Algebraic Semantics of Design Abstractions for Context-Awareness

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. We investigate essential features of contexts and proper abstractions for modelling context-awareness within CommUnity, a language that we have been developing to support architectural design of distributed and mobile system. Under the assumption that the context that a component perceives is determined by its current position, we explore the use of abstract data types for defining design primitives through which different notions of context can be modelled explicitly according to the application domain.

1 Introduction

One of the major challenges in modern distributed computing is to deal with highly dynamic operation contexts. As components are entitled to move across networks whose nodes can themselves be mobile and required to execute in different locations, the availability and responsiveness of resources and services are often difficult to predict and out of control [1]. Computational resources such as CPU and memory are no longer fixed as in conventional computing. When visiting a site, a piece of mobile code may fail to link with libraries it requires for its computation. Network connectivity and bandwidth are other factors that can affect, in a fundamental way, the behaviour of mobile computing systems. In this setting, it is no longer reasonable to treat attempts at using absent resources or accessing unavailable services as exceptions. Systems should be provided with the means to observe the context in which they operate. They should also be developed taking into account the different conditions in which they can be required to operate.

The possibility of observing the context also opens new and more sophisticated ways of designing systems. For instance, as pointed out in [2], in order to scan a database of images stored at a remote site for which there is a cheap algorithm that quickly identifies those that are potentially interesting, we may conceive a solution that takes advantage of the perception of the context as follows. First, the cheap algo-

rithm is executed remotely. Then, the size of the selected images, the current network latency and the processing power that is available in both hosts are used to decide if the remaining (intensive) computation should be performed remotely or if the images selected by the cheap algorithm should be sent back.

Context-awareness is the emergent computing paradigm that is addressing this kind of approach to mobile systems construction. By *context*, one refers to the part of the operation conditions of a running system that may affect its behaviour. Typically, different kinds of applications require different notions of context. Hence, it is important that formalisms for designing mobile systems consider contexts as first-class design entities. If a specific notion of context is fixed, for instance as in Ambients [4], the encoding of a different notion of context can be cumbersome and entangled with other aspects, if at all possible. On the contrary, if we support the explicit modelling of notions of context according to the application domain, we make it possible for such aspects to be progressively refined through the addition of detail, without interfering with the parts of the system already designed.

In this paper, we investigate essential features of contexts and proper abstractions for modelling context-awareness. Under the assumption that the context that a component perceives is determined by its current position, we explore the use of abstract data types for defining design primitives through which different notions of context can be modelled explicitly according to the application domain.

Our approach provides means for describing explicitly how contexts affect the behaviour of systems. Any notion of context defines a specific set of constructs that can be used in the specification of system actions. These constructs allow us to enrich architectural models with context-aware patterns of computation, coordination and mobility in a non-intrusive way. That is, context-awareness can be added as an orthogonal dimension without interfering with context-independent decisions made at the level of computation (e.g. properties of data returned by services), coordination (e.g. interactions managed through a shared control unit) and mobility (e.g. shared control unit at a fixed location).

We present this approach over CommUnity, a language that we have been developing to support architectural design [11]. CommUnity was recently extended in order to support the description of mobile systems [14]. This extension addresses a specific notion of context that is centred on the notions of connectivity and reachability of positions. In this paper, we take this extension of CommUnity a step further in order to support the definition of application-specific notions of context. In particular, we will take into account the availability of computational resources and services at the locations in which components are placed.

2 Designing Mobile Systems in CommUnity

CommUnity is a parallel program design language similar to Unity [6] and IP [10] in its computational model but relying on communication rather than shared memory for interaction. In CommUnity, the individual components of a system are designed in terms of *channels* and *actions*. The role of the channels is to exchange data between

different components. Actions are associated with guarded commands that manipulate and compute data and provide points for rendez-vous synchronisation with other components. In order to support distribution and mobility, CommUnity provides mechanisms for assigning channels to locations and distributing the execution of actions among different locations.

To illustrate the way components of mobile systems can be designed in CommUnity and, later on, other aspects of our model, we use the image search problem mentioned in the introduction. We start with a high-level description of a server that can be used to control the access to a shared resource:

```
design server[N:nat] is
outloc l
prv    gr@l:[0..N], rqᵢ@l:bool i=1,…,N
do     i=1,…,N
       reqᵢ[rqᵢ]@l: ¬rqᵢ → rqᵢ:=true
 []    grtᵢ[gr]@l: rqᵢ∧gr=0 → gr:=i
 []    relᵢ[gr,rqᵢ]@l: gr=i → rqᵢ:=false||gr:=0
```

This design models the basic functionalities of a server that supports up to N connections and acts like a scheduler by allowing only one connection to be on at any time. Through each action req_i it accepts requests for using the resource, which it records in the private channel rq_i. Private channels are internal in the sense that the data that they store is not available to the environment but only for interaction inside the component, namely among the actions that the component can perform. Through each action grt_i the server signals that access to the resource has been granted to the particular client that has requested it through the action req_i. This is because the condition rq_i is part of the guard of action grt_i, which means that a request must be pending on the i-th connection. The other conjunct of the guard – $gr=0$ – ensures that no other request has been granted. The private channel gr is used, precisely, to communicate the status of the connections: it takes the value 0 if the resource is free and the value $i:1..N$ if a request has been granted along the i-th connection. The server acknowledges the release of the resource through the actions rel_i by resetting both gr and rq_i.

