Fact Spaces: Coordination in the Face of Disconnection

Stijn Mostinckx*, Christophe Scholliers, Eline Philips,
Charlotte Herzeel* and Wolfgang De Meuter
{smostinc | cfscholl | ephilips | caherzee | wdmeuter}@vub.ac.be

Programming Technology Laboratory
Vrije Universiteit Brussel, Belgium

Abstract. Coordination languages for ad hoc networks with a fluid topology do not offer adequate support to detect and deal with device disconnection. Such a disconnection is particularly relevant if the device provided context information rather than emitting messages, as such context information then becomes invalid. This paper proposes the Fact Space Model which establishes a logic coordination language on top of LIME's federated tuple space. In the model, the federated space offers applications a consistent view of their environment over which they can reason using logic rules. These rules encode which conclusions may be drawn from the presence of particular facts, and are similarly used to ensure the consistency of these conclusions when devices go out of range. By allowing applications to add application-specific hooks to these rules, the application programmer is offered a general-purpose mechanism to respond to the discovery and disconnection of devices.

1 Introduction

The increasing popularity of cellular phones, PDAs and numerous other mobile devices heralds the realisation of the ubiquitous computing and ambient intelligence research visions [1, 2]. A crucial aspect of the various scenarios put forward as part of these visions is that mobile users can reap the benefits of being surrounded by a cloud of small, interconnected computing devices. These benefits come in two distinct flavours: providing either additional or smarter behaviour. An example of the former is offering printing facilities to users in the proximity of a printer. A classical example of providing smarter behaviour is signalling a user's cellular phone that it is in a meeting room, allowing it to forward incoming calls to the user's voice mail.

As managing the interplay of services in an ad hoc network with a fluid topology and transient connectivity can become quite complex, this task is typically entrusted to a dedicated coordination language. A well-established coordination language for mobile ad hoc networks is provided by the LIME [3] middleware.

* Funded by a doctoral scholarship of the Institute for the Promotion of Innovation through Science and Technology in Flanders (IWT-Vlaanderen), Belgium.

LIME allows publishing information that steers the coordination of a set of applications in a tuple space that is *transiently shared* between all devices that are currently in range. Devices can respond to the appearance of such tuples by registering *reactions*. Unfortunately, LIME keeps no record of the causal link between a tuple and the reactions the tuple provoked. For the examples described above, this implies that there is no automatic mechanism to remove the printing facilities nor to signal that the cellular phone should stop forwarding calls when the user is no longer in the proximity of the printer respectively when he is no longer in the meeting room.

This paper proposes the Fact Space Model which conceives the federated space as a distributed knowledge base. The transiently shared facts describe the current environment of a device and may be used to customize the behaviour of applications accordingly. These customizations are achieved using a logic coordination language whose rules record the causal link between (a set of) facts and the conclusions that may be drawn from them. As facts are retracted when the device that published them goes out of earshot, the strict enforcing of these causal links – invalidating conclusions when the supporting facts are retracted – is the basis for the fine-grained support to deal with the effects of disconnection offered by the Fact Space Model.

The remainder of the paper is organised as follows: the next section introduces the relevant elements of the Fact Space Model in more detail. Subsequently, section 3 describes CRIME (Consistent Reasoning in a Mobile Environment), a prototypical implementation of the Fact Space Model in which we have conducted our experiments. A selection of these experiments is then presented in section 4. Finally, we provide an overview of related work and present our conclusions.

2 The Fact Space Model

The Fact Space Model is a coordination model offering applications deployed on a mobile ad hoc network a consistent view on their environment. According to this view of their environment, the application may offer additional or adapted functionality to its user. An application's view of its environment consists of facts published in a *federated fact space*. Concretely every application can locally publish facts and transparently shares the facts of all nearby devices as long as they remain within communication range. Applications may react to the appearance of facts, using rules specified in a *logic coordination language*. These rules map (a combination of) facts onto a conclusion, which may involve adding new facts to the fact space or triggering application-specific actions. The Fact Space Model allows applications to intercept the retraction of these actions, at which point a compensating action will be performed. Given that facts are retracted automatically when devices disconnect, this mechanism provides fine-grained control over the effects of disconnection. The remainder of this section will explain both the federated fact space and the logic coordination language in more detail.

2.1 Federated Fact Spaces

Applications in the Fact Space Model contribute to the shared view of the environment by publishing facts. Such facts can uniformly represent various types of information ranging from context information over service descriptions up to tasks to be performed. Ensuring these facts are transparently distributed is done using a federated space, as was originally proposed in LIME [3]. With regards to the distribution architecture, the only difference with LIME is that the federated space represents a knowledge base containing facts. Consequently, both the assertion and the retraction of facts are meaningful events which may have repercussions on how the applications behave.

Applications are equipped with at least two fact spaces, a private one to store application-specific facts and one or more *interface fact spaces* for facts that need to be disseminated. In the cell-phone example, the private fact space could contain user preferences detailing which behaviour to adopt when in a particular (kind of) room. The location of a device is typically derived by the proximity to a predefined device, which published location information in an interface fact space, as illustrated by figure 1. The federated fact space consists of all host level fact spaces for devices which are currently in range. The aggregation of all interface fact spaces on a single host allows reusing context information that was previously computed by another application (e.g. a printer is available at a given ip-address) as well as coordination within the boundaries of a single device.

