Context-Awareness in Software Architectures

Antónia Lopes[1] and José Luiz Fiadeiro[2]

[1] Department of Informatics, Faculty of Sciences, University of Lisbon,
Campo Grande, 1749-016 Lisboa, Portugal
mal@di.fc.ul.pt
[2] Department of Computer Science, University of Leicester,
University Road, Leicester LE1 7RH, UK
jose@fiadeiro.org

Abstract. The growing importance of context-awareness in the construction of adaptable systems requires the development of formal models and notations that can bring this new dimension from middleware concerns into the higher levels of modelling. In this paper, we propose a formal approach to the design of context-aware systems that is well integrated with the concepts and techniques that have been proposed for software architectures. This approach is based on a set of primitives through which the notion of context can be modelled as a first-class entity and context-awareness addressed explicitly as an additional dimension of architectural elements. We illustrate the approach around an image search system.

1 Introduction

In recent years, we have been witnessing a growing interest for software systems that are able to adapt autonomously at run-time. This is justified in part by the need for developing applications that can cope with highly dynamic execution environments. Typical examples are distributed applications with components executed in distinct devices under a wide range of operational conditions that can change over time. This new breed of applications are usually known as *context-aware* systems because they must be responsive to the context in which they execute in order to adapt to changes as they occur.

Much of the work in what has become known as *context-aware computing* is being devoted to the development of middleware infrastructures that facilitate the implementation of this new generation of systems (e.g., [3,7,8,19,20]). In these approaches, the context-aware aspects are divided between the application logic and an infrastructure responsible for the gathering, management and dissemination of contextual information [6,13]. As a result, software developers can concentrate on the application logic without having to be concerned with the way context information needs to be sensed. This separation is important because it promotes the development of general context widgets that can be used, and reused, as components in different configurations of the infrastructure [8].

The added level of complexity that this new dimension brings to software development suggests that context-awareness should be also addressed at higher levels of

R. Morrison and F. Oquendo (Eds.): EWSA 2005, LNCS 3527, pp. 146 – 161, 2005.

abstraction and in earlier phases of the development lifecycle. As remarked in [22], formal models and high-level notations are needed that provide suitable support for modelling and designing context-aware systems before infrastructural concerns come into play. Software designers should have the means for exploring different design solutions that take advantage of contextual information and defining the specific notions of context that are best suited for the systems that are envisaged.

Consider, as an example, the problem of searching a database of images stored in a remote site. Assume that a cheap algorithm is available that quickly identifies images that are potentially interesting. We may think of several different solutions depending on how much awareness the system can have of the context in which it is operating. For instance, a simple and context-unaware solution consists in deciding that all the images must be downloaded from the server and then processed locally. A more flexible solution proposed in [2] consists in executing the cheap algorithm remotely and, depending on the size of the selected images and the current bandwidth, deciding if the remaining more intensive computation should be performed remotely or if the selected images should be dispatched to be locally processed. Other context information, such as the processing power that is available, could be used for taking the decision on where to process the selected images, giving rise to different design solutions for the problem.

In most approaches to software development, context-awareness is not addressed explicitly and, hence, it is not possible to represent explicitly this kind of design decisions: they are simply programmed. This implies that if, for some reason, one needs to change the system to operate according to a different strategy, the system needs to be reprogrammed, possibly interfering with the way orthogonal concerns have been captured in the code. In this paper, we address the design of context-aware systems at the higher architectural levels where such decisions can be modelled in terms of first-class entities and evolved in a compositional way.

Given that context-awareness is especially relevant in the presence of distribution and mobility, we focus our attention in software architectures that address location-dependence explicitly. More concretely, we show how the description of context-awareness aspects of systems can be integrated with the techniques that we have been developing within the IST-2001-32747 project *AGILE – Architectures for Mobility* for supporting distribution and mobility in software architectures [14,15]. The resulting architectural approach promotes the separation of concerns by awarding first-class status to the notion of context. On the one hand, it supports the description of context dependencies of a system's architecture in an explicit way through *context models* that can be understood independently of the specification of the system behaviour. On the other hand, it allows these aspects to be refined and evolved independently of the other concerns.

As such, this paper extends preliminary work presented in [16] where we have focused on the algebraic semantics of much simpler and less expressive design abstractions for context-awareness. We decided to use the same example so that the reader who is familiar with that first proposal can appreciate the added expressive power that we are now proposing. On the other hand, we are now omitting much of the algebraic semantics of the whole approach, and concentrate on the new and revised features.