A location variable l is declared for handling distribution and mobility. In fact, the server is modelled as a centralised component because all its constituents are located at l. Furthermore, the server is not mobile because this location variable is declared to be output – which means that it is under the control of the component – but is not updated by any of its actions – meaning that it remains invariant.

In CommUnity, a component is designed in terms of a set of channels V (declared as input, output or private), a set of location variables L (input or output) and a set of actions Γ (shared or private) that, together, constitute what we call a signature. We use X to denote $V \cup L$. Input channels are used for reading data from the environment. Output and private channels are controlled locally by the component. Output channels allow the environment to read data processed by the component.

Locations variables, or locations for short, are used as "containers" that may transport data and code while moving. The association of a channel x with a location l is described by $x@l$. Intuitively, this means that the position of the space where the values of x are made available is given by the value of l. Every action g is associated with a set of locations $\Lambda(g)$ meaning that the execution of g is distributed over those

locations. 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 and, hence, the movement of any constituent located at an output location is under the control of the component.

Private actions represent internal computations in the sense that their execution is uniquely under the control of the component; shared actions represent possible interactions between the component and the environment. The computational aspects are described by associating with each action *g*:

- a set *D(g)* with the local channels and locations into which executions of *g* can write;
- for each $l \in \Lambda(g)$, an expression of the form

  $$\mathtt{L(g@l),\ U(g@l)} \rightarrow \mathtt{R(g@l)}$$

  Two conditions *L(g@l)* and *U(g@l)* on *X* establish the interval in which the enabling condition *e* of any guarded command that implements *g@l* must lie: the *lower bound L(g@l)* is implied by *e*, and the *upper bound U(g@l)* implies *e*. When the enabling condition of *g* is fully determined we write only one condition. The parameter *R(g@l)* is a condition on *X* and *D(g)'* where by *D(g)'* we denote the set of primed symbols in *D(g)*. As usual, these primed symbols account for references to the values that they take after the execution of the action. When *R(g)* is such that the primed channels and locations in *D(g)* are fully determined, we obtain a conditional multiple assignment, in which case we use the notation that is normally found in programming languages $(||_{x \in D(g)}\ x{:=}F(g,x))$.

A CommUnity design is called a program when, for every $g \in \Gamma$, $\bigwedge_{l \in \Lambda(g)} L(g@l)$ and $\bigwedge_{l \in \Lambda(g)} U(g@l)$ are equivalent, and the relation $\bigwedge_{l \in \Lambda(g)} R(g@l))$ defines a conditional multiple assignment.

In order to illustrate how CommUnity can handle distribution and mobility, we now address the design of a client whose purpose is to search a remote image database for a particular type of images:

```
design client1 is
outloc lf, lc
in     lr:Loc, db:set(image)
out    res@lc, img@lf:set(image), size@lf:nat
prv    st_f@lf:[0..5], st_c@lc:[0..4], home@lc:Loc
do     gof@lf: st_f=0 → st_f:=1
          @lc: st_c=0 → st_c:=1|| lf:=lr
 []    req@lf: st_f=1 → st_f:=2
 []    filter@lf: st_f=2 → st_f:=3|| img:=filterop(db)
 []    rel@lf: st_f=3 → st_f:=4|| size:=imgsize(img)
 []    backf@lf: st_f=4 → st_f:=5
            @lc: st_c=1∧small(size) → st_c:=2|| lf:=lc
 []    goc@lf: st_f=4 → st_f:=5
          @lc: st_c=1∧¬small(size) → st_c:=2|| lc:=lr|| home:=lc
 []    check@lc: st_c=2 → st_c:=3|| res:=checkop(img)
 []    backc@lc: st_c=3∧lc≠home → st_c:=4|| lc:=home
```

The images are modelled through an abstract data type that involves the domain *image*. The database itself is modelled as a value of type *set(image)* that the environment makes available through the channel *db*.