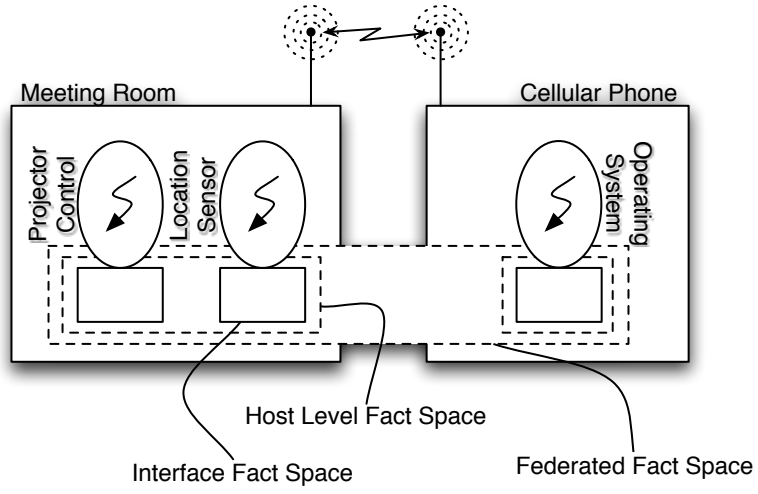

Fig. 1. A Federated Fact Space

The Fact Space Model handles the discovery of devices in much the same way as LIME; whenever a host discovers the presence of a new device, it will *engage* the fact space of that device. This implies that all facts in the fact space of that device are atomically and transparently *asserted* in the host's fact space. At this point each application may adapt to its new environment using the logic coordination language described in the next section. When a device goes out of earshot, its fact space is *disengaged* implying that the facts published by that device are *retracted* from the host's fact space. The Fact Space Model offers applications provisions to respond to such disconnection, as will be explained in the next section.

2.2 Logic Coordination Language

The federated fact space described in the previous section provides each application with a consistent view of their environment. Applications can react to changes in their environment using rules in the logic coordination language of the Fact Space Model. A rule could specify for instance that upon detecting a printer in the environment, it should be added to the list of available printers. Figure 2 illustrates how such a rule could be written in the Fact Space Model. The rule can be read as follows: the application-specific action `addToPrinterList` should be performed (with a given set of arguments) *if and only if* the `public` federated fact space contains a `printer` fact (with the same arguments) whose `dpi` is at least 300.

```
:addToPrinterList(?name, ?ip) :-
  public -> printer(?name, ?dpi, ?ip),
  ?dpi >= 300.
```

Fig. 2. Rule to add printer functionality to an application.

For consistency, the rule in figure 2 uses the syntax of CRIME, our implementation of the Fact Space Model. As can be seen from the example, logic variables in the rules are prefixed with a question mark. Furthermore, a colon prefix is used to differentiate between application specific actions and logic facts. The former denote implicit method invocations on the `Action` subclasses described below. Finally, facts can be quantified to denote the fact space in which they should be found or asserted.

The main difference between the rule specified above and a similar *reaction* in LIME, is that the former implicitly provides a hook to respond to the disappearance of the fact. This is achieved by enforcing that any custom actions inherit from a common abstract class `Action`. Subclassing from this class implies that application developers need to implement an `activate` method which describes how to respond when the action is derived, as well as a `deactivate` method which describes a compensating action to be performed when the action is retracted as a response to changes in the environment. Note that compensating actions are not required to restore the application's state prior to the execution of the `activate` method.

```
room(meetingRoom, silent).
room(office, general).

:switch(?profile), profile(?profile) :-
  public -> location(myID, ?room),
  room(?room, ?profile).

:switch(default) :- not profile(?p).
```

Fig. 3. Facts and rules to change the profile of a cellular phone.

Figure 3 exemplifies how the Fact Space Model can be used to switch the profile of a cellular phone depending on its location. The example shows a set of private `room` facts detailing the user's preferred profile for a particular room. Furthermore, it contains a rule which triggers the `switch` application-specific action as well as adding a private `profile` fact, upon detecting that the cellular phone is located in a particular room for which the user has configured a preferred profile. Finally, a rule is provided to switch to the `default` profile when no explicit `profile` is prescribed[1].

3 Crime : Implementing the Fact Space Model

Crime (Consistent Reasoning in a Mobile Environment) is an experimental implementation in which we have explored the advantages of the Fact Space Model in building context-aware software. This section highlights some of the key points of the implementation before moving on to the examples presented in the next section.

3.1 Federated Fact Spaces

Crime's implementation of the federated fact space is achieved by building on top of Lime, which is possible due to the fact that the underlying distribution architecture of both systems are identical. This means that the interface fact spaces of an application are in fact wrappers around interface tuple spaces in Lime. Whenever a fact is added to an interface tuple space – either directly by the application or through derivation – this corresponds to adding the fact to the interface tuple space using the `out` operation. Similarly, the atomic engagement and disengagement of fact spaces is transparently handled by the underlying Lime implementation.