Most of the added expressive power comes from context models, which are also explored in this paper as requirement specifications for the gathering and dissemination of contextual information. Indeed, context models provide an important abstraction mechanism for modelling, in a formal way, context information in middleware infrastructures that support context sensing.

In Section 2, we briefly review the basic principles of our approach to architectural modelling of distributed and mobile systems and the way it is supported in CommUnity, a prototype language for architectural description. In Section 3, we discuss the context-sensitive behaviour of distributed systems and present the primitives through which architectural models can be made context-aware. We show how higher level notions of context can be modelled and how they can support separation of concerns. In Section 4, we address context modelling in the development of systems that are responsible for the gathering and dissemination of contextual information. We conclude in Section 5 by discussing related and future work.

2 Distributed and Mobile Architectures in CommUnity

As already mentioned, context-awareness is particularly relevant in the presence of mobility. When components are allowed to move across networks, the availability and responsiveness of resources and services are often difficult to predict and out of control [1]. For instance, when visiting a site, a piece of mobile code may fail to link with the libraries that it requires for execution according to its specification. Computational resources such as CPU and memory can no longer be assumed to be fixed as in conventional computing.

Given that, in these situations, the context that a component perceives is to a great extent related with its location, it is important that context-awareness be addressed in approaches that, like CommUnity [10], do not adopt location-transparency as an abstraction principle and address distribution as a first-class concern in par with computation and coordination. In this section, we provide a brief review of the primitives that are used in CommUnity to capture distribution and mobility and illustrate how they support the description of mobile systems at an architectural level of design. A more detailed account of this approach can be found in [14,15].

CommUnity is a parallel program design language in the style of Unity [5] and Interaction Processes [11] that we have been developing for formalising architecture description primitives. A CommUnity design is defined in terms of *channels*, *actions* and *location variables*.

Channels provide the means for the exchange of data between different components. The declaration of a channel as *input* or *output* defines its role in the exchange of data with the environment. Declaring a channel to be *private* means that it models internal exchanges of data, i.e. between different parts of the component, and that these exchanges are not perceived by the environment. Output and private channels are said to be *local* because they are controlled by the component, i.e. the environment cannot modify the values that are made available on these channels. Input channels are used by the component to read data from the environment.

Actions provide points for rendez-vous synchronisation. Each action is associated with a set of guarded commands that is executed atomically. These commands are of the form

```
exp → x1:=exp1 || x2:=exp2 || …
```

and define computations over the data that is available in the channels. The expression *exp* defines the enabling condition of the command. When the command is executed, all the assignments are preformed atomically.

Location variables act as "containers" for data and code that can be moved across a communication network. Every local channel x is statically associated with a location variable l (we write $x@l$). The same happens for the guarded commands associated with actions. The idea is that the position of the space where the values of a channel are available or a command is executed is determined by the position of the container in which the channel or the command was placed.

We start with a very simple design of one of the components of the image search system – the filter. The role of this component in the system is to interact with the database in order to obtain the images that are potentially interesting and calculate the size of these images.

```
design Filter is
inloc  lf:Loc
in     db:set(image)
out    img@lf:set(image), size@lf:nat
prv    s@lf:[0..4]
do     beg@lf: s=0 → s:=1
  []   req@lf: s=1 → s:=2
  []   filt@lf: s=2 → s:=3||img:=filterop(db)
  []   rel@lf: s=3 → s:=4||size:=imgsize(img)
  []   end@lf: s=4 → skip
```

Fig. 1. The design *Filter*

According to this design, the database is made available by the environment through the input channel *db*. Once the filter is requested to begin its activity through the execution of the action *beg,* it first requests access to the database (action *req)*, then it proceeds with the filtering activity, after which it computes the size of the images. The selected images and their size are made available to the environment through the output channels *img* and *size*. After releasing the database (action *rel*), action *end* becomes enabled meaning that the activities of the filter have ceased.

Designs are defined over a collection of data types that are used for structuring the data that the channels transmit and define the operations that perform the computations that are required. In order to remain independent of any specific language for the definition of these data types, we take them in the form of an algebraic specification. In the example, the images are modelled through a data type that involves thedomain *image*. *Filter* also makes use of the operations *filterop* and *imgsize* that abstract the selection process of the images considered to be of interest and the computation of the size of these images, respectively.

In what concerns distribution, the design *Filter* models a centralised component because all its constituents are located at the same variable *lf*. The fact that *lf* is declared as input means that it is under the control of the environment, i.e. the position where the filter performs its activities is determined by the rest of the system in which it is integrated as a component. The underlying space of distribution and mobility is constituted by the set of possible values of a special data sort *Loc* and whatever operations are necessary to characterise locations such as hierarchies or taxonomies.

