs in various contexts. Due to the technicalities of the categorical semantics of COMMUNITY, bindings cannot be established directly between components' actions, channels and location variables, and instead have to be established through so-called *cables* (see [8] for details). A cable connects two designs and is itself a restricted form of design, just with input location variables, input channels and shared actions without bodies, because cables do not introduce behaviour. In the metamodel we require three operations on cables (represented by class $Cable$): one to obtain all component actions synchronised through a given cable action, another to obtain all component channels shared through a given cable input channel, and a third one to obtain all component location variables shared through a given cable input location variable. These operations are necessary to specify the consistency constraints on cables (constraints over location variables are similar to those over channels). For example the following constraint prevents the connection of two channels of the same design with the same channel of the cable. Moreover, it verifies the two channels belong to the two different designs involved in the cable, and finally it prevents the sharing of two output channels:

```
context Cable
```

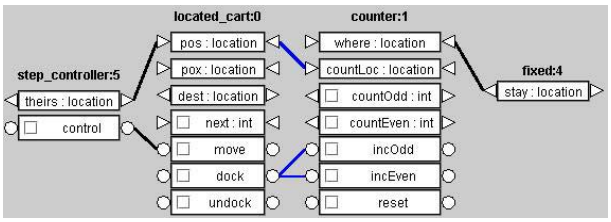


**Figure 3. Interactions**

```
inv ConnectionChannels:
 self.inChannel->forAll(elem |
  self.getConnectionIn(elem)->size() = 2 and
  self.getConnectionIn(elem)->forAll(elem1 |
   self.getConnectionIn(elem)->forAll(elem2 |
    elem1=elem2 or
    (elem1.design <> elem2.design and
     self.design->exists(elem1.design) and
     self.design->exists(elem2.design) and
     not(elem1.isInstanceOf(OutChannel) and
         elem2.isInstanceOf(OutChannel))))))
```

Continuing the example, Fig. 3 taken from the workbench shows the connections between *located_cart* and *counter*, where input channels and location variables are depicted by inward triangles, output channels and location variables by outward triangles, and actions by circles. Note that private constituents are not used in interactions and hence do not appear. Although the formal semantics of COMMUNITY (and hence the metamodel) requires cables, the workbench's GUI uses bindings directly between components for simplification. Design *counter* (which we omit for lack of space) simulates the tracking of the handling of luggage. We assume there are two kinds of locations for handling the luggage, called odd and even locations, with a counter for each kind, to be incremented when the cart is handling luggage at such a location. To illustrate the possibilities of COMMUNITY, action *dock* is synchronized with two actions *incOdd* and *incEven*, so that for each execution of *dock* either of those actions is executed and hence *countOdd* or *countEven* is incremented. The choice is based on the omitted guards of *incOdd* and *incEven*.

To specify that two components are co-located, we just have to share their location variables: in Fig. 3, we bind *pos* of *located_cart* with *countLoc* of *counter* to "include" the counter in the cart. To provide a value for an input location variable, it must be shared with an output location variable: in our example, the cart's position is provided by variable *theirs* of the *step_controller*. As we saw before, the action *move* of the design *located_cart* has an empty body. The cart moves when this action is executed, because it is synchronized with the action *control* of *step_controller*, and the movement is achieved when the action *control* makes the assignment to the output location variable *theirs* which is connected with *pos* of *located_cart*.

Complex interaction behaviour between components has to be defined using connectors. As shown in Fig. 1 connectors are made of a glue and a set of roles, all given by designs. The glue is connected to each role through a cable. The glue specifies the interaction behaviour, while roles serve as a kind of formal parameters, restricting the components to which the connector can be applied.

The application of a connector, in the construction of an architecture, consists in instantiating (formally: refining) each role with a specific component of the architecture (see class $Refinement$ in the metamodel). Each channel of the role has to be refined by a channel of the same type and kind (i.e., input, output or private) of the component, and similarly for location variables. But COMMUNITY allows an action of the role to be refined into a set of component actions that break a computation step into multiple sub-steps. We show here only one OCL constraint, for input channels, whereby the getter implements the mapping between the role's and the component's channels:

```
context Refinement
 inv RefiningInChannels:
  role.inChannel->forAll(elem |
   let c : Channel = self.getRefinementChannel(elem)
   in c.isInstanceOf(InChannel) and
    c.type = elem.type)
```

In the example, the previously shown design *step_controller* is the glue of a unary connector, the role of which is refined by *located_cart* and Fig. 3 shows the binding between the glue and the component once the connector has been applied.