Achieving correct behaviour requires that the reasoning engine be informed of the appearance and disappearance of facts which are relevant to its rule base. This is achieved by installing *once-per-tuple* reactions whenever compiling a rule

[1] The observant reader may notice that this rule is not strictly necessary as similar behaviour could be achieved using the compensating action of `switch`.

that depends on public facts. These reactions will trigger the reasoning engine and inform it that new facts have become available. The disappearance of hosts is observed in the system tuple space of LIME, which posts a `_host_gone` tuple upon such an event. The reasoning engine is notified of this event (by means of a reaction) and will subsequently handle the disconnection as described in the next section.

3.2 Logic Coordination Language

In correspondence with the event-driven nature of the systems CRIME caters for, it uses a forward-chaining inference engine to trigger rules when fact are asserted to and retracted from the federated fact spaces. The major benefit of forward-chaining is that rather than answering queries, all valid conclusions from a given set of facts are automatically derived. This implies that applications never have to query their context explicitly, instead they are automatically notified of all relevant changes in their environment.

CRIME optimizes the derivation of valid conclusions for all available fact using the RETE algorithm [4], which we briefly explain below. The gist of the RETE algorithm is to combine the actual derivation of conclusions with an optimized caching of intermediate results in a so-called RETE network. This caching strategy minimizes the set of rules to be re-evaluated whenever new facts are asserted. Dealing with the frequent retraction of facts – caused by devices which routinely go out of earshot – requires additional support. Whereas the RETE algorithm can inherently deal with the retraction of facts, this requires the negated fact to be propagated through the network. This process can be optimized by explicitly keeping track of the dependencies between conclusions and facts using a justification-based truth maintenance system. The remainder of this section will briefly describe both components which jointly implement the logic coordination language prescribed by the Fact Space Model.

Inference Engine The RETE algorithm allows for the efficient derivation of valid conclusions given a set of facts by compiling these rules into a so-called RETE network [4] with built-in caches for intermediate results. We exemplify the compilation of such a rule using the example given in figure 3, which dealt with changing the profile of a cellular phone according to the room it was in. The corresponding network for this rule is shown in figure 4.

Deciding when to change the profile of the cellular phone requires the presence of multiple facts. First of all, context information from the `public` interface fact space is needed to determine in which room the phone currently resides. Secondly, a private fact is needed which states the user's preferred profile for that room. Compiling the right-hand side of the rule, involves two steps :

1. Every fact is represented in the network by a *filter node* which will be triggered whenever a new fact is asserted with the correct name. When the fact is qualified, the filter node ensures it will be notified by registering a *reaction* for facts of that type in the underlying LIME distribution layer. As part of

filtering out relevant events, the filter nodes also check simple constraints with respect to constants. In the example, the left-most filter node is used to filter out any `location` facts which are not related to the cellular phone.
2. Facts are subsequently combined pairwise using *join nodes* which ensure that constraints spanning multiple facts are upheld. A typical example of such a constraint is that variables with the same name have the same value. Similarly, if a rule contains explicit constraints on two variables (using relational operators such as `<=`, `>=` or `=\=`), these are checked by the first join node which has access to both variables.

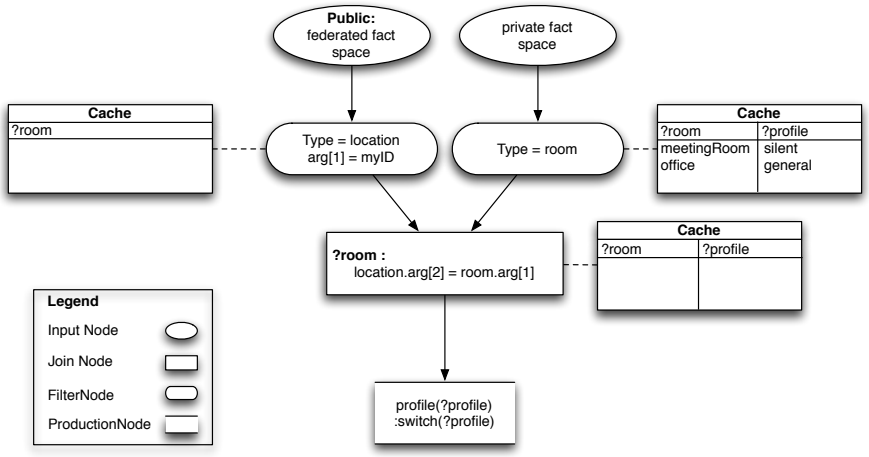

Fig. 4. RETE network to switch the profile of a cellular phone based on location.

Having built a network which filters out the relevant combinations of facts that may trigger a rule, compiling its left-hand side is simply adding a *production node* as the child of the lowest join node for that rule. Upon triggering a production node, facts are added to their designated fact spaces (recall that if no qualification is given the fact is considered private) and application-specific actions need to be performed (invoking the `do` method). It is critical that a production node encapsulates an atomic activation, implying that the inference engine may only re-evaluate rules after the production node completed its task. If this constraint is not upheld, inconsistent behaviour may be triggered by mutually exclusive rules with negated clauses in the right-hand side (e.g. `A and !B`, and `A and B`).