By taking advantage of mobility, we may opt for the migration of the filter to the host of the database. This decision can be integrated in the design of the filter as follows.

```
design MobileFilter is
inloc  lr, lc:Loc
outloc lf:Loc
in     db:set(image)
out    img@lf:set(image), size@lf:nat
prv    s@lf:[0..4], q@lf:[0..2]
do     beg@lf: s=0 ∧ q=0 → s:=1 || q:=1
 []    move@lf: q=1 → q:=2
           @lc: true → lf:=lr
 []    req@lf: s=1 ∧ q=2 → s:=2
 []    filt@lf: s=2 → s:=3 || img:=filterop(db)
 []    rel@lf: s=3 → s:=4 || size:=imgsize(img)
 []    end@lf: s=4 → skip
```

Fig. 2. The design *MobileFilter*

This design has two input location variables *lr* and *lc* accounting for the location of the database server and the client application, respectively. The location of the filter (*lf*) is now captured by an output variable because it became under the control of the extended component. *MobileFilter* models a filter that can only migrate after it has begun its activities. Furthermore, it can only request access to the database after it has been moved to the database host. The command that gives rise to the migration, modelled by the assignment *lf:=lr*, is issued at the location of the client.

It is important to notice that the design decisions concerning filter migration can be modelled in an independent way through the following "mobility controller":

```
design MobCont is
inloc  lr, lc: Loc
outloc l: Loc
prv    q@l:[0..2]
do     pre@l: q=0 → q:=1
 []    move@l: q=1 → q:=2
           @lc: true → l:=lr
 []    pos@l: q=2 → skip
```

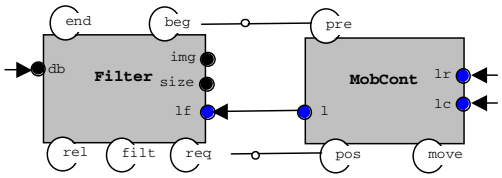
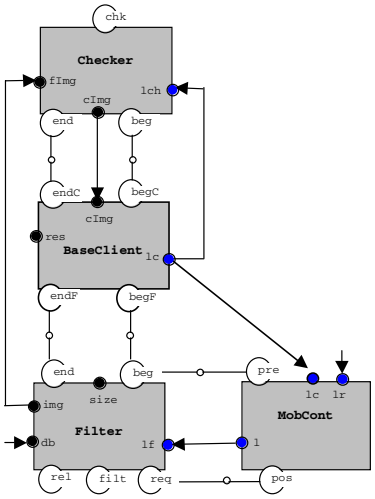


Fig. 3. Externalisation of the design decisions concerning the mobility of the filter

This controller externalises design decisions into an explicit connector that can be superposed over different components. In particular, it can be applied to the filter at hand, as depicted above.

In CommUnity, interaction between a component and its environment relies on the synchronisation of actions and exchange of data through input and output channels. The design of interactions between different components is supported through configurations; these are diagrams that exhibit interconnections between components. In configuration diagrams, components only depict their public elements. The lines connecting actions establish synchronisation points – these actions have to be executed synchronously. The lines connecting channels or location variables establish I/O communication. In contrast with most architecture description languages, these "boxes and lines" have a mathematical semantics: configuration diagrams are in fact diagrams in a category of CommUnity designs whose morphisms capture notions of superposition [10]. The semantics of such a diagram is given by its colimit [9]. In the case of the configuration above, this colimit returns exactly the design *MobileFilter*.

For completeness, we conclude this section by providing the architecture of an image search system. In this system, the filter algorithm is executed remotely. Upon its termination, the filtered images are downloaded by the *Checker* and checked locally, wherever the client is located.




```
design BaseClient is
outloc lc:Loc
in   cImg:set(img)
out  res@lc:set(img)
prv  t@lc:[0..4]
do   begF@lc: t=0 → t:=1
[]   endF@lc: t=1 → t:=2
[]   begC@lc: t=2 → t:=3
[]   endC@lc: t=3→ t:=4|| res:=cImg

design Checker is
inloc lch:Loc
in   fImg:set(img)
out  cImg@lch:set(img)
prv  t@lch:[0..2]
do   beg@lch: t=0 → t:=1
[]   chk@lch: t=1 → t:=2||
                    cImg:=checkop(fImg)
[]   end@lch: t=2 → skip
```

Fig. 4. An architecture of an *Image Search System*

3 Context-Aware Architectures in CommUnity

Section 2 introduced the primitives made available in CommUnity for the design of mobile systems and illustrated how they support the explicit representation of distribution/mobility in software architectures. In this section, we show how context-awareness can be integrated in this architectural approach in a way that supports the definition of application specific notions of context and the design of components

and connectors that have to deal with changes of context as part of their intrinsic behaviour.

