

Hybrid Models for Mobile Computing

Mika Katara

Software Systems Laboratory
Tampere University of Technology
P.O. Box 553, FIN-33101 Tampere, Finland
Tel. +358 3 365 3822, Fax +358 3 365 2913
`mika.katara@cs.tut.fi`

Abstract. Hybrid specifications, i.e. ones containing both discrete and continuous changes, are used mainly in modelling systems that control some physical phenomena. In this paper, we propose hybrid modelling of novel wireless mobile systems, which generally fall under the term mobile computing. The particular systems consist of agents capable of moving in a physical reality and communicating wirelessly when in each other's proximity. In this paper we concentrate on agents not capable of controlling their own movement, e.g. the ones designed to be carried around. Their environment comprises of a user and the physical reality whose nature is continuous rather than discrete. An approach to modelling of such systems is presented. The approach, which is based on the DisCo method, takes into account the continuous nature of the environment. Joint actions and closed system modelling are used to coordinate multi-agent interactions at a high level of abstraction. An example is presented where the approach is illustrated by a specification modelling file transfer operation between two agents.

1 Introduction

Hybrid specifications, i.e. ones containing both discrete and continuous changes, are used mainly in modelling systems that control some physical phenomena. In this paper, we propose hybrid modelling of novel wireless mobile systems, which generally fall under the term mobile computing.

The systems of interest are those consisting of agents capable of moving in a physical reality and communicating wirelessly when in each other's proximity. Consider for instance PDAs (Personal Digital Assistants) communicating with each other through short range radios. In this paper we concentrate on agents not capable of controlling their own movement, e.g. the ones designed to be carried around. Their environment comprises of a user and the physical reality whose nature is continuous rather than discrete.

There is interaction between an agent and its continuous environment. Applications available to the user of the agent may depend on the physical characteristics of the agent's environment. For example, communication between other agents depends on mutual distance which changes continuously. Capturing this

interaction in the specifications calls for hybrid models. In this paper an approach to hybrid modelling of mobile systems is presented.

The approach is based on the DisCo¹ method [8, 1], a formal specification method for distributed *reactive systems*, i.e., systems that are in constant interaction with their environments. The basis of DisCo is the *joint action* theory [4, 5], which enables the specification of *collective behavior* at a high level of abstraction. Specifications describe *closed systems*, i.e. systems together with their environments. Joint actions and closed system modelling are used to coordinate multi-agent interactions at high levels of abstraction. Specifications are refined towards implementation using a form of *superposition* [7], which preserves all *safety* properties (“something bad will never happen”) by construction, while *liveness* properties (“something good will eventually happen”) lead to proof obligations.

The rest of the paper is structured as follows. In Section 2, an introduction to the DisCo method is given. The hybrid approach to modelling of mobile systems is presented in Section 3. In Section 4, an example is presented. Conclusions are stated in Section 5.

2 Introduction to DisCo

2.1 Classes and Actions

The DisCo method [8, 1] is based on the joint action theory [4, 5] which has been found suitable for specifying and reasoning about reactive and concurrent systems. In DisCo, the focus is set on collective behaviour at high levels of abstraction, where objects communicate by *participating* in atomic actions. The formal basis of the DisCo language is in the *Temporal Logic of Actions*, *TLA* [13]. TLA is a linear time logic where attention is on infinite sequences of states called *behaviours* and their properties. An infinite number of *state variables* is assumed and in each state of a behaviour every variable has a unique value.

A DisCo specification has an operational interpretation. All variables are encapsulated in objects. DisCo also has *classes* and (multiple) *inheritance*, familiar from other object-oriented languages. Methods are replaced by multi-object actions, having neither callers nor callees. Objects can be associated with each other using *relations* and *references*. The life cycle of objects begins at *creation*, where the initial state of the specification is given, they cannot be created or destroyed at run time.

In an action each participant is assigned a *role*. Objects are capable of participating in actions in certain roles. No object can participate in more than one role at a time. Actions may also have *parameters* which refer to immutable values rather than mutable objects. Parameters, which have nondeterministic values, can be used to introduce nondeterminism.

In an action definition the roles and the classes of participating objects are given. A Boolean expression, called the *guard*, and the *body* of the action are also

¹ Acronym for *Distributed Co-operation*.

given. In the body, unprimed and primed variable names are used to distinguish values of variables before and after executing the action, respectively. Functions can be used to abbreviate complex expressions.

Actions in DisCo are atomic, i.e., their executions are bound to be finished without outside interference. Concurrency is modelled by *interleaving* the executions of actions. An action is said to be *enabled* if there exist potential participants and values for parameters so that the guard evaluates to true. If more than one action is enabled the one to be executed is selected nondeterministically.