Truth Maintenance System The Fact Space Model introduces a logic coordination language which offers applications the ability to reason over a consistent view of their environment. Given that this environment is a mobile ad hoc network with a fluid topology and transient connectivity, this implies that suites

of facts will be frequently retracted as devices move out of range. Such retraction cannot be delayed as this would compromise the consistency of the view an application has of its environment. Moreover, as the connectivity between the devices in a mobile ad hoc network is transient, it is quite likely that the same suite of facts will be reasserted later on as the communication is restored. This particular set of constraints necessitates the introduction of additional support to handle the retraction of facts in an acceptable way.

The RETE algorithm can handle the retraction of facts using rematch-based removal which involves propagating the dual of the removed fact through the network. However, this is a rather costly operation as all checks in the filter and join nodes need to be recomputed, and more importantly, it also removes useful information which could be used when the same fact is reasserted later on.

To optimize the retraction of facts, we use the scaffolding technique described by Perlin [5]. Concretely, this implies that a truth maintenance system is used which keeps track of the causal links between facts and conclusions. These causal links are said to be the *justification* for believing the conclusion. When a fact is subsequently retracted, it is straightforward to identify precisely those conclusions that need to be undone. Moreover, rather than deleting the information upon retracting, the causal link is preserved yet marked to be currently deactivated. If the same fact is reasserted shortly afterwards, this information can be used to reactivate the tuple.

4 Building Context-Aware Applications

We have employed CRIME to build a suite of context-aware applications all of which rely on location-based information. Throughout this paper we have already illustrated a simple application which allows switching the profile of a cellular phone. We have built similar light-weight applications such as an in/out board and a messenger service [6] which are presented in more detail at our website[2]. In this paper we present a slightly more involved application, namely a context-aware jukebox [7, 8]. The precise scenario is described first, followed by a discussion of the location-based support for the application before we describe the rules which comprise the juke-box application itself.

4.1 A Context-Aware Jukebox

Alice, Bob and Carol are students which share an apartment. When they are not attending classes or studying, a great deal of their life is all about music. As a consequence, when one of them is relaxing in the joint living room of their apartment, it is quite common to find their jukebox playing music. Unfortunately, the students do not always share one another's taste in music. Whereas this might be a recipe for endless quarrels in any other situation, there is no arguing over who is in charge of choosing the music being played. This is due

[2] `http://prog.vub.ac.be/amop/research/crime`

to the fact that the jukebox is in fact a small computer (a Mac Mini in our setup) which combines information regarding the presence of its users with their respective musical preference to construct a playlist which is acceptable for all present users. Moreover, If Alice, Bob and Carol invite some friends, their musical preferences can be taken into account as well. Finally, the jukebox can also stop playing automatically when it detects that no users are present.

A variety of issues are important to build a similar system. First of all, a fairly reliable mechanism is needed to detect when users are in a particular room. Second, when a user enters the room, his musical preferences should be queried and possibly the jukebox should be turned on. Thirdly, when a guest is invited, he may not have the appropriate software to cast his vote in the music selection and this software should be offered to him. Finally, the music selection should be designed to take into account the musical preferences of the present users. We will present the necessary building blocks for this setup in the remainder of this section.

4.2 Detecting a User's Location

Over the past few years, a large quantity of systems to derive a person's location have been developed. Positioning users indoor is usually achieved by giving them a small device which is tracked using wireless communication. The technology used may vary from infrared signals (as used in the seminal work on the Active Badge system [9]) up to the recently developed rfid tags [10]. Whereas the choice of which technology to use may be critical as it influences the size and battery consumption of the device that needs to be carried around, the effect on the proposed application is rather minimal. While developing the context-aware jukebox we have therefore adopted a conservative stance and employed the bluetooth connectivity offered by the jukebox to detect users (by means of their bluetooth-equipped cellphones).

Our concrete setup consists of an event-driven application which is used to detect all reachable bluetooth devices. Upon detecting a new device, it triggers an embedded Crime process by asserting a private fact (e.g. `observed ("Alice's Phone")`). The application further consists of a trivial mapping of such facts onto public `location` facts, as we have used in previous examples in the paper.

4.3 Deploying Applications

We have identified the deployment of applications, in this case to specify one's musical preference, as an essential aspect of the scenario at hand. Obviously, the deployment of Crime applications requires the presence of a minimum of infrastructure. Therefore, every Crime engine is equipped with a minimal application which listens for facts describing available applications. These applications are presented to the user who can opt to deploy them. Figure 5 illustrates how a simple rule can be used to notify the user interface (using a custom action `offer`). The actual deployment (i.e. downloading the application from the url and running it) is done by the underlying application.

```
:offer(?name ?url) :-
  application(?name ?url).

application(jukebox, "http://prog.vub.ac.be/amop/crime/jukebox.zip").
```

Fig. 5. Supporting the advertisement of available applications.

One notable property of this application is that the list of available applications can be dynamically updated as the topology of one's mobile network changes. As the service descriptions are represented as public facts, these facts are automatically retracted from the fact space. In response, the `undo` method of the `offer` action will be invoked. In our current implementation, this method will simply remove the application from the list of available applications. It is also possible to treat applications which were deployed separately using a small set of additional rules. The gist of the system would be to let the deployment application assert a private fact whenever it deploys an application, such that the CRIME engine can include this information in its reasoning.