In our approach, "local contexts" exist in the positions of the underlying space of mobility. They are all instances of a given type. This type captures static properties of contexts, i.e. features that are common to all instances as available in given locations. However, not all instances need to be the same, of course. Hence, for instance, when an action migrates from a location to another, the type of the resources that it needs for its computations will have been statically determined, i.e. at design time, but the actual resources available (say arithmetic precision) will only be known at run time.

3.1 Context-Sensitiveness

So far, we have neglected how behaviour is affected by factors like network connectivity or the set of services that are available at each location. Consider, for instance, *MobileFilter* as presented in Section 2. The filter may not be able to migrate to the database host for reasons such as restrictive security policies enforced at the destination or simply because of lack of connectivity. Once at the database host, the behaviour of the filter still depends on the context of execution. For instance, the filtering of the images cannot be performed if the computational services that this activity requires cannot be found at the given location.

This shows that there exists a dimension of context-sensitiveness that is orthogonal to context-awareness: even a component that, like *MobileFilter*, does not take advantage of context information, has a context-sensitive behaviour. This dimension is related with what was identified in [6] as the *active aspect* of context – the aspect that concerns the characteristics of the surrounding environment that are *determinant* in the behaviour of mobile computing systems. In contrast, the *passive aspect* of context consists of the characteristics that are *relevant* but not critical.

In CommUnity, each of the architectural dimensions – computation, communication and distribution – depends on different characteristics of the environment: *Computations*, as performed by individual components, are constrained by the *resources* and *services* available at the positions where the components are located; *Communication* among components can only take place when they are located in positions that are *"in touch"* with each other; *Movement* of components from one position to another is constrained by *"reachability"*.

Therefore, in CommUnity, the active aspect of context consists of *computational resources*, *computational services*, *connectivity* and *reachability*. These characteristics of the environment are considered as part of the context of any system design, regardless of its particular application domain.

More concretely, a *context type* in CommUnity has an application-independent part defined by the four special observables <*rsc,serv,bt,reach*>. The intuition behind these observables and the nature of their observations is given below.

– The observations of *rsc* are natural numbers and should be regarded as measures of the computational resources available locally.

– The observable *serv* provides access to the local interpretations (in the sense of implementations) of the operations of the underlying data type specification. Given an operation $f:s_1...s_n \rightarrow s$, *serv(f)* can be undefined, meaning that there is not a local interpretation of *f*. If defined, *serv(f)* is a pair (*F*,*amm*) where *F* is a computational service that transforms *n*-tuples of values of type $s_1...s_n$ in a value of type *s* and *amm* is a natural number that represents the level of resources required by *F*. In order to remain independent of any language for the definition of these computational services, we take them simply as mathematical functions. We shall use $[s_1...s_n \rightarrow s]$ to denote the type of (*F*,*amm*).
– The observations of *bt* and *reach* are sets of positions of the space (i.e., values of type *Loc*). They represent the positions reachable from the *self* position through, respectively, communication and movement.

These observables play a determinant role in the behaviour of a system, which is captured by the context-sensitive semantics of CommUnity designs as formally defined in [16]. The enabling condition of a distributed action depends on the values of *rsc*, *serv*, *bt* and *reach*. More specifically, a distributed action *g* is enabled in a certain state iff, in this state, the positions where its local actions execute are mutually in touch (*bt*) and, for each such position *l:*

– the channels that need to be read or written are located in positions that are in touch with the position of *l* (*bt*);
– the operations and resources necessary to evaluate the guard and perform the computations are available;
– location variables are only assigned positions that are within reach (*reach*);
– the guard evaluates to true.

Furthermore, the effects of an action on the system state are determined by the "interpretations" of the operations used to specify these effects *(serv)* at each of the locations where the action execution is distributed.

CommUnity, equipped with this context-sensitive semantics, provides us with the means to design systems that deal with situations where the availability of critical resources is not guaranteed. The absence of a critical resource is not regarded as a runtime error but rather as a blocking condition of the actions whose execution depends on this resource. By defining alternative actions that are enabled in these situations, we can specify how the system is required to operate in such situations.