The client is a distributed component: it involves two locations, *lf* and *lc*, both under its control. The location *lf* is where the client runs, through action *filter*, the

"cheap algorithm" – captured by the user-defined operation *filterop* on *set(image)* – that quickly identifies the images that are potentially interesting, which it makes available in the channel *img*. The size of this set is computed by the action *rel* using the operation *imgsize* and made available through the channel *size*. Access to the database itself is requested through action *req*. Release of the database takes place through the action *rel*, i.e. when the size of the extracted set is evaluated. The private channel st_f located at *lf* ensures the correct sequencing of these actions.

As motivated in the introduction, the filtering activity needs to be performed wherever the database is located. This location is made available by the environment through the input channel *lr*. Hence, through action *gof*, the client moves the filtering activity to this location, i.e. assigns *lr* to *lf*.

The action *gof* is distributed between *lf*, where it initialises the filtering process by setting st_f to *1*, and the second location *lc*, where the actual migration is performed. The location *lc* is where the client determines how to proceed. If *size* is small, the client, through action *backf*, sends back the filter with the extracted images by assigning *lc* to *lf*. Otherwise, the *checker* moves itself to the remote host by assigning *lr* to *lc*. The search of the interesting images, performed by the operation *checkop* on *set(img)*, is executed by action *check* at *lc*. The final result is returned, at *lc*, through the channel *res*. Finally, the client returns home if it ever migrated to the remote host. The private channel st_c located at *lc* ensures the correct sequencing of these actions.

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. Hence, the choice of data types determines, essentially, the nature of the elementary computations that can be performed locally by the components, which are abstracted as operations on data elements. We consider that the collection of data types appropriate for the design of a specific system is explicitly specified through a first-order algebraic specification. That is to say, we assume a data signature $\Sigma=<S,\Omega>$, where S is a set (of sorts) and Ω is a $S^*\times S$-indexed family of sets (of operations), to be given together with a collection Φ of first-order sentences specifying the functionality of the operations.

In CommUnity, the space within which movement takes place is explicitly represented with a distinguished sort *Loc*. Location variables are all implicitly typed with this sort. *Loc* models the positions of the space in a way that is considered to be adequate for the particular application domain in which the system is or will be embedded. Together with the definition of the operations on locations, this provides a description mechanism that is expressive enough to establish, for instance, location hierarchies or taxonomies. The only requirement that we make is for a special position $\perp$ to be distinguished that accounts for a special position of the space where context-transparency is supported. That is, the behaviour of any component located at $\perp$ is context-unaware.

In the sequel, we use Θ to refer to the extension of the data type specification with what concerns the space of mobility. In the image search example, the specification Θ includes the specification of Booleans, natural numbers and sets, which are standard data types. Moreover, Θ includes a specific sort *img* accounting for images, the specific operations *filterop* and *checkop* over sets of images for the searching of the

interesting images, and operations *imgsize:set(img)→nat* and *small:nat→nat* that establish the size of a set of images and a threshold over which the set of images is considered too large, respectively. In this way, we have a design of the system that abstracts from specific representation of images and the specific types of images that have to be filtered. In this example, the space of mobility just has to include three different positions. In addition to $\perp$, it is only necessary to account for the host where the client runs and the location of the database server. We have not identified any need for special operations over positions at this stage of the design.

3 Moving Contexts into the Picture

So far we have focused on the features that support the description of mobile systems. The question that concerns us most in this paper is the extent to which the behaviour of these systems is affected by the context surrounding them.

Most approaches to the specification of distributed and mobile systems adopt context-transparency as an abstraction principle and define the behaviour of systems regardless of their context (e.g., [12], [16], [20]). This implies that network connectivity between two hosts is guaranteed whenever it is needed, and that it is possible to migrate code anywhere and anytime without restrictions. The few formalisms that adopt a context-aware approach do not address contexts explicitly and assume specific notions of context (e.g., [3], [4], [5], [17]). This is also the case of our previous work in CommUnity [15]. In this section, we analyse several issues that are central to the choice of an abstract notion of context and the development of design primitives for context-awareness. This a first step towards a more expressive model for context-aware computing that supports the definition of application specific notions of context and the design of components that deal with changes in the operating conditions as part of their intrinsic behaviour.

We should start by making clear that, by context, we refer to any collection of characteristics and properties that are relevant to the system and are not under its direct control. For instance, in network applications that are able to establish firewalls or security policies, neither the locations of the firewalls nor the structure of the space can be considered as part of the context. However, latency can be considered as part of the context of systems that use the network. Although these systems necessarily affect latency, these effects are achieved in an indirect way.

We also assume that a system is not involved in the monitoring of the contextual information. Any such activity has to rely on another system – the *context-provider* – that supports the gathering of context information from relevant sources (e.g. from the network layer or physical sensors) and its delivery to the system. Such a clear separation of context monitoring from the rest of the system is important because it contributes to the taming of the complexity of designing and building context-aware mobile systems: software designers only have to define the notion of context that best fits the system at hand and do not have to be concerned with the way context information needs to be sensed. This separation is important also because it promotes the development of general context sensing systems that can be used (and reused) as the

basic building blocks in the construction of application-specific context-provider systems.