If a role name is given in braces, the role is said to be *quantified*. It means that a nondeterministic (possibly empty) set of objects that satisfy the guard can participate in the role. The guard and the body are evaluated for each object participating in a quantified role.

As an example, consider a simple specification given below, which models sending and delivering messages holding integer values. It includes classes *Message* and *Receiver* and actions *Sending* and *Delivery*. The parameter *data* of action *Sending*, the value of which is copied to the message participating as *m*, models the data to be sent. In action *Delivery* the value within participant *m* is delivered to all receivers participating in the quantified role *r*. The sequence *received* models the sequence of received values in each *Receiver*:

$$\text{class } Message = \{data : integer\}$$

$$\text{class } Receiver = \{received : \text{sequence } integer\}$$

$$Sending(m : Message; data : integer) :$$

$$\begin{aligned} & m.data = 0 \\ & \wedge data \neq 0 \\ & \rightarrow m.data' = data \end{aligned}$$

$$Delivery(m : Message; \{r\} : Receiver) :$$

$$\begin{aligned} & m.data \neq 0 \\ & \rightarrow r.received' = r.received + \langle m.data \rangle \\ & \wedge m.data' = 0 \end{aligned}$$

Messages that are currently in use are distinguished from others by non-zero value of variable *data*. These variables should be initialized as zero, which can be required by an initial condition:

$$\text{initially } \forall m \in Message : m.data = 0$$

The execution model does not guarantee that all enabled actions are finally executed. Fairness requirements have to be given to ensure liveness. Prefixing the name of a (possibly quantified) role with an asterisk indicates that if an action is infinitely often enabled so that the same object can participate in the

same prefixed role, then the action has to be executed infinitely often with this object in the prefixed role. If fairness is required for more than one participant, the above is required for each combination of such participants. Explicit fairness requirements are the biggest difference between the execution models of DisCo and UNITY [6].

2.2 Timed Specifications

Real time is a continuous quantity that can be incorporated in the above scheme as follows (for more detailed discussion, see e.g. [10]). Each action is assumed to be executed instantaneously. A clock variable $\Omega \in \mathbb{R}^{0+}$, belonging to the set of nonnegative reals and initialized as 0, is introduced to record time from the beginning of a behaviour. An implicit parameter τ representing the time when an action is executed is added to each action. Also, all guards are implicitly strengthened by the conjunct

$$\Omega \leq \tau \leq \min(\Delta) ,$$

where Δ denotes a multiset of *deadlines*. Moreover, conjunct $\Omega' = \tau$ is added to the bodies of all actions.

Minimal separation requirement between actions A and B can be enforced by strengthening the guard of action B by conjunct $\tau \geq \tau_A + d$, where τ_A denotes the most recent execution moment of A . For bounded response requirements, deadlines are used. When a deadline $\tau + d$ is needed for some future action, a conjunct of the form $x' = \Delta_{on}(d)$ is given in the action body to add this deadline to Δ and to store it in a variable x . An implicit conjunct $\tau \leq \min(\Delta)$ in all guards then prevents advancing Ω beyond this deadline, until some action has removed the deadline with $\Delta_{off}(x)$. In the initial state, Δ can hold initial deadlines. Furthermore, a type time, a synonym type of real, can be used in timed specifications.

It should be noted that fairness still remains the only execution force. Actions are not executed because time passes, but the passing of time is noticed as a result of executing an action. This may lead to *Zeno behaviours*, where time is not allowed to grow beyond any bound. However, as discussed in [3], Zeno behaviours are harmful only if there are no other alternatives where time may grow unboundedly.

2.3 Superposition and Layers

In DisCo, the specification process is started at a high level of abstraction. Specifications are *stepwise refined* towards implementation applying the *superposition* principle. In superposition, classes can be extended with new variables, and totally new classes can be added. New actions can be given, and new participants and parameters can be added to the old actions. Guards can be strengthened and action bodies extended. Different refined versions for one action can be given.

However, actions are not allowed to modify old variables. This restriction is necessary to ensure the preservation of safety properties. Superposition relation is trivial to check mechanically.

A DisCo specification consists of a set of superposition steps called *layers*. One specification can be refined in several parallel layers yielding several branches of the specification. These branches correspond to different aspects of the system. Different specifications and different branches of the same specification can be *composed* together. In composition, actions can be *synchronized* to be executed in parallel.