For instance, in order to model a filter that performs the filtering activities locally if migration to the database host is not possible, we only need to introduce the action

```
[]    stay@lf: q=1 → q:=2
```

in *MobileFilter*. This action can be executed in the same system state as *move* and, hence, if the action *move* is blocked because the remote location *lr* is unreachable, the system can make progress through the execution of *stay*. If the location *lr* is reachable, then both actions can be executed and a non-deterministic choice will be made.

```
design FlexibleFilter is
inloc  lr, lc:Loc
outloc lf:Loc
in     db:set(image)
out    img@lf:set(image), size@lf:nat
prv    s@lf:[0..4], q@lf:[0..2], op@lf:[set(image)->set(image)]
do     beg@lf: s=0 ∧ q=0 → s:=1|| q:=1|| op:=serv(filterop)
  []   move@lf: q=1 → q:=2
            @lc: true → lf:=lr
  []   stay@lf: q=1 ∧ lr∉reach → q:=2
  []   req@lf: s=1 ∧ q=2 → s:=2
  []   filt@lf: s=2 → s:=3|| img:=op(db)
  []   rel@lf: s=3 → s:=4|| size:=imgsize(img)
[]   end@lf: s=4 → skip
```

Fig. 5. A context-aware *Filter*

3.2 Context-Awareness

In order to obtain a more expressive model of context-aware computing, software designers should be able to take advantage of contextual information by explicitly defining how it affects the behaviour of the system. For instance, in the case of the filter, we would like to be able to express that the decision to download the images from the remote database is restricted to the situations in which the migration to the database host is not possible. This can be achieved as presented in Figure 5.

The design *FlexibleFilter* uses two new primitives – the special observables *reach* and *serv*. The observable *reach* is used for expressing that the remote execution of the filter is preferred to the local execution. This is achieved by defining that action *stay* is blocked if $lr \in reach$. Because this expression is evaluated at *lf*, the contextual information that is used is the one available there and, hence, $lr \notin reach$ means that *lr* is not reachable from *lf*. The observable *serv* is used for ensuring that the filtering activity, even when performed remotely, uses the interpretation of *filterop* available locally in the client host. To this purpose, we introduced a new private channel *op* that, at the beginning of the filtering activity, is assigned the local interpretation of *filterop* and keeps it with the filter, as part of its state. Moreover, the command associated to action *filt* was modified so that the filtering activity is performed with the computational service available in channel *op*.

Indeed, the abstract data type specification associated with a CommUnity design identifies the nature of the data and operations that may be required at the positions of the distribution topology, e.g. in terms of libraries or packages. Different positions may provide different implementations for these data types. It may also happen that a given position does not provide an implementation for all the operations that may be required for the execution of a given action. We know all too well that software installation often fails because required libraries are missing in the target platform…

This justifies that, in CommUnity, part of the context identifies the libraries available at every location *(serv)* and that given functions can be transmitted as data from one location to another, either because they are missing, or to make sure that a specific version is used instead of the default available at the target. Such facilities are already available in most platforms. For instance, one of the central and unique fea-

tures of RMI is its ability to download the bytecodes (or simply code) of an object's class if the class is not defined in the receiver's virtual machine.

We have illustrated the use of application-independent observables in the specification of designs. If other characteristics of the environment are considered relevant for the design of the system, they have to be defined as part of its context type.

3.3 Context Types

A *context type* includes the fixed set of observables defined in 3.1 and an application-dependent set of other observables. Each observable is of the form $obs{:}s_1 \ldots s_n \rightarrow s$, where $s_1,\ldots,s_n$, s are sorts of an abstract data type specification that includes the sort *Loc* of locations. These data sorts are used for structuring the required contextual data. Other operations may be defined for manipulating that data as required by the application.

Each observable represents a specific context concept. It identifies the nature of observations that are relevant for the system at hand and the way this information is accessed by the application. Dependencies between different types of contextual information can be expressed through axioms of the specification.

The integration of context-aware decisions in the behaviour of CommUnity designs is based on the simple use of terms built over observables in the guards and effects of local actions. In the evaluation of these terms, it is the location of the action that determines the position of the space where the required observations are made.

Consider the design of an image search system in which the choice of where to perform the checking of the images selected by the filter is based on their size, the processing power available in the remote and local machines, and the bandwidth available between the two hosts. This context-aware decision can be achieved by superposing the following mobility controller over the *checker*.

```
design FlexMobCont is
inloc  lr, lc:Loc
outloc l:Loc
in     sz:nat
prv    q@l:[0..2]
do     pre@l: q=0 → q:=1
 []    move@l: q=1 → q:=2
           @lc: crit(bdw(lr),ppw(lc),ppw(lr),sz) → l:=lr
 []    stay@l: q=1 → q:=2
           @lc: ¬crit(bdw(lr),ppw(lc),ppw(lr),sz) → skip
 []    pos@l: q=2 → skip
```

Fig. 6. A context-aware *Mobility Controller*

FlexMobCont models a controller similar to the one we designed for controlling the mobility of the filter. In order to accommodate the criteria for migration, we introduced an action *stay* and an input channel *sz* (accounting for the size of the selected images) and we modified the guard of *move*.