## 3. Architectural Views

An architectural view describes a part of a system according to some perspective of interest, helping the designer to validate the architecture. Each view highlights and makes explicit some aspects, while omitting others.

In the setting of COMMUNITY, the views should be defined in such a way as to make it easy for the designer to answer questions like:

1. Which actions *cannot* occur simultaneously and which ones *must* occur together?
2. Which channels are actually the same (i.e., shared)?
3. Which channels are used by which actions?
4. Which actions from distinct components use a common channel? Are those actions synchronized?
5. Which constituents (channels or actions) are always co-located?
6. Which locations might have to be in touch with each other?
7. Which locations (more precisely channels and actions located at those locations) are controlled by which actions?

Such checks will help the designer to validate the architecture, detect potential problems (e.g., a design with too many constituents at the same location), and take corrective actions (e.g., by relocating some of the constituents). Questions like the above focus on actions, channels and locations and can therefore be answered by switching between three basic views: computation, coordination, distribution. For example, the coordination view might help the designer spot a large set of actions that always execute synchronously, whereas the distribution view shows if they are scattered among many locations or not. But this switching can be in part avoided by combining the views, meaning that, being in a given view it is possible to obtain adapted version(s) of the other(s) and combine them. The combination of $X$ with $Y$ is therefore the enriching of view $X$ with information from view $Y$. However, some combinations are not considered in this paper as we explain next.

The combination of the computation view with the coordination view is already subsumed by the architecture, which is centered on the designs (computation view) combined with the interactions between designs (coordination view). The combination of the computation view with the distribution view doesn't add new insights either, because in the architecture each design has its channels (computation view) mapped to a location (distribution view) and each action is explicitly distributed or not over several locations. Finally, the combination of coordination with distribution is not presented because combining them the other way round is more clear and succinct. Due to space restrictions we omit the simplest view (computation), but we refer to its information in the other views in order to maintain coherence.

Each one of the three view types is defined by a metamodel, which is obtained from the architectural metamodel by adding the necessary new entities and associations. The view's metamodel must also show (through the class diagram and OCL expressions) how the new entities are related to those of the architecture, e.g. how co-located actions dispersed throughout various designs in the architecture become aggregated in a single location in the distribution view. The aggregation entities of each view will be called *units*, to use a neutral term that doesn't clash with component or design.

There is one single diagram including all entities necessary for modeling an architecture and its views, and the associations between those entities and the additional OCL constraints can be seen as a declarative specification of the model transformation that is necessary to obtain a view from the architecture.

The views for our example will be given by very informal graphs (Fig. 5 and 7), because using the correct UML notation, with one object for each channel, location variable, action, etc., is infeasible due to space restrictions.

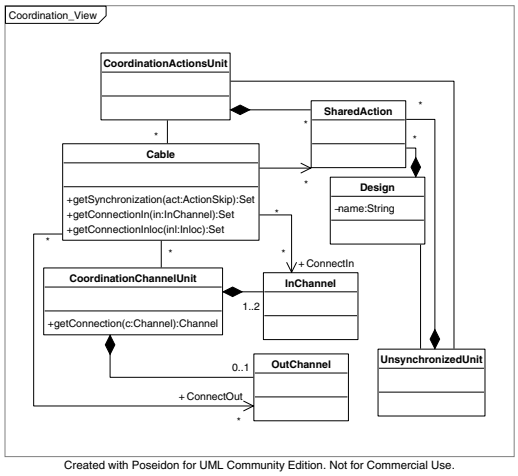


**Figure 4. The coordination view metamodel**

## 4. Coordination View

This view shows which channels are shared and which actions can (not) execute simultaneously. As for channels, the transformation of the architecture into coordination units is easy:

1. Create one coordination unit for each channel.
2. Merge two units into one if a channel of one unit is shared with a channel from the other one in the architecture.