To illustrate the use of superposition, a simple refinement of the previous specification is presented. In superposition, actions can be *specialized*. Specializing means introducing a new action that is specialized for a subclass, i.e., it is enabled only for objects of the subclass. The guards of the other versions of the action are implicitly strengthened so that they are no longer enabled for the subclass.

A subclass `FirstClassMessage` of `Message` is introduced and both `Sending` and `Delivery` are specialized for it. Moreover, every sent `FirstClassMessages` *fcm* must be read by some number of `Receivers` within certain time interval from sending. The minimum number of `Receivers` and the interval are given by variables *read_by* and *d* of class `FirstClassMessage`, respectively. The above is satisfied by a simple real-time requirement and a conjunct in the guard of action `FirstClassDelivery`. The conjunct requires that the number of objects participating in role *r* is greater than or equal to the value of *fcm.read_by*²:

```
class FirstClassMessage = Message +
    {d : time;
     t : time;
     read_by : integer}
```

```
FirstClassSending(fcm : FirstClassMessage; data : integer) :
    refines Sending(fcm, data)
→ fcm.t' = Δon(fcm.d)
```

```
FirstClassDelivery(*fcm : FirstClassMessage; {r} : Receiver) :
    refines Delivery(fcm, r{})
∧ |r{}| ≥ fcm.read_by
→ Δoff(fcm.t)
```

After this superposition step there are two versions of `Sending` and `Delivery`, the ones for `Messages` which are not `FirstClassMessages`, and the others, named

² The notation $r^{\{}}$ refers to the set formed by all objects participating in role r .

FirstClassSending and FirstClassDelivery, for FirstClassMessages. As discussed earlier, a fairness requirement is needed for a participant in action FirstClassDelivery to force the action into execution.

3 Hybrid Modelling of Mobility

3.1 Capturing Hybrid Interaction

In *closed system* modelling the system under development is specified together with its assumed environment. The greatest advantage of closed specifications over *open specifications* is that the specification process can be started without first fixing the interfaces between different components of the system.

In DisCo, joint actions allow specification of *interaction* between agents at a high level of abstraction. Specifications are always closed allowing to capture the essential collective behaviour before partitioning the specification into components with fixed interfaces [11, 12].

The behaviour of a system consisting of mobile agents depends on the physical localities of the agents. Locations of the agents change as they move. The movement of agents may affect the enabledness of actions. Consider for example a situation where the locations of agents A and C stay the same and agent B moves from the proximity of A to the proximity of C. An action modelling communication between two agents which are in each other's proximity, may first be enabled for A and B, later disabled for all agents and again later enabled for B and C. Alternatively, the action may at some point be enabled for both A and B, and B and C.

A fundamental question is whether an agent can *control* or only *observe* its motion. The former approach relates to mobile robotics and controller design. The latter relates to systems designed to be carried around in a pocket or, for instance, moved in a car or train. In this case the system can only give suggestions to the user on where to move next, but the decision to do so lies beyond its capabilities. In the sequel we will concentrate on the latter scenario.

An environment of such an agent consists of a user, other agents, and the physical reality in which it exists. The user uses the applications provided by the agent and decides to move in a physical reality to a certain direction with a certain speed. Depending on the location, the agent may be able to communicate with other agents. The applications that are available to the user may depend on the proximity of other agents. The physical reality serves as a medium for wireless communication between agents.

The behaviour of an agent can be described as a sequence of states. A state consists of values of variables. Actions are used to describe how states in a behaviour relate to each other. The values of variables can change only in actions. The nature of phenomena in the physical reality is continuous, i.e. changes do not occur only in discrete steps but also between them.

There is interaction between an agent and its continuous environment. Consider for example an agent transmitting data to another agent. On one hand the

transmission causes changes in the environment, which make it possible for the other agent to receive the transmission. On the other hand the changes in the signal to noise ratio caused by movement of the agent, for instance, can affect the data transfer rate between the agents.

Capturing this interaction in the specifications of mobile agents calls for hybrid models, i.e. models capable of describing not only discrete changes but also continuous ones. In the DisCo approach, joint actions and closed system modelling are used to capture hybrid interaction.

3.2 Superposing Localities

A group of mobile agents communicates in joint actions. Mobile agents are modelled as objects with localities. Their behaviours and willingness to communicate depends on their locations relative to other objects.

We will assume that the movement of an agent is continuous, but the agent has no control over it. However, the agent is always capable of measuring its location. The measuring takes place in actions. The current location of a participating agent can be modelled as a parameter and assigning a parameter value to a state variable corresponds to a measurement.

Based on the measurements, an agent can estimate its future motion or even guide movement of the user. This approach, where ordinary state variables and time are used to estimate continuous quantities was applied in [9] to general hybrid systems.