The fact that we are considering distributed systems implies that contexts are also distributed, which raises the question of whether this distribution should be abstracted or not. The design primitives that can be made available for modelling context-awareness clearly depend on this decision. A common choice in this respect is to consider that a system has transparent access to the distributed context information (e.g., [8]). This approach abstracts the fact that some properties considered relevant for the system, for instance network connectivity, affect necessarily the gathering of remote context information. In CommUnity, we adopt an alternative approach in which all parts of a context can be sensed locally. Notice that this does not mean that distant entities cannot affect the behaviour of a component but, rather, that the transmission of any remote context information that is relevant has to be explicitly designed as part of the behaviour of the system.

Any notion of context is constrained by the unit of mobility. In models with a fine-grained unit of mobility, such as CommUnity, the software designer should write a single context definition that applies to the entire system. This is because a component may have several constituents distributed over different locations that can be moved independently and, hence, the context perceived by a component results from what its constituents perceive in their current locations.

The notion of context in mobile computing encompasses two different aspects. The *active* aspects includes the properties that affect the behaviour of a system even if the design of the system does not explicitly use any context information. In CommUnity, the identification of the *active* characteristics, and the definition of how these characteristics influence the behaviour of a system, are part of the formal semantics. For instance, a specific command may be defined to have different results according to the context in which it is executed. The *passive* part of the context includes the characteristics that only affect the behaviour of a system if we use them explicitly in the design. That is, the way in which each *passive* characteristic affects the behaviour of the system is explicitly coded in the design of the system. In CommUnity, we propose that abstract data types be used for defining design primitives through which different notions of context can be modelled explicitly as part of the application domain. Through the specification of abstract data types, software developers can define the structure of the contextual information demanded by the system and the operations that are needed to access this data.

To be more precise, in CommUnity, the context definition is an explicit and central part of the design of any context-aware system. A single notion of context, that applies to the entire system, is defined by a data type specification χ and a subset O of its operations. We take χ in the form of a first-order algebraic specification. Each operation symbol *obs* in O represents an observable that can be used to describe the behaviour of the system. More concretely, in a CommUnity design defined over a context specification $<\chi,O>$, the conditions that establish the guards and effects of the actions are built with terms involving operations of Θ and O.

We require that every context description χ includes standard specifications of sets and natural numbers (extended with ∞) and we assume they are represented by *set(T)*

and nat^{∞}, respectively. This is because we require four special observables – *rs:* nat^{∞}, *sv:set(*Ω*)*, *bt: set(Loc)* and *reach: set(Loc)* – be included in O. These observables constitute the active aspect of the context and capture the fact that, in location-aware systems, regardless of their particular application domains:

- *Computations*, as performed by individual components, are constrained by the *resources* and *services* available at the positions where the components are located. For instance, a piece of mobile code that relies on high-precision numerical operations may fail when placed in a location where memory is scarce, computations will not be able to proceed if the operations that the code requires are not available.
- *Communication* among components can only take place when they are located in positions that are *"in touch"* with each other. For instance, the physical links that support communication between the positions of the space of mobility (e.g., wired networks, or wireless communications through infrared or radio links) may be subject to failures or interruptions, making communication temporarily impossible.
- *Movement* of components from one position to another is constrained by *"reachability"*. Typically, the space of mobility has some structure that can be given by walls and doors, barriers erected in communication networks by system administrators, or the simple fact that not every position of the space has a host where code can be executed.

The purpose of *rs* and *sv* is to represent respectively, the *resources* and *services* that are available for computation. The observable *rs* quantifies the resources available. It may be defined as a function of other observables in χ (for instance, the remaining lifetime of a battery or the amount of memory available) through the inclusion in χ of whatever axioms are appropriate. The observable *sv* represents the services available and it is taken as a subset of the operations of the data type signature Σ. This is because, as we have seen in Section 2, the services that perform the computations are abstracted as operations on data elements. The intuition behind *bt* and *reach* is even simpler: both represent the set of locations within reach. The former represents the locations that can be reached through communication while the latter concerns reachability through movement. The actual meaning of these four special observers is defined by the formal semantics of CommUnity designs.