The second step basically uses the architectural cables to compute the transitive closure of sharing. However, this will lead to many singleton units (e.g. all those with a private channel, which cannot be shared by definition). Such a proliferation of entities impairs the clarity that each view is supposed to contribute to system design. Hence, we add a third transformation step:

3. Remove all singleton units.

Fig. 4 shows the diagram, where the coordination unit for channels has been called "CoordinationChannelUnit", and the OCL expressions that capture the above algorithm are:

```
context CoordinationChannelUnit
 inv CableBetweenChannels: self.InChannel->
  forAll(elem | self.cable->select(c |
   c.getConnectionIn(elem)->exists(
   self.getConnection(elem)))->notEmpty())

 inv NoCableBetweenChannels:
  self.allInstances()->forAll(cc1, cc2 |
   cc1 <> cc2 implies
   cc1.cable->intersection(
    cc2.cable)->isEmpty())
```

As for the actions, the transformation is more complex:

1. Create one "UnsynchronisedUnit" (Fig. 4) for each design, containing its unsynchronised actions.
2. Create one synchronised action unit for each maximal combination of actions $a_1, a_2, \ldots, a_n$ belonging to $n$ different designs, such that $a_i$ is synchronised with $a_{i+1}$, for $i = 1, \ldots, n-1$.
3. Associate two units $A$ and $B$ if there is an action in $A$ and an action in $B$ that belong to the same design.

Fig. 5 shows the coordination units for our example's actions: *control* is synchronised with *move* (Fig. 3), *undock* is associated to the unit with *move* because both belong to *located_cart*, *reset* is a singleton because it is the only unsynchronised action in *counter*, etc.

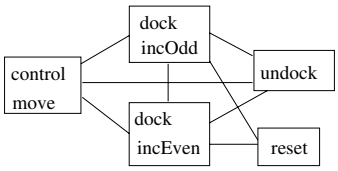


**Figure 5. Coordination view for actions**

The rationale for this transformation is to answer question 1 of Section 3, i.e., to know which actions must and which cannot execute simultaneously. The operational semantics of COMMUNITY states that two actions in different designs *may* execute in parallel, i.e. in some execution steps only one of them may execute. Actions are only *forced* to execute in parallel if they are synchronised. Synchronisations are transitive (i.e., if $a$ is synchronised with $b$ and $b$ with $c$, then all three must co-occur in the trace) and hence the necessity for maximal groups of actions in step 2 of the transformation. Moreover, only one action from each design can be executed at each step. The coordination view puts all this information together. For each execution step, first select a group of units without an association between any pair of them. Then, for each selected synchronised action unit take all its elements, and for each selected unsynchronised action unit just take one of its elements: the actions selected this way will be executed together if their guards allow it.

For our example, four of the units in Fig. 5 form a complete graph and hence only one of them can execute at a time. Moreover, *reset* is "incompatible" with two other units; so it may only be executed together with *control* and *move* or with *undock*.

The diagram of Fig. 4 includes these two kinds of action units and the association between pairs of units, which is subject to an OCL invariant that says: "The two units share an action of the same design".

To sum up, the action units and their associations provide an answer to question 1 of Section 3, while the channel units answer question 2. This is quite useful because while building the architecture, several errors can be made when modelling the interactions: interactions are created between two components at a time and therefore undesirable indirect sharings and synchronisations can be inadvertently created. For example, indirect sharing of two output channels can be easily detected in this view with an OCL consistency check that prevents more than one output channel in each unit.

Having in mind that one of the goals is to improve the design of the system by being able to answer questions 3 and 4 of Section 3, the coordination view will now be enriched with computation information (from the omitted computation view), looking into the guards and assignments. In particular, the transformation has one more step:

4. Create an association between an action unit and a channel unit if an action of the former reads or writes a channel of the latter.

Since the channel units only include shared channels, this new association allows one to see which actions depend indirectly on which channels, the direct dependencies being visible in the omitted computation view. Question 3 is therefore answered by two different views, each only providing the part of the answer that is more relevant to the aspect under consideration. Moreover, question 4 is answered by looking for channel units that are associated to two different action units.

## 5. Distribution View

In its basic form, this view just shows which code and data are co-located. Enriching it with computation and coordination information makes it possible to also tell the designer which locations might have to be in touch during the execution of the system, and which parts of the system are mobile.

Let us start with the basic view, only using location information. In this case, each distribution unit contains a group of location variables that are shared, and all channels and action bodies at those locations (which are actually the same). Notice that although not visible in the concrete syntax, the model of any architecture also includes the location variable $\lambda$, and as such there is one distribution unit for all channels and action bodies located at $\lambda$. The view is constructed as follows:

1. Create one location unit for each location variable, including the one named $\lambda$.
2. Put into each unit all channels and action bodies whose location is given by the corresponding variable.
3. Merge two units if in the architecture a location variable of one unit is shared with some location variable of the other unit.