Actions are used to coordinate both individual and joint behaviour of agents. Aspects related to physical location, the corresponding state variables and coordination can be given in one or more superposition steps.

For simplicity and without loss of generality, we will concentrate on agents moving in a two-dimensional Euclidean space. Type `Physical_Location` describes the actual physical location of an instance of class `Object` in a two-dimensional space. The variables x and y are the values of the coordinates, and ref is a reference to the associated instance:

```
type Physical_Location = {x,y : real
                          ref : reference Object}
```

Whenever an action that has parameters of type `Physical_Location` is executed, the values of the parameters correspond to the actual physical locations of the referenced objects at the moment the action is executed. Additionally, function `Distance` is defined. It gets two physical locations as parameters and returns their mutual distance:

```
function Distance(pl1,pl2 : Physical_Location) : real is
   $\sqrt{(pl1.x - pl2.x)^2 + (pl1.y - pl2.y)^2}$ 
```

3.3 Hybrid Deadlines

Typical inputs to a hybrid system come from sensors monitoring a physical phenomena. Values from sensors are typically obtained at discrete moments, thus actions and non-deterministic parameters can be used to capture the correct behavior. However, modelling an action that has to be executed at a moment when a continuous quantity reaches some limit poses a problem. The problem is determining the right moment of time when the action should happen. Obviously, the moment the quantity reaches the limit is not known beforehand, so we cannot give a time deadline for the action.

One solution is to introduce priorities to actions and require that actions should be chosen for execution based on them. However, we consider priorities as an implementation level mechanism and feel that fairness is a much more convenient execution force at high levels of abstraction.

For other continuous quantities than time we will use *hybrid deadlines* which are analogous to time deadlines. In this paper, they are applied to mutual distances between objects which change continuously as the objects move.

For this purpose, for each pair $o1, o2 \in Object$, a multiset $\Delta^{Distance(o1,o2)}$ is introduced to hold hybrid deadlines concerning the mutual distance between the objects. Furthermore, it is assumed that between executions of actions, the mutual distance between any pair of objects changes *monotonically*.

We can use hybrid deadlines in conjunction with fairness to force some actions into execution before certain distance grows beyond some deadline. Similarly to time deadlines, hybrid deadlines can be added and removed using Δ_{on} and Δ_{off} , respectively. However, the parameter of Δ_{on} is treated as an absolute value, not as an offset from the current value of distance. Every new deadline less than the current value is treated as the current value.

In each action there is an implicit parameter PL which is a set of type `Physical_Location` and it holds that

$$\forall o \in Object \exists pl \in PL : pl.ref = o$$

and

$$\forall pl1, pl2 \in PL : pl1.ref = pl2.ref \Rightarrow pl1.x = pl2.x \wedge pl1.y = pl2.y .$$

Furthermore, in all guards there is an implicit conjunct

$$\forall pl1, pl2 \in PL : Distance(pl1, pl2) \leq \min(\Delta^{Distance(pl1.ref, pl2.ref)}) ,$$

which prevents the mutual distance between any pair of objects to grow beyond any hybrid deadline set for the distance until some action removes the limiting hybrid deadline with $\Delta_{off}^{Distance(pl1.ref, pl2.ref)}$.

The requirement of monotonicity means that, between executions of actions, for every pair of objects their mutual distance is either nonincreasing or nondecreasing. Nonincreasing means that the distance cannot increase and nondecreasing that the distance cannot decrease. If actions are executed with sufficient frequency, this is not a vital restriction.

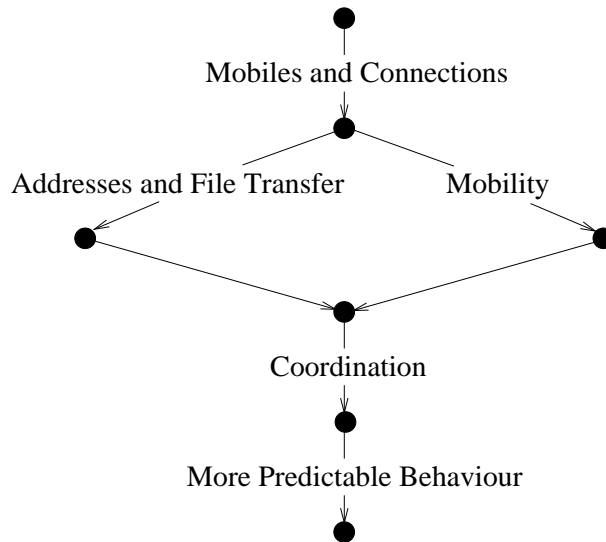


Fig. 1. The layered structure of the specification

If a distance for which there is a hybrid deadline, reaches this deadline and then starts to decrease, the action that removes the deadline may or may not be executed. The deadline mechanism ensures only that deadlines are not passed.