Before embarking on the definition of the semantics of CommUnity, we consider again the image search problem. We address the design of the solution presented in the introduction, which relies on the observation of two properties of the context: network latency and processing power. In this case, the specification of the context is rather simple. We define two specific observers, both of them constants and returning natural values.

```
observer operations
   lat: nat  // latency
   ppw: nat   // processing power available
```

Additionally, we may specify a relation between the possibility of communication and latency. In the same way, we can relate the processing power with the special observable *rs*.

```
axioms l:Loc
   belongs(l,bt)∧l≠⊥ ⊃ positive(lat)
   positive(rs) ⊃ positive(ppw)
```

In the envisaged system, the choice of where to execute the compute-intensive algorithm is based on the size of the selected images, on the processing power available in the remote and local machines and on the latency measured in the local host. The concrete criteria have to be available in the form of an operation in the data type specification Θ.

```
operations
   crit: nat*nat*nat*nat->bool    //Is it ok to compute locally?
```

The design *client1* of the client that we gave in Section 2 is now modified in order to accommodate the new requirement.

```
design cwt-client is
outloc lf,lc
in     lr: Loc, db:set(img)
out    res@lc, img@lf:set(img),
       size@lf:nat, rpw@lf:nat
prv    st_f@lf:[0..5], st_c@lc:[0..4], home@lc:Loc
do     gof@lf: st_f=0 → st_f:=1
          @lc: st_c=0 → st_c:=1|| lf:=lr
 []    req@lf: st_f=1 → st_f:=2
 []    filter@lf: st_f=2 → st_f:=3|| img:=filterop(db)
 []    rel@lf: st_f=3 → st_f:=4|| size:=imgsize(img)||rpw:=ppw
 []    backf@lf: st_f=4 → st_f:=5
            @lc: st_c=1∧crit(size,pw,rpw,lat) → st_c:=2|| lf:=lc
 []    goc@lf: st_f=4 → st_f:=5
          @lc: st_c=1∧¬crit(size,pw,rpw,lat) → st_c:=2|| lc:=lr|| home:=lc
 []    check@lc: st_c=2 → st_c:=3||res:=checkop(img)
 []    backc@lc: st_c=3∧lc≠home → st_c:=4||lc:=home
```

This design introduces a new channel *rpw* through which the *filter*, once in the remote location, sends to the *checker* the processing power that is measured there. The enabling condition of the actions that model the return of the *filter* and the *migration* of the *checker* were changed to reflect the new criteria for migration.

Consider now a design *client* only differing from *client1* in the choice of where to run the checker operation, which is made nondeterministic. That is to say, in *client*, actions *backf* and *goc* have exactly the same enabling condition:

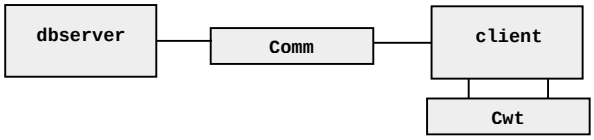
```
       backf@lf: st_f=4 → st_f:=5
            @lc: st_c=1 → st_c:=2|| lf:=lc
 []    goc@lf: st_f=4 → st_f:=5
          @lc: st_c=2 → st_c:=2|| lc:=lr|| home:=lc
```

Both *client1* and *cwt-client* can be obtained from *client* through the superposition of additional behaviour. From a methodological point of view, what is interesting is that it is possible to capture the superposed aspects as an architectural element (a connector) that is plugged to the *client* to control in which situations the *checker* have to migrate. Changing from one design decision to another is then just a matter of unplugging a connector and plugging a new one.

For instance, the aspects that need to be superposed to the *client* in order to obtain *cwt-client* are captured by what in CommUnity is called a connector *Cwt* with the following glue:

```
design cwt-glue is
inloc  lc,lf
in     size:nat,
out    rpw@lf:nat
do     rel@lf: true → rpw:=ppw
 []    backf@lc: crit(size,pw,rpw,lat)→ skip
 []    goc@lc: ¬crit(size,pw,rpw,lat)→ skip
```

An architecture of the image search system that makes use of the connector *Cxt* is presented below. The other connector – *Comm* – accounts for the communication protocol that is adopted for the communication between the *filter* component of the *client* and the database server. This architecture shows how the introduction of context-awareness in the architectural model of a system can be achieved in a non intrusive way.



The advantage of this architecture is that in order to change the design decision we adopted for the migration of the checker, we just need to replace *Cwt* by an appropriate connector.

4 Semantic Aspects

Consider a CommUnity program P defined over a data specification $\Theta=<\Sigma,\Phi>$, where $\Sigma=<S,\Omega>$, and a context specification $Cxt=<\chi,O>$, where $\chi=<\Sigma^{\chi},\Phi^{\chi}>$.

In order to define the behaviour of P, we have to fix, first of all, the carrier sets U_s that define the possible values of the each data sort s. In particular, the set U_{Loc} defines the positions of the space of mobility for the situation at hand. These sets of values are considered to be global and invariant over time. In contrast, as shown below, the interpretation of the operation symbols in Σ is considered to be local to each position of the space and may change over time. This accounts for the possible evolution of the actual *implementations* of the operations that perform the computations that are required in P.