Proceedings of the Working IEEE/IFIP
Conference on Software Architecture (WICSA'07)
0-7695-2744-2/07 $20.00 © 2007 IEEE

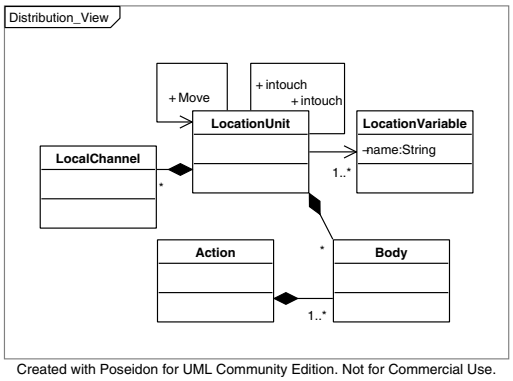


**Figure 6. The distribution view metamodel**

Again, the repeated application of the last step while it is possible means that the transitive closure of the location sharing is computed. The diagram in Fig. 6 and OCL expressions capture this algorithm. The OCL constraints are similar to those used in the coordination view and are therefore omitted.

For our example, the binding between the location variables *theirs*, *pos* and *countLoc* will produce one unit in this view. The distribution units are shown in Fig. 7, with the location variables, the channels, and the actions in separated parts of each unit. Note how *dock* is distributed over two locations. The arcs will be explained later.

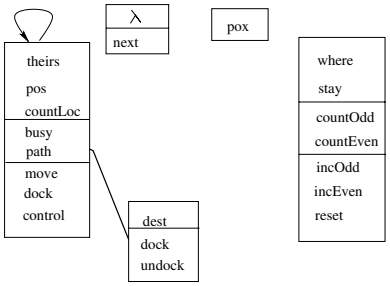


**Figure 7. The example's distribution view**

For the moment, the distribution view allows to easily check one consistency constraint of COMMUNITY: output location variables cannot be shared. It is straightforward to specify an OCL invariant checking that each location unit contains at most one output location variable. But more importantly, the view allows the designer to answer question 5 on which entities are always co-located (Section 3). If there are very few location units, the system is highly centralised, while too many units means a potential high degree of distribution, because all the values of the location variables might be distinct at some point during the system execution. This might confirm what the designer intended or might point to potential problems, requiring the designer to re-define the sharing of location variables or the location of the channels and action bodies.

The next step is to enrich the view with information about which locations might have to be in touch during execution. The operational semantics of COMMUNITY imposes the following restrictions:

- A distributed action can only execute if the locations of its bodies are in touch with each other.
- A guarded command (i.e., action body) can only execute if the location of the body is in touch with the locations of all the channels read and written by the guarded command.
- Synchronised actions can only execute if all their bodies' locations are in touch with each other.

For actions to execute synchronously, or for a distributed action to execute atomically, the involved action bodies must be able to communicate with each other to achieve the transaction, and they must be able to access all channels they read and write. Hence the three constraints impose the involved locations to be in touch.

The evaluation of the three conditions requires using computation information (i.e., looking into the guard expressions and into the assignments) but also coordination information (e.g., which actions are synchronised). Therefore, two location units $A$ and $B$ will be connected by an "InTouch" association (see Fig. 6) if either of the following conditions occurs:

4. A body in $A$ belongs to the same action as some body in $B$.
5. A body in $A$ reads/writes some channel in $B$.
6. A body in $A$ and a body in $B$ belong to two actions that are synchronised.

These conditions can be defined by OCL constraints.

For our example, Fig. 7 shows the "InTouch" associations as undirected arcs. Note that the *dest* and *theirs* locations might have to be in touch due to the distributed *dock* action. Such a view allows the designer to answer question 6 of Section 3, which is important to have a better understanding of the potential distributed execution of the system.

Finally, we wish to know which location units are potentially mobile. By looking into the omitted computation view, one could see which location variables are on the left-hand side of assignments. We could simply flag the location units containing those variables as being mobile. However, we can provide more information by also taking note in which action bodies those assignments occur and what is the location of those bodies. With this information, the designer can easily see which location units move which other ones. The last construction step for our complete distribution view is thus:

7. Create an association "Moves" from location unit $A$ to location unit $B$ if an action body of $A$ assigns to some location variable of $B$.

The diagram in Fig. 6 shows the new association.

In our example there is a "Moves" arrow in Fig. 7 from the *theirs* distribution unit to itself, because that position is changed by *control* located in the same unit.