4 Example

To illustrate the use of hybrid modelling in conjunction with mobility, a simple example is presented. The specification describes agents capable of moving and transferring files when in each other's proximity. The specification consists of five layers (see Fig. 1).

4.1 Mobiles and Connections

In the highest level layer named Mobiles and Connections, the agents are modelled as instances of class *Mobile*, a subclass of *Object*. At this level there are no attributes in the class. Relation *Connection* can hold between two instances of class *Mobile*:

$$\text{class } Mobile = Object + \{\}$$

$$\text{relation } (0..1) \cdot Connection \cdot (0..1) : Mobile \times Mobile$$

Action `Connect` sets a relation between two participating Mobiles. It can be executed if neither one is already in relation with some other Mobile:

$$\begin{aligned}
& \text{Connect}(m1, m2 : \text{Mobile}) : \\
& \quad \forall m \in \text{Mobile} : \neg(m1 \cdot \text{Connection} \cdot m \vee m \cdot \text{Connection} \cdot m1 \\
& \quad \quad \vee m2 \cdot \text{Connection} \cdot m \vee m \cdot \text{Connection} \cdot m2) \\
& \quad \rightarrow m1 \cdot \text{Connection}' \cdot m2
\end{aligned}$$

Action `Disconnect` resets a relation between two Mobiles:

$$\begin{aligned}
& \text{Disconnect}(m1, m2 : \text{Mobile}) : \\
& \quad m1 \cdot \text{Connection} \cdot m2 \\
& \quad \rightarrow \neg m1 \cdot \text{Connection}' \cdot m2
\end{aligned}$$

At a high level of abstraction the system comprises only of entities called Mobiles. They can be connected to at most one other Mobile at a time. This safety property will be preserved by all later refinements, because superposition preserves all safety properties by construction.

4.2 Addresses and File Transfer

In layer `Addresses and File Transfer` a file transfer operation between two Mobiles is specified. A Mobile can push a file to another Mobile.

The layer introduces types `Address` and `File`:

type *Address, File*

Moreover, the class `Mobile` is extended with four variables. Variable *address* is a unique physical network address hard coded into each device and *others* is a set of addresses of other instances of class `Mobile` known by this instance. Variable *files* is a set of `File`s and *pushing* is a variable where a file to be pushed to another Mobile is stored. There is also an initial condition requiring that the address fields have unique values:

```

class Mobile = Mobile +
    {address : constant Address;
      others : set Address;
      files : set File;
      pushing : File}

```

initially $\forall m, n \in \text{Mobile} : m.\text{address} = n.\text{address} \Rightarrow m = n$

If a Mobile is not connected, it can broadcast its address to other Mobiles. This is modelled as action `Broadcast_Address`. In the action, the address of the

participant $m1$ is made known to Mobiles participating in the quantified role $m2$:

$$\begin{aligned} & \text{Broadcast_Address}(m1, \{m2\} : \text{Mobile}) : \\ & \quad \forall m \in \text{Mobile} : \neg(m1 \cdot \text{Connection} \cdot m \vee m \cdot \text{Connection} \cdot m1) \\ & \quad \rightarrow m2.\text{others}' = m2.\text{others} + \{m1.\text{address}\} \end{aligned}$$

Action Connect is refined and renamed to Start_File_Push. A file to be pushed and its recipient are chosen nondeterministically from sets $files$ and $others$, respectively. The body of the action assigns the file to variable pushing:

$$\begin{aligned} & \text{Start_File_Push}(m1, m2 : \text{Mobile}; f : \text{File}) : \\ & \quad \mathbf{refines} \text{Connect}(m1, m2) \\ & \quad \wedge f \in m1.\text{files} \\ & \quad \wedge m2.\text{address} \in m1.\text{others} \\ & \quad \rightarrow m1.\text{pushing}' = f \end{aligned}$$

Nondeterministic choice is a natural way to model that the user chooses the file and its recipient from the lists displayed in the graphical user interface of an agent. However, the Mobiles whose addresses are in the set may have connected to other Mobiles after broadcasting their addresses. Nevertheless, the guard of Connect ensures that only those Mobiles that are not connected can be connected to.