4.4 Music Selection

One of the major advantages of using a coordination language is that it offers a separate medium to specify the coordination and distribution aspects of an application. Concretely this implies that a traditional, non-distributed application can be adapted to provide additional or smarter behaviour when employed in a mobile context, as long as it provides the adequate hooks. To illustrate the potential of the Fact Space Model to be employed in this context, we have opted to use an existing and highly scriptable music player in our jukebox example. The iTunes music player[3] is an established software artifact which provides a rich set of hooks through its AppleScript support.

The CRIME support for determining which music to play is described by the rules in figure 6. The first rule uses an application-specific action `toggle` to switch the jukebox on as soon as it detects a person in the living room. This detection is based on the `location` facts which were published by the location sensing support described in section 4.2. The second rule will use `updateRating` to change the average preference attributed to a genre based on the amount of people preferring a particular genre and the total amount of people in the room. This is sufficient to ensure that the music played by the jukebox will be appreciated by the users presented in the living room, as iTunes offers a dedicated *party shuffle* playlist which plays a selection of highly-rated music.

The rating of a particular genre depends on the number of present users which prefer a particular genre, as well the total amount of users in the living room. To compute these numbers, CRIME integrates support for two accumulation techniques borrowed from Prolog, namely `findall` and `bagof`. The former is

[3] Copyright 2000-2006 Apple Computer Inc.

```
:toggle() :-
  location(?person, "Living Room").

:updateRating(?genre, ?rating) :-
  category(?genre, ?absolute),
  total(?total),
  rating is ?absolute / ?total.

total(?quantity) :-
  findall(?person, (
    location(?person, "Living Room")),
    ?persons),
  length(?persons, ?quantity).

category(?genre, ?quantity) :-
  bagof(?person, (
    location(?person, "Living Room"),
    prefers(?person, ?genre)),
    ?persons),
  length(?persons, ?quantity).
```

Fig. 6. Rule set to customize jukebox playlist

used to find all values for the `?person` variable who are located in the living room and accumulate them in the `?persons` variable. To find out how many people like a particular genre, the `bagof` construct is used, which also accumulates all values for the `?person` variable, yet groups them according to the corresponding values for other free variables (the `?genre` variable in the example).

5 Related work

The main contribution of the Fact Space Model is that it introduces the notion of causality in a coordination language as a fundamental mechanism to offer applications fine-grained control over the effects of device disconnection in mobile ad hoc networks. The introduction of causality is achieved by using a logic language which essentially reasons over a distributed fact base in order to coordinate the different applications involved. Finally, we have illustrated the applicability of the Fact Space Model to develop context-aware application. Therefore, the Fact Space Model can be contrasted to previous work in three fields of research, namely *coordination languages*, *distributed reasoning systems* and *context-aware computing middleware*. Our discussion of related work will therefore highlight the contributions in each of these research fields which are the most closely related to the Fact Space Model.

5.1 Coordination Languages

Coordination languages can be categorized according to the primitive communication support they offer: communication can be based on *directed* channels which connect two processes (the endpoints of such channels may vary over time), or it can be achieved using the notion of a shared blackboard which provides processes with an *undirected* communication model. The Fact Space Model employs the latter model of communication, making it akin to tuple space-based coordination languages.

During the discussion of the Fact Space Model in section 2, we have already indicated that it borrows the notion of a federated space from LIME [3]. Moreover, through the introduction of an inference engine, our model allows responding to changes in the state of the federated space (rather than operations performed on them) allowing behaviour similar to LIME's reactions. At first sight, reacting to disconnection can be introduced in LIME by allowing such reactions to include a compensating action which is then triggered upon disconnection. To the best of our knowledge, such support has not been introduced thus far. Even with such support in place, some differences remain with respect the Fact Space Model. First of all, the model has innate support for dealing with disconnection, requiring no additional infrastructure to detect and propagate disconnection events. More importantly, by keeping track of the causal connection between events, hand-coding compensating actions is only necessary when the reaction is related to the application's behaviour rather than with how it interacts with other applications.

Tuple space-based communication can equally well be achieved without the notion of a transiently shared tuple space, as is exemplified by TOTA [11]. Instead of implicitly sharing tuples when the hosting devices come into one another's range, TOTA tuples are instrumented with behaviour dictating when they may be copied (or moved) from one tuple space to the next. As tuples are copied, these tuples are not automatically retracted when the emitting device goes out of range. Using TOTA's support to notify tuples of system events, tuples can be removed after a certain period is elapsed. Broadcasting such tuples periodically would then allow to detect disconnection. Another possibility would be to reify the transitive unreachability of a device as an event of its own. However, even if the tuple can be removed when necessary, TOTA suffers from the same problems as LIME, namely that for each reaction that needs to be undone the corresponding compensating action needs to be hand-coded.

5.2 Distributed Reasoning Systems

The Fact Space Model goes beyond traditional coordination languages in that it introduces the notion of an inference engine coupled to a truth maintenance system. It therefore embodies a distributed reasoning engine, with explicit support for dealing with a fluid topology of fact providers. Whereas abundant distributed reasoning engines exist, most of them were introduced purely for the sake of

parallelism [12, 13], rather than to support reasoning in and about a physically distributed context.