The complete distribution view, with the two associations among units, allows the designer to detect several potential problems. Firstly, if there is any unit just with one location variable, it means that no action body or channel is located there, which can be an error or simply denote a yet incomplete architecture. Fig. 7 shows one of those empty units (with location variable *pox*). Secondly, any unit without an incoming "Moves" arrow has only static locations (i.e., no action ever changes their values). Hence the code and data in those units never move and stay at the position the variable was initialised with. This kind of information is useful to study the mobility (or lack of it) of the system. For our example, it is clear that a cart destination is at a fixed position. Thirdly, it is easy to see which location units control the movement of which ones, which can lead to the detection of potential problems. A loop arrow indicates that (part of) a component is controlling its own movement; in the case of our example it is the *step_controller*, located at *theirs* and including action *control*.

## 6. Related Work

The need for multiple architectural views has been advocated at least since Perry and Wolf's seminal paper [23]. However, there is no consensus on which view system should be adopted, what are the most relevant views and in what notations they should be specified; the result is a wide range of proposals (e.g., [14, 13, 12, 11, 6]), some of them omitting details about the exact notation to use or the precise definition of views.

Nevertheless, some fundamental types of views emerged from various proposals, and we used them to test our approach: our computation view (omitted from this paper) is like the functional view of [12], the logical view of [14], the conceptual view of [11], and the module views of [6]; our coordination view (Section 4) corresponds to the interaction view of [12], the process view of [14], the execution view of [11], and the C&C views of [6]; and the distribution view (Section 5) is similar in spirit to the physical view of [14] and the allocation views of [6].

Using heterogeneous notations for multiple views has the advantage of using the most expressive and appropriate language for each view. Various approaches (like [14, 10, 11]) have taken advantage of UML's diagrammatic notations and extension capabilities to use it as an ADL that allows the description of multiple views. However, not even the UML, one of the largest "family" of languages ever specified, is capable of encompassing all the views that might be required, as argued in [10]. Moreover, there will be always an architectural abstraction mismatch between the concepts provided by a specialist ADL and those of a generalist language like the UML, as pointed out in [18]. A further disadvantage is that multiple languages make it even harder to handle two basic problems recognised in the SA [19] and Requirements Engineering [20] communities: to establish relations between different views and to check for consistency.

Our approach instead assumes as a starting point that the architect has chosen an ADL that covers those design concepts that are central to the problem at hand. Our goal is to help the architect make the best use of such a language, by extracting multiple view representations from it in order to facilitate the architectural design task. Given that a "one-unified-language-fits-all-views" approach is infeasible anyhow, we believe the restricted set of views that can be derived from a single language is not a major drawback, because the views will correspond to those concerns that lead the architect to choose the ADL.

Given that the approach is based on an overall metamodel that encompasses the ADL's concepts and aggregates them into various views that can be associated in any way, there is no prevailing nor subordinate view, contrary to approaches like [19, 12, 13]. In the latter, the main view describes the structure and other views provide additional information, in different notations. This means that views can only be related through the main system decomposition into components. By contrast, we do not constrain the ways the system is decomposed or views are related. This flexibility also makes non-symmetric combination of views possible, in which any view can be taken as the prevailing one, enriched with information from other views. As we have illustrated in the preceding sections, this allows for the same design question to be answered with different foci, according to the chosen view or combination thereof.

Our approach also differs from [2], in which each component is described in an aspect-oriented way, and there is a metamodel for the aspects, but there is no notion of architectural view to put together the information about a single aspect across all components.

## 7. Concluding Remarks

Views are helpful in the architectural design process because they hide some concerns and focus on others, making it easier for stakeholders to find errors or potential problems. However, views are often defined in an ad-hoc and informal fashion, using loosely related heterogeneous notations, with an incomplete or informal treatment of inconsis-

tency due to inexistent, weak or implicit relationships between views.

This paper proposes an approach to define multiple views that are homogeneous, coherent, relevant, and explicitly related, because they stem from the constructs of a language suitable for the description of important architectural concepts. The approach is based on two principles. First, architectural concepts, their relationships, and their aggregations into various different views should be explicitly defined through a metamodel that enables to relate the various views explicitly and enforce their mutual consistency through constraints. Each view can be described in a declarative way through the metamodel, and operationally as a transformation from the architecture. Second, the decisions on which views to define and how to define them should be guided by an explicit enumeration of the design questions the architect would like the views to answer.

Our approach does not cover all views that might be needed, only those that can be extracted from the ADL. This work is therefore not a replacement of other approaches to architectural views, but an additional technique in the architect's toolbox, when a restricted set of tightly integrated views is sought.