We also consider that part of the data type specification that defines the context has a global and static interpretation. The exception is the interpretation of the observables that account for the actual contextual information and which is considered to be local and dynamic. More concretely, we fix an algebra U' for the sub-specification of χ that is obtained by forgetting the subset of the operation symbols of O and the axioms involving these symbols. In the sequel, we designate this specification by χ'. The algebra U' should provide the standard interpretation to nat^{∞} and *set(T)*.

Because P is a context-aware program, the surrounding context can affect its behaviour. Therefore, we also need to fix a model of the "world" where P is placed to run. In fact, we have to provide a model for the context defined by *Cxt*.

The model of the context should capture the fact that it may change continuously. In the trace-based semantics of CommUnity, we take context models in the form of infinite sequences of states; such states capture the contextual information at a particular instant of time. As motivated in Section 3, we consider that the state of contextual information is distributed and put together from what is sensed locally at each position of the space.

Local states have three dimensions. The first addresses the interpretation of the operations of Σ. It consists of an algebra U for the part of Θ that captures the operations that are available in the state. This algebra is based on the carriers sets U_s that were fixed before. This partiality captures that some locations may not have local implementations of some types of computations that are required in P.

The second dimension concerns the level of resources required for the computation of each operation in Σ. It consists of a partial function $\rho:\Omega\rightarrow N^{\infty}$ that must be defined for every operation symbol for which the algebra U establishes an interpretation.

The third dimension is about the contextual information that can be observed by the program P. It provides the current values of the observables in a particular location and consists of an O-indexed set $o=\{obs_U\}_{obs\in O}$ of functions defining an interpretation for each observable obs in O based on the carrier sets defined by U'. This set, together with the χ'-algebra U', should define a χ-algebra. Furthermore, the interpretation of the special observables *rs*, *sv*, *bt* and *reach*, provided by the constant functions $rs_U:N^{\infty}$, $sv_U:2^{\Omega}$, $bt_U:2^{Loc_U}$ and $reach_U:2^{Loc_U}$ is constrained as follows.

- The set of available services, sv_U, must be the set of Σ-operations for which the algebra U establishes an interpretation.
- In any local state associated to position m, the sets of positions bt_U and $reach_U$ must include m. Intuitively, this means that we require that *be in touch* and *reachability* are reflexive relations. Furthermore, bt_U must include the special position $\perp_U$. This condition establishes part of the special role played by $\perp_U$: at every position of the space, the position $\perp_U$ is always "in touch". In addition, we require that in any local state associated to position $\perp_U$, bt_U be the set U_{Loc}. In this way, any entity located at $\perp_U$ can communicate with any other entity in a location-transparent manner and vice-versa.
- The position $\perp_U$ is also special because it supports context-transparent computation, i.e. a computation that takes place at $\perp_U$ is not subject to any kind of restriction. This is achieved by requiring that the values of rs_U and sv_U in any state associated to the position $\perp_U$ be $+\infty$ and Ω, respectively. In other words, the computational resources available at $\perp_U$ are unlimited and all services are available.

In summary, a context model is an infinite sequence of functions $M_0.M_1\ldots$ in which each M_i is a function over U_{Loc} that returns a three-dimensional state. For ease of presentation of the program behaviour, we use $\alpha_i(m)$ and $\rho_i(m)$ to denote the first and the second component of $M_i(m)$, respectively. Moreover, we use $o_i(m)$ to denote the interpretation of the observable o provided by the third component of $M_i(m)$.

The behaviour of the program P running in a world modelled by $M_0.M_1\ldots$ is defined in terms of set of traces as follows.

We take traces in the form of $V_0.\gamma_0.V_1.\gamma_1\ldots$ where each V_i is a valuation of the channels and locations of P (an S-indexed set of functions $V_s:V_s\rightarrow U_s$) and γ_i is a set

of actions of P. Notice that transitions of the form $V_i.\varnothing.V_{i+1}$ capture state transitions that are performed by other components of the systems in which P is integrated as a component.

The terms and propositions used for defining the guards and effects of the actions of P are built over the signature Σ and the set of observables of Cxt. The local interpretation of these terms and propositions over a trace, i.e. from the point of view of a specific position of the space, can be defined in a straightforward way. It should just be noted that the interpretation $I_i(t)(m)$ of the term t at position m at time i over a trace depends not only on the valuation V_i of the channels and location variables of the program but also on the local state of the context at time i. This is because, on the one hand, the interpretation of the data operations of Σ is defined by $\alpha_i(m)$ and, on the other hand, the actual values of the observables are defined by $o_i(m)$. In particular, if t involves data operations for which $\alpha_i(m)$ does not establish an interpretation, then $I_i^m(t)$ is undefined.