The observables *bdw* and *ppw,* the nature of their observations and the criteria for migration are defined by the context type *network&cpu*. This context type de-

fines that, in a given position of the space, the system is interested in a measure of the bandwidth available between this position and all the others, delivered as a natural number. These values are constrained to be related with the observations of connectivity (*bt*) in the obvious way. It is also defined that every local context must have information about the processing power *(ppw)* available at every position of the space.

The structure of the observations of *ppw* is defined by *ppwData* and operations *newPpw, better* and *crit*. The operation *better* is used to decide if the checker should migrate to the database host. The criteria to perform the migration, modelled by the operation *crit*, are based on the estimated transfer time of the images: migration should be carried out if the transfer of the images will take too long (above *timeThereshold*) or if there are too many images to scan (above *sizeThereshold*) and the processing power available remotely is considered better than the one available locally.

Context types may also include derived observables – observables whose values are completely determined by the values of the other observables. This is extremely useful for defining higher-level notions of context based on simpler sensed contexts.

```
context type network&cpu is
   sensed observables
         bdw: Loc -> nat
           // bandwidth available between self and given position
         ppw: Loc -> ppwData
           // processing power available in the given position
   constrained by p:Loc
          bdw(p)=0 ≡ p∉bt
   sorts
         ppwData
           // defines the nature of observations of ppw
   operations
         newPpw: nat natPercentage -> ppwData
           // creates a ppwdata from the power of the processor and the
           // percentage of processing power in use
         better: ppwData ppwData -> bool
           // Is the first ppw "better" than the second one?
         crit: nat ppwData ppwData nat -> bool
           // Are the given conditions favourable for migration?
         factor: nat
         timeThreshold, sizeThreshold: nat
   axioms v1,v2,perc1,perc2,b,s:nat, p1,p2:ppwData
         (1) better(newPpw(v1,perc1), newPpw(v2,perc2)) ≡
                (v1/v2)>factor ∧ (100-perc1)*v1>(100-perc2)*v2
         (2) crit(b,p1,p2,s) ≡
                s*b>timeThreshold ∨ (s>sizeThreshold ∧ better(p2,p1))
```

Fig. 7. The context type *network&cpu*

For instance, we can extend the context type *network&cpu* with:

```
 derived observervables
         migr: Loc Loc nat ->bool
         migr(l1,l2,s)=crit(bdw(l1),ppw(l2),ppw(l1),s)
```

This defines a new observable *migr* whose values have not to be sensed but rather inferred from *bdw* and *ppw*. Intuitively, *migr* indicates whether, according to the criteria defined by *crit*, the actual context is favourable for migration or not.

We can now use this new observable to design a controller with the same functionally of *FlexMobCont*: we only need to replace *crit(bdw(lr),ppw(lc),ppw(lr),sz)* by *migr(lr,lc,sz)* in the guards of actions *move* and *stay*.

Although the functionality expressed in the two designs is the same, they provide different support for evolution and reuse. Because the condition *migr(lr,lc,sz)* is less specific than *crit(bdw(lr),ppw(lc),ppw(lr),sz)*, if we use the new controller for the mobility of a component, we have more chances of being able to change the design decision adopted for migration just by replacing *network&cpu* by an appropriate context type. All that is required is that the new condition can be expressed in terms of the values available in the channel *sz* and the locations *lr* and *lc*.

4 Context Modelling

So far, we have addressed context modelling from the perspective of the software designer that is in charge of the application logic. Our focus has been on the definition of modelling primitives that allow software architects to represent and organise the contextual information in which a system is interested and to take advantage of contextual information in the specification of system components and connectors.

The use of contextual information by an application assumes the existence of another system that senses the current context and delivers it to the application. In this section, we address context modelling from the perspective of the development of context-sensing systems – the systems responsible for the gathering, management and delivery of contextual information.

We start by analysing the role of context types in the development of context-sensing systems. Our view is that there exists one context-sensing system working on behalf of each context-aware application. In situations in which several applications are interested in the same contextual data, this does not mean that the sensing work has to be replicated because, for instance, the corresponding sensing systems can be components of an infrastructure that centralises the collection of all contextual data.