Action Stop_File_Push is a refinement of action Disconnect. It completes the file transfer by adding the file that was pushed to the set $files$ of the recipient:

$$\begin{aligned} & \text{Stop_File_Push}(m1, m2 : \text{Mobile}) : \\ & \quad \mathbf{refines} \text{Disconnect}(m1, m2) \\ & \quad \rightarrow m2.\text{files}' = m2.\text{files} + \{m1.\text{pushing}\} \end{aligned}$$

There is also another refinement of action Disconnect, modelling the end of an unsuccessful file transfer operation. It is named Abort_File_Push and it is similar to the original action:

$$\begin{aligned} & \text{Abort_File_Push}(m1, m2 : \text{Mobile}) : \\ & \quad \mathbf{refines} \text{Disconnect}(m1, m2) \end{aligned}$$

Moreover, there is an action modelling removal of obsolete addresses from the set $others$ of a participating Mobile:

$$\begin{aligned} & \text{Remove_Addresses}(m1 : \text{Mobile}; \text{obsolete} : \mathbf{set} \text{Address}) : \\ & \quad \text{obsolete} \subseteq m1.\text{others} \\ & \quad \rightarrow m1.\text{others}' = m1.\text{others} - \text{obsolete} \end{aligned}$$

4.3 Mobility

In layer Mobility the variables needed to store the measured location information are added. Type *Location* is used to store values of physical locations and also the speed of the Mobile relative to both x and y axes:

$$\mathbf{type} \textit{Location} = \{x, y : \mathit{real}; \\ x_speed, y_speed : \mathit{real}\}$$

Variables *x* and *y* should be initialized to reflect the initial physical location of the Mobile.

Class *Mobile* is extended to include variable *l* of type *Location*. Variable *period* indicates the time period between successive updates of the location information and variable *deadline* holds the next moment of time when the update is to be done:

$$\mathbf{class} \textit{Mobile} = \textit{Mobile} + \\ \{l : \textit{Location}; \\ \textit{period} : \mathit{time}; \\ \textit{deadline} : \mathit{time}\}$$

The update is modelled by action *Update* which is executed periodically. It has a parameter of type *Physical_Location*, which is the physical location of the participant *m1*. There is also the implicit parameter *PL* containing the physical locations of all Objects. The action computes the speed relative to both axes based on the current place and the previous place. The speed is stored with current place in *l*. The guard of the action ensures that it is not executed too early and the deadline mechanism that it is not executed too late. The fairness requirement ensures that it will be executed. When executed, the deadline is removed and a new one is set to $\tau + \textit{period}$:

$$\begin{aligned} & \textit{Update}(*m1 : \textit{Mobile}; pl1 : \textit{Physical_Location}) : \\ & \quad \tau \geq m1.\textit{deadline} \\ & \quad \wedge pl1.\textit{ref} = m1 \\ & \quad \rightarrow m1.l.x_speed' = (pl1.x - m1.l.x)/m1.\textit{period} \\ & \quad \wedge m1.l.y_speed' = (pl1.y - m1.l.y)/m1.\textit{period} \\ & \quad \wedge m1.l.x' = pl1.x \\ & \quad \wedge m1.l.y' = pl1.y \\ & \quad \wedge \Delta_{off}(m1.\textit{deadline}) \\ & \quad \wedge m1.\textit{deadline}' = \Delta_{on}(m1.\textit{period}) \end{aligned}$$

This layer leaves actions *Connect* and *Disconnect* unchanged.

4.4 Coordination

Layer Coordination refines the composition of specification branches corresponding to layers Addresses and File Transfer, and Mobility (see Fig. 1).

A real-valued constant $Max_Distance$ defines a maximum distance between two Mobiles capable of communicating with each other:

constant $Max_Distance : real$

Action $Broadcast_Address$ is refined by strengthening its guard by two conjuncts. The first one requires that all participants in the quantified role $m2$ must be within $Max_Distance$ from $m1$. The second one requires that all Mobiles within $Max_Distance$ from $m1$ are participating in the role:

$Broadcast_Address(*m1, \{m2\} : Mobile) :$
refines $Broadcast_Address(m1, m2^{\{\}})$
 $\wedge \exists pl1, pl2 \in PL : (pl1.ref = m1 \wedge pl2.ref = m2$
 $\quad \wedge Distance(pl1, pl2) \leq Max_Distance)$
 $\wedge \neg \exists pl1, pl2 \in PL : (pl1.ref = m1 \wedge pl2.ref \in Mobile \wedge pl2.ref \neq m1$
 $\quad \wedge Distance(pl1, pl2) \leq Max_Distance$
 $\quad \wedge \neg pl2.ref \in m2^{\{\}})$

The guard of action $Start_File_Push$ is strengthened so that the distance between the two participants must be less than or equal to $Max_Distance$. When executed, it adds a hybrid deadline regarding the distance between the participants to $Max_Distance$:

$Start_File_Push(m1, m2 : Mobile; f : File; pl1, pl2 : Physical_Location) :$
refines $Start_File_Push(m1, m2, f)$
 $\wedge pl1.ref = m1$
 $\wedge pl2.ref = m2$
 $\wedge Distance(pl1, pl2) \leq Max_Distance$
 $\rightarrow \Delta_{on}^{Distance(m1, m2)}(Max_Distance)$

The guard of action $Stop_File_Push$ is strengthened similarly. It removes the hybrid deadline added in action $Start_File_Push$:

$Stop_File_Push(*m1, m2 : Mobile; pl1, pl2 : Physical_Location) :$
refines $Stop_File_Push(m1, m2)$
 $\wedge pl1.ref = m1$
 $\wedge pl2.ref = m2$
 $\wedge Distance(pl1, pl2) \leq Max_Distance$
 $\rightarrow \Delta_{off}^{Distance(m1, m2)}(Max_Distance)$

Action `Lose_Connection` is a refinement of action `Abort_File_Push`. It is assumed to be executed by the environment. It models the losing of the connection in the case when the distance between connected Mobiles grows too large. When this happens, the guard, the hybrid deadline and the fairness requirement ensure that the action is executed exactly when the distance equals `Max_Distance`. When executed, it removes the hybrid deadline:

$$\begin{aligned}
& \mathit{Lose_Connection}(*m1, m2 : \mathit{Mobile}; pl1, pl2 : \mathit{Physical_Location}) : \\
& \quad \mathbf{refines} \ \mathit{Abort_File_Push}(m1, m2) \\
& \quad \wedge \ pl1.ref = m1 \\
& \quad \wedge \ pl2.ref = m2 \\
& \quad \wedge \ \mathit{Distance}(pl1, pl2) \geq \mathit{Max_Distance} \\
& \quad \rightarrow \ \Delta_{\text{off}}^{\mathit{Distance}(m1, m2)}(\mathit{Max_Distance})
\end{aligned}$$

Fairness is required for a participant in all actions except `Start_Object_Push`, which is considered to be executed by the user. This layer leaves action `Update` unchanged.

4.5 More Predictable Behaviour

In the final superposition step, state variables and time are used to estimate the future values of distance between two Mobiles. The objective is to make the file transfer operation more predictable.

An integer-valued constant `Bandwidth` describes the data transfer rate between any two Mobiles:

$$\mathbf{constant} \ \mathit{Bandwidth} : \mathit{integer}$$

The type `File` is extended with attribute indicating the size of the file:

$$\mathbf{type} \ \mathit{File} = \mathit{File} + \{ \mathit{size} : \mathit{integer} \}$$

Function `Within_Range_Till_End` computes a Boolean return value estimating whether or not the two Mobiles are still within each other's range after some time t assuming the latest coordinates and speed:

$$\begin{aligned}
& \mathbf{function} \ \mathit{Within_Range_Till_End}(m1, m2 : \mathit{Mobile}; \\
& \quad pl1, pl2 : \mathit{Physical_Location}; t : \mathit{time}) : \mathit{boolean} \ \mathbf{is} \\
& \quad \sqrt{((pl1.x + m1.l.x_speed \times t) - (pl2.x + m2.l.x_speed \times t))^2 + \\
& \quad ((pl1.y + m1.l.y_speed \times t) - (pl2.y + m2.l.y_speed \times t))^2} \leq \mathit{Max_Distance}
\end{aligned}$$

The only action that is refined in this layer is `Start_File_Push`. To make the file transfer more predictable its guard is strengthened by a conjunct that ensures that either the transfer is likely to succeed or the user confirms the transfer:

$$\begin{aligned}
 & \textit{Start_File_Push}(m1, m2 : \textit{Mobile}; f : \textit{File}; pl1, pl2 : \textit{Physical_Location}; \\
 & \quad \textit{confirmation} : \textit{boolean}) : \\
 & \quad \mathbf{refines} \textit{Start_File_Push}(m1, m2, f, pl1, pl2) \\
 & \quad \wedge (\textit{Within_Range_Till_End}(m1, m2, pl1, pl2, f.size / \textit{Bandwidth}) \\
 & \quad \vee \textit{confirmation})
 \end{aligned}$$

From the point of view of the user, the effects of the refinement can be seen as follows. After the user has chosen a file and a recipient, an estimation is made on whether the file can be transferred successfully or not. In the former case the transfer begins immediately, but in the latter case a dialogue box is displayed to the user asking whether to begin transfer or not. If the user chooses to continue, the transfer is started, otherwise it is cancelled.