Given that the views emanate from a single architectural description, the design questions can also be answered directly from that description. However, this requires navigating the description and then aggregating various pieces of information. Moreover, often the sought answer is not just a set of architectural entities but also particular relationships, i.e., the answer is basically a graph. Defining views is thus a way of factoring out the navigation and aggregation computations common to various queries, a way of pre-processing information into a graph that makes it easy to answer various design questions at once.

We believe that a metamodel integrating both architecture and views also makes it possible for changes in a concrete view to be propagated to other views, because each view is a model that conforms to part of the overall metamodel that relates all architectural concepts. Such a view updating mechanism requires further research.

## References


[1] IEEE Std 1471-2000 recommended practice for architectural description of software-intensive systems, 2000.

[2] N. Ali, J. Perez, C. Costa, I. Ramos, and J. A. Carsi. Mobile ambients in aspect-oriented software architectures. In *IFIP Working Conf. on Soft. Eng. Techniques*, Springer, 2006.

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

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

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

[6] P. Clements, F. Bachmann, L. Bass, D. Garlan, J. Ivers, R. Little, R. Nord, and J. Stafford. *Documenting Software Architectures: Views and Beyond*. Addison-Wesley, 2002.

[7] J. L. Fiadeiro. *Categories for Software Engineering*. Springer, 2004.

[8] J. L. Fiadeiro, A. Lopes, and M. Wermelinger. A mathematical semantics for architectural connectors. In *Generic Programming*, LNCS 2793, pp. 190–234. Springer, 2003.

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

[10] R. Hilliard. Using the UML for architectural description. In *2nd Intl. Conf. on the Unified Modeling Language*, LNCS 1723, pp. 32–48. Springer, 1999.

[11] C. Hofmeister, R. Nord, and D. Soni. *Applied Software Architecture*. Reading MA Addison-Wesley, 2000.

[12] V. Issarny, T. Saridakis, and A. Zarras. Multi-view description of software architectures. In *3rd Intl. Workshop on Soft. Architecture*, pp. 81–84. ACM, 1998.

[13] J. Kramer and J. Magee. Exposing the skeleton in the coordination closet. In *2nd Intl. Conf. on Coordination Languages and Models*, LNCS 1282, pp. 18–31. Springer, 1997.

[14] P. Kruchten. The 4+1 view model of architecture. *IEEE Software*, 12(6):42–50, 1995.

[15] A. Lopes, J. Fiadeiro, and M. Wermelinger. Architecural primitives for distribution and mobility. In *10th Symposium on the Foundations of Soft. Eng.*, pp. 41–50. ACM, 2002.

[16] A. Lopes and J. L. Fiadeiro. Adding mobility to software architectures. In *2nd Workshop on Foundations of Coordination Languages and Soft. Archs.*, pp. 241–258. Electronic Notes in Theoretical Computer Science 97, 2004.

[17] J. Magee, N. Dulay, S. Eisenbach, and J. Kramer. Specifying distributed software architectures. In *5th European Soft. Eng. Conf.*, 1995.

[18] N. Medvidovic, D. S. Rosenblum, D. F. Redmiles, and J. E. Robbins. Modeling software architectures in the unified modeling language. *ACM Trans. Soft. Eng. Methodol.*, 11(1):2–57, 2002.

[19] J. Muskens, R. Bril, and M. Chaudron. Generalizing consistency checking between software views. *5th Working IEEE/IFIP Conf. on Soft. Arch.*, pp. 169–180, 2005.

[20] B. Nuseibeh, J. Kramer, and A. Finkelstein. Viewpoints: meaningful relationships are difficult! In *25th Intl. Conf. on Soft. Eng.*, pp. 676–681. IEEE Computer Society, 2003.

[21] C. Oliveira and M. Wermelinger. The COMMUNITY workbench. In *26th Intl. Conf. on Soft. Eng.*, pp. 709–710. IEEE Computer Society, 2004.

[22] C. Oliveira, M. Wermelinger, J. Fiadeiro, and A. Lopes. An architectural approach to mobility - the handover case study. In *4th Working IEEE/IFIP Conf. on Soft. Arch.*, pp. 305–308. IEEE Computer Society, 2004.

[23] D. E. Perry and A. L. Wolf. Foundations for the study of software architecture. *ACM SIGSOFT Soft. Eng. Notes*, 17(4):40–52, 1992.



Proceedings of the Working IEEE/IFIP
Conference on Software Architecture (WICSA'07)
0-7695-2744-2/07 $20.00 © 2007 IEEE