From the perspective of context-sensing systems, context types play the role of requirement specifications. They define what must be sensed, how the sensed data must be abstracted and the interface offered by the sensing system to the application layer. This interface consists of the *observables* through which it is possible to gain access to contextual data (both sensed and derived) and the *operations* through which it is possible to manipulate this data. Because the sensing system provides an encapsulation of contextual data as an abstract data type, all the operations that manipulate contextual data have to be provided by this system.

As specifications of requirements for context-sensing systems, there are important issues that context types do not address. One of these issues is the type of sensing that should be adopted for each sensed observable – *on demand* or *continuous*. In the case of *on demand* sensing, the information about the actual context is collected only

when there is a request issued by the application, which waits for this information to proceed. In the case of *continuous* sensing, the sensing system is supposed to be proactively collecting the contextual information. Because this tends to be a costly activity, and different applications typically have different needs, it is important that designers have the means to describe at which frequency new data has to be obtained.

Mechanisms for expressing this kind of requirements can be easily integrated with the notion of context type, giving rise to what we designate by *context model*. An example of a context model for the image search system is presented in Figure 8.

```
context model network&cpu is
      contex type network&cpu
      on-demand
         ppw: Loc->ppwData
      continuous
         bdw: Loc->nat with frequency 50:natPercentage
```

Fig. 8. The context model *network&cpu*

In a context model, sensed observables are grouped according to the type of sensing they require. For observables that require continuous monitoring, the frequency at which data should be gathered is also defined using data types, keeping the level of abstraction (it would not make sense to ask designers to provide real time requirements for their high-level designs).

Another important concern in context modelling is to guarantee that the application and the context-sensing system have a common semantic understanding of the sensed information. This requires context models that support the definition of what has to be sensed in a precise way. Because the models we proposed are not powerful enough, we envisage their extension with mappings to standard ontologies representing the diversity of environment characteristics that is reasonable to sense in specific domains. The idea is for these mappings to establish the semantic understanding of each sensed observable of the context type. Unfortunately, despite recent developments in the area of the Semantic Web in this direction [19,21], such standardised ontologies are not yet available.

5 Concluding Remarks

In this paper, we addressed the integration of context-awareness in the set of aspects that systems architectures should be able to deal with. Having adopted an infrastructure-centred view of context-awareness development, we focused on the definition of modelling primitives that allow software architects (i) to represent and organise the contextual information that a system requires and (ii) to take advantage of contextual information in the specification of the components and connectors of the system.

Our proposal is based on the extension of system architectures with a new design element – *context models*, which supports the modelling of the context of a system as a first-class entity separable from its application logic.

Taking an example in which mobility is used as a tool to adapt to variations in the execution environment, we illustrated how CommUnity supports the design of components and connectors that take advantage of contextual information to adapt their behaviour. Adaptable behaviour is specified essentially through the specification of sets of alternative actions – actions that are enabled at the same system's states but not in the same context.

A complementary and important approach to adaptation in software architectures is the one that addresses adaptation at the configuration level through dynamic reconfiguration. Most of the work devoted to adaptation in software architectures focuses on this kind of adaptation (e.g., [4,12,18]). However, existing approaches fail to give a first-class status to the notion of context. Either they are developed having in mind specific aspects of the execution environment (essentially, performance-oriented aspects) or they support implicit definitions of context, hard-wired in different parts of the system architectural description. For this reason, we plan to investigate structural adaptation of systems architectures that include context models as design elements.

A related approach that is important to mention is the conceptual model for context-aware architectures proposed in [17]. In this work, an infrastructure-centred view is not adopted. Instead, context sensing is viewed just as an aspect of the system; components and connectors are themselves involved in the gathering of context information.

Because context models can be understood independently of the rest of the architecture, they were also investigated in this paper as specifications of requirements for context-sensing systems. This is an important perspective because it contributes to the understanding of the abstractions that should be provided by the middleware infrastructures that support the development of context-aware applications. As mentioned in [13], most existing infrastructures are built upon informal context models that lack in expressive power.

Context modelling concepts and techniques have also been investigated in the field of Pervasive Computing where some formal models of context have been proposed [13,20]. Compared with ours, these models are more powerful: for instance, they address uncertainty in and quality of contextual information. However, because these models were developed with different aims, they do not provide adequate support for defining abstract notions of contexts as required for the high-level design of context-aware systems. For instance, the aim of the context model of [20] is to provide a uniform representation of contextual information in context infrastructures for entities such as context providers, synthesizers and consumers. The language used in [13] is an information modelling technique suited for describing the context information available through a context infrastructure. Based on these fine-grained models, higher-level concepts can be defined and used as programming abstractions.