The same applies to propositions. A proposition that involves one of these terms cannot be evaluated at position m and time i. We use $S,i,m \vdash \phi$ to denote that proposition ϕ is evaluated to true at position m and at time i of S.

Formally,

Given a trace $V_0.\gamma_0.V_1.\gamma_1\ldots$, an action g of P is enabled at time i iff

- *for every $l_1,l_2 \in \Lambda(g)$, $V_i(l_2) \in bt_i(V_i(l_1))$ and $V_i(l_1) \in bt_i(V_i(l_2))$*
- *for every $l \in \Lambda(g)$, $g@l$ is enabled at time i, i.e.,*
  - ***(a)*** *$S,i,V_i(l) \vdash L(g@l)$;*
  - ***(b)*** *for every $x \in D(g)$, $I_i(F(g@l,x),V_i(l))$ is defined;*
  - ***(c)*** *for every $f \in \Omega$ used in $L(g@l)$ or $F(g@l,x)$ and every $x \in D(g)$, $\rho_i(m)(f) \leq rs_i(V_i(l))$;*
  - ***(d)*** *for every $x \in local(V)$ used in used in $L(g@l)$ or $F(g@l,x)$ and $x \in D(g)$, if $l' \in \Lambda(x)$ then $V_i(l') \in bt_i(V_i(l))$;*
  - ***(e)*** *for every location $l' \in D(g)$, $I_i(F(g@l,l')) \in reach_i(V_i(l'))$.*

The intuition behind these conditions, under which a distributed action g can be executed at time i, are the following:

- the execution of g involves the synchronisation of its local actions and, hence, their positions have to be mutually in touch;
- the local guards evaluate to true (in particular, they can be evaluated);
- the operations necessary to perform the computations that are required by $g@l$ are available as well as the resources they demand;
- the execution of the guarded command associated with $g@l$ requires that every channel in its frame can be accessed from its current position and, hence, l has to be in touch with the locations of each of these channels;
- if a location l' can be effected by the execution of $g@l$, then the new value of l' must be a position reachable from the current one.

Formally,

A trace $V_0.\gamma_0.V_1.\gamma_1\ldots$ is a behaviour of P iff, for every

- *$i \in \omega$, $V_{i+1}(x) = V_i(x)$ for every $x \in local(L \cup X) \setminus \cup_{g \in \gamma_i} D(g)$ and for every $g \in \gamma_i$:* ***(a)*** *g is enabled at time i;* ***(b)*** *$S,i,V_i(l) \vdash R(g@\lambda)$.*

- $g \in prv(\Gamma)$, if g is infinitely often enabled then there are infinitely many i s.t. $g \in \gamma_i$.

This defines that the execution of an action consists of the transactional execution of its guarded commands at their locations, which requires the atomic execution of the multiple assignments. Moreover, private actions are subject to a fairness requirement: if infinitely often enabled, they are guaranteed to be selected infinitely often.

5 Conclusions

In this paper, we addressed the design of context-aware systems by proposing design primitives that support the explicit description of contexts as part of the application domain. The idea of having individualized context definitions as part of system designs and the corresponding contextual information transparently provided by the context has many advantages. The maintenance of contextual information tends to be a complex task. For instance, it may require the interaction with heterogeneous physical sensors or the network layer. The separation of the design and construction of the context-provider system from the rest of the application helps to cope with this complexity and allows that context-provider systems be reused in different applications.

Having contexts defined through data type specifications, we showed that the mechanisms available in CommUnity to specify how a system should behave in different situations are applicable also when these situations are characterised by different context states. Moreover, we illustrated around an example, how these mechanisms support the introduction of context-awareness in architectural models in a non-intrusive way.

The importance of a clear separation of the context-aware aspects of system behaviour from the other aspects has been widely recognised. In infrastructure-centred approaches, e.g. [13] and [18], this separation is achieved through the adoption of special mechanisms for the specification of how context influences the behaviour of an application, different from the mechanisms available for the design of the application. However, in general, these approaches do not provide an abstract semantics of these mechanisms and, often, not even address their "physiological structure". For instance, in [18], context-awareness is specified through rules consisting of a context expression and a set of actions that must be performed when the context expression becomes true. However, the notion of action is left undefined and it is not explained to which extent the execution of these actions can interfere with the application behaviour.

In fact, much of the work that has been done in the area of context-aware computing has been devoted to the development of middleware infrastructures that facilitate the implementation of context-aware software by taking the responsibility for the gathering and dissemination of contextual information (e.g. [8], [9]). This work is generally based on rigid and narrow notions of context.