UbiES is an expert system for modelling context-aware applications in a nomadic network setting [14]. The lack of support for ad hoc networks shows in various parts of the proposal: First of all, context information is managed in a context database which is apparently centralized. Moreover, clients are equipped only with a web browser leaving the actual functionality on a server. Finally, UbiES focusses on a single application at a time rather than at coordinating the interplay of various applications.

The Fact Space Model is closely related to DJess [15], a distributed variant of the Jess forward chainer [16]. Similar to the Fact Space Model, DJess connects different inference engines allowing them to share contextual information and trigger one another's reactive behaviours. However, the fundamental difference between both systems is that DJess does not consider applications for a mobile ad hoc network with a fluid topology. A first indication is that DJess features a single centralized server which serves as an intermediate to communicate facts between the individual clients. Furthermore, concurrency control is pessimistic as all facts needs to be locked before executing the actions associated with a rule. Finally, DJess provides no support to detect nor respond to disconnection.

Cooperative artefacts [23] sport a reasoning engine which is specifically tailored for mobile ad hoc networks. Similar to the Fact Space Model, it offers devices the means to reason about a distributed knowledge base representing their immediate environment using Prolog-like rules. The chief difference between both systems lies in the type of applications they support. Cooperative artefacts are embedded devices with very limited resources designed to address one specific problem, whereas the Fact Space Model addresses the collaboration of different idiosyncratic applications on high-end mobile devices. One aspect where the difference in both the underlying hardware and the supported applications has a considerable influence is the inference strategy. Cooperative artefacts rely on a backward chaining inference engine whereas the Fact Space Model emphasizes the importance of a forward chaining inference engine. Section 6.2 explains in more detail why this difference is relevant.

5.3 Context-aware Computing Middleware

As illustrated in section 4, the Fact Space Model lends itself quite well to the development of context-aware applications. Its forward-chained rules could be interpreted as a structured and declarative subscription to relevant context information publishers. However, comparing it to classical middleware such as the Context Toolkit [6], JCAF [17] and WildCAT [18], a few differences are immediately apparent. First of all, the distribution model underlying the Fact Space Model is based on a federated space rather than on event channels. This is essential as it permits detecting when facts are retracted, and to act upon this observation. Secondly, the Fact Space Model relies on a separate logic language to describe the influence of context, a trait it shares with the approaches discussed in the remainder of this section.

Chisel is a system which primarily focusses on describing the concrete effects of contextual changes by assigning meta-types to runtime objects [19]. As a coordination language, the Fact Space Model was not designed to support such advanced adaptation strategies. Nevertheless, Chisel and the Fact Space Model are quite complementary as the latter offers a declarative language with support to respond to a combination of events as well as a clear distribution model for context information. Both features are lacking in Chisel's event-condition-action language to specify when adaptations should be plugged in.

Gaia's active spaces [20] have been extended with a dedicated language based on first order logic to describe how a system should adapt according to context information [7]. The context model for Gaia's Active Spaces uses first order logic to describe how to adapt to context information. Whereas the use of first order logic results in a similar expressive power to our proposal, quite a few differences remain between Gaia and the Fact Space Model: First of all, Gaia relies on context providers which publish information on channels managed by a centralised infrastructure. Another notable difference is that Gaia uses a standard Prolog implementation, rather than a forward chainer, which necessitates a manual triggering of rules upon context changes. Most importantly, Gaia assumes reliable connections and does not specify how to respond when no context information can be read, nor how this affects previously made decisions.

The Fact Space Model bears the most similarity to the Cortex middleware [21] which uses clips [22], a production system to reason about context information and to trigger reactions using clips's foreign function interface. The chief differences between Cortex and the Fact Space Model is that the latter is based on the notion of a federated space to exchange information, rather than on the publish/subscribe paradigm and most importantly, that the Fact Space Model provides support to meaningfully deal with the retraction of information from the said federated space.

6 Discussion

The Fact Space Model is a logic coordination language which allows reasoning about a distributed knowledge base which functions as a view on the (physical) environment. Moreover, the Fact Space Model requires this view to be kept consistent to ensure that applications do not act upon stale facts. This requirement has a direct influence on two important aspects of the Fact Space model which will be the topic of discussion in this section.

Automatic Retraction We have adopted the stance that facts whose provider has gone out of earshot are to be retracted from the knowledge base. This paradigmatic decision is evaluated with respect to the opportunities it leaves the programmer to encode persistent facts, *i.e.* facts which are not automatically retracted when its provider goes out of earshot.

Inference Strategy The choice for a forward chaining inference engine also follows from the requirement that applications should not act upon stale facts. Its data-driven reasoning strategy aligns well with the fact that changes to the knowledge base should be promptly reflected in the application's behaviour. This section provides a more thorough analysis of the differences with a goal-driven backward chaining strategy such as the one used in cooperative artefacts [23].