As far as modelling techniques are concerned, ContextUnity [22] is the approach most closely related to ours. In ContextUnity, systems are also designed by assuming that contextual information is transparently maintained. However, although the model of context is, as in CommUnity, explicit and separable from the behaviour specification, it cannot be externalised and modelled as an independent system dimension. In ContextUnity each component of the system is regarded as an autonomous agent and has a specific notion of context that is defined by a set of observables

whose values depend exclusively on variables of other components in the system. As we have seen, context information in CommUnity is orthogonal to the decomposition of the system in components; it refers to any collection of characteristics and properties of the environment that are relevant to the system and *are not under its direct control*. This, we believe, adds flexibility and adaptability to system models as context-awareness is addressed as an independent architectural concern.

Acknowledgements

This work was partially supported by the EC through the IST-2001-32747 Project AGILE – *Architectures for Mobility* and by FCT through the POSI/EIA/60692/2004 Project MICAS – *Middleware for Context-aware and Adaptative Systems*. We wish to thank our partners in both projects for much useful feedback.

References

1. IST Global Computing Initiative, http://www. cordis.lu/ist/fet/gc.html.
2. M.Acharya, M.Ranganathan and J.Saltz, "Sumatra:A Language for Resource-aware Mobile programs", *Mobile Object Systems: Towards the Programmable Internet*, 1997.
3. L.Capra, W.Emmerich and C.Mascolo, "CARISMA:Context-aware Reflective mIddleware System for Mobile Applications", *Trans. Softw. Eng. Journal*, **29**(10), 929-245, 2003.
4. M.Castaldi, A.Carzaniga, P.Inverardi and A.Wolf, "A Light-weight Infrastructure for Reconfiguring Applications", *Software Reconfiguration Management Workshop*, LNCS 2649, Springer, 2003.
5. K.Chandy and J.Misra, *Parallel Program Design - A Foundation*, Addison-Wesley, 1988.
6. G.Chen and D.Kotz, "A Survey of Context-Aware Mobile Computing Survey", Dartmouth CS-TR-2000-381, 2000.
7. G.Chen and D.Kotz. "Context-sensitive resource discovery", *Pervasive 2003*, 243-252, 2003.
8. A.Dey, D.Salber and G.D.Abowd, "A conceptual framework and a toolkit for supporting the rapid prototyping of cw applications", *Human-Computer Interaction*, **16**(2-4), 97-166, 2001.
9. J.L.Fiadeiro, *Categories for Software Engineering*, Springer 2004.
10. J.L.Fiadeiro, A.Lopes and M.Wermelinger, "A Mathematical Semantics for Architectural Connectors", *Generic Programming*, LNCS 2793, 190-234, Springer, 2003.
11. N.Francez and I.Forman, *Interacting Processes*, Addison-Wesley, 1996.
12. D.Garlan *et al*, "Software Architecture-based Adaptation for Pervasive Systems", *Trends in Network and Pervasive Computing*, LNCS 2299, Springer, 2002.
13. K.Henricksen and J.Indulska, "A Software Engineering Framework for Context-aware Pervasive Computing", *Pervasive 2004*, 77-86, 2004.
14. A.Lopes, J.L.Fiadeiro and M.Wermelinger, "Architectural Primitives for Distribution and Mobility", *Proc. FSE-10*, 41-50, ACM Press, 2002.
15. A.Lopes and J. L.Fiadeiro, "Adding Mobility to Software Architectures", *ENCTS* **97**, 241-258, Elsevier Science, 2004.
16. A.Lopes and J.L.Fiadeiro, "Algebraic Semantics of Design Abstractions for Context-Awareness", *Algebraic Development Techniques*, LNCS 3423, 79-93, Springer, 2005.

17. J.J.Martínez and I.Ramos, "A Conceptual Model for Context-Aware Dynamic Architectures", *Proc. ICDCSW*, 2003.
18. P.Oreizy *et al*, "An Architecture-based Approach to Self-Adaptive Software", *Intelligent Systems* **14**(3), 54-62, 1999.
19. M.Roman, C.Hess, R.Cerqueira and A.Ranganathan, "GAIA: A Middleware Infrastructure to Enable Active Spaces", *Pervasive Computing* **1**(4), 74-83, 2002.
20. A.Ranganathan and R.Campbell, "An infrastructure for context-awareness based on first-order logic", *Pers Ubiquit Comput*, 7, 353-364, 2003.
21. A.Ranganathan *et al*, "Ontologies in a Pervasive Computing Environment", *Proc. IJAIC*, 2003.
22. G.-C.Roman, C.Julien and J.Payton, "A Formal Treatment of Context-Awareness", *FASE*, LNCS 2984, 12-36, Springer, 2004.