5 Conclusions

We have proposed the use of hybrid modelling in specification of novel wireless mobile systems. We believe that their development would benefit from modelling also aspects related to the continuous nature of the environment in which they operate. Furthermore, an approach to hybrid modelling of mobile systems was presented, where the locations of mobile agents change continuously. Moreover, it was demonstrated how discretely changing state variables and time can be used to estimate continuously changing distance between two agents. The approach can be generalized to other kinds of continuous quantities as well.

The way mobility was modelled in this paper was inspired by previous work done in related state-based formalisms, especially action systems [15] and Mobile UNITY [16]. Real time has been modelled previously in mobile specifications, e.g. in [14] where integer valued clock variables were used. However, the author is unaware of any previous work on modelling other continuous quantities in high level specifications of mobile systems.

To support the DisCo method a new toolset is under construction [2]. However, supporting all the issues raised in this paper remains as future work. Other interesting directions for future work include modelling variable bandwidth and other characteristics related to quality of service (QoS).

Acknowledgements

The author would like to thank all members of the DisCo project for their creative ideas and constructive comments. Tampere Graduate School in Information Science and Engineering (TISE) and Academy of Finland (project 57473) have funded the research presented in this paper. Moreover, the anonymous reviewers provided helpful comments.

References

1. The DisCo project WWW page. At URL <http://disco.cs.tut.fi>, 1999.
2. T. Aaltonen, M. Katara, and R. Pitkänen. Disco toolset – the new generation. FM-TOOLS 2000: The 4th Workshop on Tools for System Design and Verification, July 2000.
3. M. Abadi and L. Lamport. An old-fashioned recipe for real time. *ACM Transactions on Programming Languages and Systems*, 16(5):1543–1571, Sept. 1994.
4. R. J. R. Back and R. Kurki-Suonio. Distributed cooperation with action systems. *ACM Transactions on Programming Languages and Systems*, 10(4):513–554, Oct. 1988.
5. R. J. R. Back and R. Kurki-Suonio. Decentralization of process nets with centralized control. *Distributed Computing*, 3:73–87, 1989.
6. K. M. Chandy and J. Misra. *Parallel Program Design, A Foundation*. Addison-Wesley, 1988.
7. E. W. Dijkstra and C. S. Scholten. Termination detection for diffusing computations. *Information Processing Letters*, 11(1):1–4, Aug. 1980.
8. H.-M. Järvinen, R. Kurki-Suonio, M. Sakkinen, and K. Systä. Object-oriented specification of reactive systems. In *Proceedings of the 12th International Conference on Software Engineering*, pages 63–71. IEEE Computer Society Press, 1990.
9. R. Kurki-Suonio. Hybrid models with fairness and distributed clocks. In *Hybrid Systems*, number 736 in Lecture Notes in Computer Science, pages 103–120. Springer-Verlag, 1993.
10. R. Kurki-Suonio and M. Katara. Logical layers in specifications with distributed objects and real time. *Computer Systems Science & Engineering*, 14(4):217–226, July 1999.
11. R. Kurki-Suonio and T. Mikkonen. Liberating object-oriented modeling from programming-level abstractions. In J. Bosch and S. Mitchell, editors, *Object-Oriented Technology, ECOOP'97 Workshop Reader*, number 1357 in Lecture Notes in Computer Science, pages 195–199. Springer-Verlag, 1998.
12. R. Kurki-Suonio and T. Mikkonen. Harnessing the power of interaction. In H. Jaakkola, H. Kangassalo, and E. Kawaguchi, editors, *Information Modelling and Knowledge Bases X*, pages 1–11. IOS Press, 1999.
13. L. Lamport. The temporal logic of actions. *ACM Transactions on Programming Languages and Systems*, 16(3):872–923, May 1994.
14. P. J. McCann and G.-C. Roman. Modeling mobile IP in mobile UNITY. *ACM Transactions on Software Engineering and Methodology*, 8(2):115–146, Apr. 1999.
15. L. Petre and K. Sere. Coordination Among Mobile Objects. In P. Ciancarini and A. Wolf, editors, *Coordination Languages and Models, Third International Conference, COORDINATION'99*, number 1594 in Lecture Notes in Computer Science, pages 227–242, Amsterdam, The Netherlands, Apr. 1999. Springer-Verlag.
16. G.-C. Roman, P. J. McCann, and J. Y. Plun. Mobile UNITY: Reasoning and specification in mobile computing. *ACM Transactions on Software Engineering and Methodology*, 6(3):250–282, July 1997.