6.1 Persistent Facts

A distinguishing characteristic of the Fact Space Model is that it reifies device disconnection by retracting all facts published by that device. This is crucial to ensure that applications have a consistent view of their environment where all facts in the knowledge base are guaranteed to be true. Without the automatic retraction of facts, applications could be presented with stale data which leads to a variety of problems. Consider the following example: a location sensor detects that Alice is in her office and publishes this information. Bob's PDA receives this information and subsequently goes out of range. If the location information would not be retracted, the PDA would need to assume that Alice stays put, as it is unable to receive any new information. Such assumptions are clearly a source for unadjusted behaviour such as calling Alice's office number rather than her cellular phone since we presume to know her location. In this case, having no information available is clearly better than having incorrect information.

The semantics of the Fact Space Model align with those of the underlying distribution model of federated spaces, and may seem quite natural in the location-based example given above. However, the retraction of facts is not always the desired semantics. For instance, when a specific printer is out of reach, the fact representing its *availability* should be retracted, yet it can be useful to keep a fact representing information about the printer such as its maximal resolution. At present, CRIME offers no direct support to achieve such behaviour, though this behaviour can be conceived in the following way:

Persistent Facts One can conceive a `:persistent` custom action which adds a fact in its `activate` method, yet does not remove the fact in its `deactivate` method. As exemplified in figure 6.1, this mechanism can be used in the example to add a fact which represents the information about a printer, whenever a (public) fact representing the *availability* of a printer is first detected in the environment. Using a similar custom action, such persistent facts can then also be removed from the system, for instance at the user's request.

```
:persistent(knownPrinter(?name, ?dpi)) :-
  public -> printerAvailable(?name, ?dpi, ?ip),
  not knownPrinter(?name, ?dpi).
```

6.2 Inference Strategy

At the heart of the Fact Space Model lies a logic-based coordination language reasoning over a distributed knowledge base. The inference engine which interprets the rules in the coordination language can employ two distinct inference strategies: namely backward and forward chaining. Backward chaining is a *goal-driven* strategy which attempts to prove queries supplied by the users, whereas forward chaining is a *data-driven* strategy which derives all valid conclusions from a given data set. This section briefly discusses the merits of both strategies and motivates CRIME's use of a forward chaining inference engine.

Backward chaining is a commendable strategy to reason over a stable distributed knowledge base which is dedicated to a single application. The underlying reason for this restriction is that in order to respond to the availability of new data, the inference engine needs to be triggered explicitly. For instance, Strohbach *et al.* issue a new attempt to prove a predefined set of goals whenever a change in the distributed knowledge base is observed [23]. Each such attempt completely reconstructs the proof which can be both costly and time-consuming (*e.g.* it may require remote communication). It is therefore important to ensure that such attempts are only made when the result produced by the inference engine is likely to have changed. Hence, the use of backward chaining inference engines for the coordination of different device should be restricted to cases where all facts in the knowledge base are related to a single application. In these cases, the addition of facts in the knowledge base (which in turn triggers the inference engine) is the most likely to affect the outcome of the reasoning process.

Forward chaining on the other hand is a useful strategy to reason over a fluctuating distributed knowledge base which is shared by different idiosyncratic applications. Forward chaining is a data-driven reasoning strategy which implies that the inference engine is triggered whenever new data becomes available. The chief difference with the strategy outline above is that when new data becomes available, the inference engine builds its proofs bottom-up. This implies that only those parts which are affected by the appearance of the new data need to be derived anew. This strategy is particularly beneficiary when changes to the knowledge base (which may originate from different applications) may not be relevant, as such changes can be filtered out in the first step of reasoning.

7 Conclusion

Starting from a traditional coordination language based on a LIME-like federated tuple space, this paper has explored how applications can be presented with a consistent view of their environment, allowing them to adapt their behaviour according to their current context. Crucial in such applications is that they are notified of *all* relevant changes in the environment, including when information becomes unavailable as a consequence of user mobility or transient disconnection. This has led us to propose the Fact Space Model, which differs from a classical tuple space in two regards: First of all, the federated space is treated as

a knowledge base where both the *assertion* and the *retraction* of facts are relevant. Secondly, to provide a minimal and reasonable behaviour when retracting facts, the Fact Space Model is a fully reactive system where the causal relations between reactions and the triggering of facts is documented using logic rules.

The Fact Space Model combines a particular set of features, which clearly sets it apart from existing work on coordination languages, distributed reasoning engines and context-aware computing. First of all, using a federated space to manage context information rather than a publish-subscribe mechanism allows detecting the retraction of information. This feature allows the Fact Space Model to respond to the disconnection of a context provider, a meaningful event when dealing with mobile ad hoc networks with a fluid topology. Furthermore, the Fact Space Model is a fully reactive coordination language based on a forward chained logic language. This language offers a similar programming model as reactive tuple spaces, with the added benefit of delimiting the causal relation between facts and the resulting actions. This causal link allows undoing the effects of a context-dependent adaptation when the context is no longer valid. This feature is currently not offered by coordination languages for mobile ad hoc networks. Finally, the Fact Space Model differs from existing distributed reasoning systems as it uses a distribution model which does not rely on reliable communication or a centralized architecture.

Acknowledgements The authors would like to thank Brecht Desmet, Kris Gybels, Tom Van Cutsem and the anonymous reviewers for their comments on earlier drafts of this paper. They have significantly improved the readability and quality of this paper.