In what concerns the development of design frameworks that support the design of context-aware systems, we are only aware of Context Unity [19]. Context Unity also

considers that context-aware systems should be designed assuming that context maintenance is provided by underlying support systems. The way the context can affect the behaviour of a component is, as in CommUnity, part of the component definition but with a completely different perspective on the notion of context. In Context Unity, the operational context with which a component may interact is defined by a set of observables whose values exclusively depend on the values of variables of other components in the system. However, in CommUnity, we consider that the operational context of a design is not under control of any part of the system in which the design is integrated as a component. The advantage of Context Unity is that it is possible to make precise for the context-provider system what the different observables have to be. The disadvantage is that it is only suitable for situations in which the context of a system is, to some extent, expressible in terms of the application domain. This is the case of the running example of [19], an application in which each component has to send messages to a group of components. This group is considered to be a part of the context of the component and is defined in terms of the messages that the component receives. For instance, any component from which a message is received is added to the group; any component that leaves a certain region around the component is removed from the group.

So far we have only addressed the use of context information at the level of the description of components and connectors, i.e., the building blocks of system architectures. It is also important to be able to take advantage of contextual information at the (re)configuration level, namely to use context information to program the dynamic reconfiguration of the system architecture. This includes, for instance, the possibility to react to context changes by removing deployed components or adding new ones, or yet by replacing the connectors in place. For instance, we may wish to specify that a GUI should be replaced by a TextualUI when the battery is low. Some of our future work will progress in this direction.

Acknowledgements

This work was partially supported through the IST-2001-32747 Project *AGILE – Architectures for Mobility*. We wish to thank our partners for much useful feedback.

References

[1] IST Global Computing Initiative, http://www. cordis.lu/ist/fet/gc.html, 2001.

[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] G.Boudol, “ULM A Core Programming Model for Global Computing”, available at http://www-sop.inria.fr/mimosa/personnel/Gerard.Boudol.html.

[4] L.Cardelli and A.Gordon, “Mobile Ambients”, in Nivat (ed), *FoSSACs’98*, LNCS 1378, 140-155, Springer-Verlag, 1998.

[5] L.Cardelli and R.Dawies, "Service combinators for web computing", *IEEE Transactions on Software Engineering* **25**(3), 303-316, 1999.

[6] K.Chandy and J.Misra, *ParallelProgram Design - A Foundation*, Addison-Wesley, 1988.

[7] G.Chen and D.Kotz, "A Survey of Context-Aware Mobile Computing Survey", Dartmouth CS-TR 2000-381, 2000.

[8] G.Chen and D.Kotz. "Context-sensitive resource discovery", *Proc. 1st IEEE International Conference on Pervasive Computing and Communications*, 243-252, 2003.

[9] A.Dey, D.Salber and G.D.Abowd, "A conceptual framework and a toolkit for supporting the rapid prototyping of context-aware applications", *Human-Computer Interaction* **16**(2-4), 97-166, 2001.

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

[11] J.L.Fiadeiro, A.Lopes and M.Wermelinger, "A Mathematical Semantics for Architectural Connectors", in R.Backhouse and J.Gibbons (eds), *Generic Programming, LNCS* 2793, 190-234, Springer-Verlag, 2003.

[12] C.Fournet, G.Gonthier, J.-J.Levy, L.Maragent and D.Remy, "A calculus of mobile agents", *CONCUR'96*, LNCS 1119, 315-330, Springer-Verlag, 1996.

[13] K.Henricksen, J.Indulska and A.Rakotonirainy, "Modeling Context Information in Pervasive Computing Systems", *Proc. of Pervasive 2002*, 167-180, 2002.

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

[15] A.Lopes and J. L. Fiadeiro, "Adding Mobility to Software Architectures", *ENCTS* **97**, 241-258, 2004.

[16] R.De Nicola, G.L.Ferrari and R.Pugliesi, "KLAIM: A Kernel Language for Agents Interaction and Mobility", *IEEE Transactions on Software Engineering* **24**(5), 315-330, 1998.

[17] G.P.Picco, A.L.Murphy and G.-C.Roman, "Lime: Linda meets Mobility", *Proc. ICSE* 1999, 368-377, 1999.

[18] A.Ranganathan and R.Campbell, "An infrastructure for context-awareness based on first order logic", *Pers Ubiquit Comput*ing **7**, 353-364, 2003.

[19] G.-C.Roman, C.Julien and J.Payton, "A Formal Treatment of Context-Awareness", *Proc. FASE 2004*, LNCS 2984, 12-36, Springer-Verlag, 2004.

[20] G.-C.Roman, P.J.McCann and J.Y.Plun, "Mobile UNITY: reasoning and specification in mobile computing", *ACM TOSEM*, **6**(3), 250-282, 1997.