References

1. Weiser, M.: The computer for the twenty-first century. Scientific American (1991) 94–100
2. Ducatel, K., Bogdanowicz, M., Scapolo, F., Leijten, J., Burgelman, J.C.: Scenarios for ambient intelligence in 2010. Technical report, EC Information Society Technologies Advisory Group (ISTAG) (2001)
3. Murphy, A., Picco, G., Roman, G.C.: Lime: A middleware for physical and logical mobility. In: Proceedings of the The 21st International Conference on Distributed Computing Systems, IEEE Computer Society (2001) 524–536
4. Forgy, C.L.: Rete: a fast algorithm for the many pattern/many object pattern match problem. In: Expert systems: a software methodology for modern applications. IEEE Computer Society Press (1990) 324–341
5. Perlin, M.: Scaffolding the RETE network. In: Proceedings of the International Conference on Tools for Artificial Intelligence, IEEE Computer Society (1990) 378–385
6. Dey, A.K., Salber, D., Abowd, G.D.: A conceptual framework and a toolkit for supporting the rapid prototyping of context-aware applications. Human-Computer Interaction **16** (2001)
7. Ranganathan, A., Campbell, R.H.: An infrastructure for context-awareness based on first order logic. Personal Ubiquitous Comput. **7** (2003) 353–364

8. Barron, P., Cahill, V.: Using stigmergy to co-ordinate pervasive computing environments. In: WMCSA '04: Proceedings of the Sixth IEEE Workshop on Mobile Computing Systems and Applications (WMCSA'04), Washington, DC, USA, IEEE Computer Society (2004) 62–71
9. Want, R., Hopper, A., Falcao, V., J. Gibbons, J.: The active badge location system. ACM Transactions on Information Systems **10** (1992) 91–102
10. Ni, L.M., Liu, Y., Lau, Y.C., Patil, A.P.: Landmarc: indoor location sensing using active rfid. Wirel. Netw. **10** (2004) 701–710
11. Mamei, M., Zambonelli, F.: Self-maintained distributed tuples for field-based coordination in dynamic networks. In: SAC '04: Proceedings of the 2004 ACM symposium on Applied computing, New York, NY, USA, ACM Press (2004) 479–486
12. Eliëns, A.: DLP: a language for distributed logic programming: design, semantics, and implementation. John Wiley & Sons, Inc., New York, NY, USA (1992)
13. Cunha, J., Medeiras, P., Carvalhosa, M., Pereira, L.: Deltaprolog: A distributed logic programming language and its implementation on distributed memory processors. In Kacsuk, P., Wise, M., eds.: Implementations of Distributed Prolog. John Wiley & Sons, Inc. (1992) 335–356
14. Kwon, O., Yoo, K., Suh, E.: ubiES: An intelligent expert system for proactive services deploying ubiquitous computing technologies. In: HICSS '05: Proceedings of the Proceedings of the 38th Annual Hawaii International Conference on System Sciences (HICSS'05) - Track 3, Washington, DC, USA, IEEE Computer Society (2005) 85.2
15. Cabitza, F., Sarini, M., Seno, B.D.: Djess - a context-sharing middleware to deploy distributed inference systems in pervasive computing domains. In: International Conference on Pervasive Services, 2005. ICPS '05., IEEE Computing Society (2005) 229–238
16. Friedman-Hill, E.: Jess in Action: Java Rule-Based Systems. Manning Publications Co. (2003)
17. Bardram, J.E., Hansen, T.R.: The aware architecture: supporting context-mediated social awareness in mobile cooperation. In: CSCW '04: Proceedings of the 2004 ACM conference on Computer supported cooperative work, New York, NY, USA, ACM Press (2004) 192–201
18. David, P.C., Ledoux, T.: Wildcat: a generic framework for context-aware applications. In: MPAC '05: Proceedings of the 3rd international workshop on Middleware for pervasive and ad-hoc computing, New York, NY, USA, ACM Press (2005) 1–7
19. Keeney, J., Cahill, V.: Chisel: A policy-driven, context-aware, dynamic adaptation framework. In: POLICY '03: Proceedings of the 4th IEEE International Workshop on Policies for Distributed Systems and Networks, IEEE Computer Society (2003) 3–14
20. Romn, M., Hess, C.K., Cerqueira, R., Ranganathan, A., Campbell, R.H., Nahrstedt, K.: Gaia: A middleware infrastructure to enable active spaces. IEEE Pervasive Computing **1** (2002) 74–83
21. Sorensen, C.F., Wu, M., Sivaharan, T., Blair, G.S., Okanda, P., Friday, A., Duran-Limon, H.: A context-aware middleware for applications in mobile ad hoc environments. In: MPAC '04: Proceedings of the 2nd workshop on Middleware for pervasive and ad-hoc computing, New York, NY, USA, ACM Press (2004) 107–110
22. Giarratano, J.C., Riley, G.: Expert Systems: Principles and Programming. Brooks/Cole Publishing Co., Pacific Grove, CA, USA (1989)
23. Strohbach, M., Gellersen, H.W., Kortuem, G., Kray, C.: Cooperative artefacts: Assessing real world situations with embedded technology. In: UbiComp 2004: Ubiquitous Computing, Springer-Verlag (2004) 250–267