Context-Aware Tuples for the Ambient

Christophe Scholliers *, Elisa Gonzalez Boix **, Wolfgang De Meuter, and Theo D'Hondt

Software Languages Lab
Vrije Universiteit Brussel, Belgium
{cfscholl,egonzale,wdmeuter,tjdhondt}@vub.ac.be

Abstract. In tuple space approaches to context-aware mobile systems, the notion of context is defined by the presence or absence of certain tuples in the tuple space. Existing approaches define such presence either by collocation of devices holding the tuples or by replication of those tuples across all devices. We show that both approaches can lead to an erroneous perception of context. The former ties the perception of context to network connectivity which does not always yield the expected result. The latter causes context to be perceived even if a device has left that context a long time ago. We propose a tuple space approach in which tuples themselves carry a predicate that determines whether they are in the right context or not. We present a practical API for our approach and show its use by means of the implementation of a mobile game.

1 Introduction

A growing body of research in pervasive computing deals with coordination in *mobile ad hoc networks*. Such networks are composed of mobile devices which spontaneously interact with other devices within communication range as they move about. This network topology is often used to convey context information to collocated devices [13]. Moreover, such context information can be used to optimize application behaviour given the scare resources of mobile devices [10]. In this paper, we focus on distributed programming abstractions to ease the development of context-aware applications deployed in a mobile environment.

Developing these applications is complicated because of two discriminating properties inherent to mobile ad hoc networks [18]: nodes in the network only have intermittent connectivity (due to the limited communication range of wireless technology combined with the mobility of the devices) and applications need to discover and collaborate without relying on a centralized coordination facility. Decoupled coordination models such as tuple spaces provide an appropriate paradigm for dealing with those properties [10]. Several adaptations of tuple spaces have been specially developed for the mobile environment (including LIME [13], L2imbo [2] and TOTA[9]). In those systems, processes communicate by reading from and writing tuples to collocated devices in the environment. Context information in such systems is thus represented by the ability to *read* certain tuples from the environment. In this paper we argue that this representation is inappropriate and can even lead to an erroneous perception of context.

* Funded by a doctoral scholarship of the IWT-Flanders, Belgium.
** Funded by the Prospective Research for Brussels program of IWOIB-IRSIB, Belgium

The main reason for this is that the ability to read a tuple from the environment does not give any guarantees that the context information carried by the tuple is appropriate for the reader. This forces programmers to manually verify that a tuple is valid for the application's context situation *after* the tuple is read.

In this paper, we propose a novel tuple space approach called *context-aware tuples* which decouples the concept of tuple perception from tuple reception. A context-aware tuple has an associated predicated called a *context rule* that determines when the receiving application is in the right context to perceive the tuple. Only when a tuple's context rule can be satisfied by the context of the receiving application, the tuple can be perceived by the application. Applications can also be notified when the tuple can no longer be perceived. The core contribution of this work lies in the introduction of a general programming concept under the form of a context rule to support development of context-aware applications in a mobile environment. Our contribution is validated by (1) a prototype implementation, (2) demonstrating the applicability of our model by using it in a non-trivial context-aware distributed application and (3) providing an operational semantics for our model.

2 Motivation

Tuple spaces were first introduced in the coordination language Linda [7]. Recently they have shown to provide a suitable coordination model for the mobile environment [10]. In the tuple space model, processes communicate by means of a globally shared virtual data structure (called a tuple space) by reading and writing tuples. A tuple is an ordered group of values (called the tuple content) and has an identifier (called the type name). Processes can post and read tuples using three basic operations: `out` to insert a tuple into the tuple space, `in` to remove a tuple from the tuple space and `rd` to check if a tuple is present in the tuple space (without removing it). Tuples are anonymous and are extracted from the tuple space by means of pattern matching on the tuple content.

In order to describe the main motivation behind context-aware tuples, we introduce a simple yet representative scenario and show the limitations of existing tuple space approaches. Consider a company building where each room is equipped with devices that act as *context providers* of different kinds of information. For example, information to help visitors to orient themselves in the building or information about the meeting schedule in a certain room. Employees and visitors are equipped with mobile devices which they use to plan meetings or to find their way through the building. Since each room is equipped with a context provider, a user located in one room will receive context information from a range of context providers. Only part of this context information which is broadcasted in the ambient is *valid* for the current context of the user.

Figure 1 illustrates the scenario where ω represents the company building, τ a meeting room in the building, and γ the communication range of a device located in the meeting room. The star denotes a tuple space (acting as the context provider) which injects tuples into the ambient, i.e. all devices (depicted

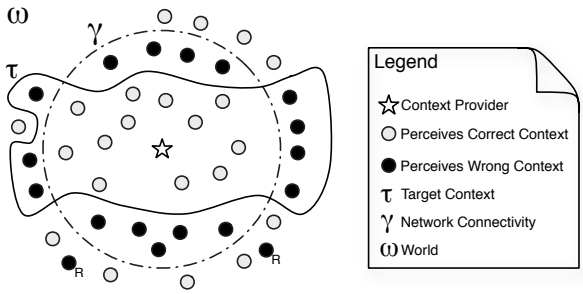

Fig. 1. Context perception in existing tuple space approaches

as dots) $\in \gamma$. Those tuples are aimed to be perceived by devices in the meeting room, i.e. in the target area τ. This device injects a tuple in the ambient to signal receivers that they are currently in the meeting room. Note that location is just one example of context, τ could involve more complex constraints, e.g. being located in the meeting room while there is a meeting.

A large body of tuple space systems targeting a mobile setting follows a *federated* tuple space model [13] in which the visibility of the tuples (and thus context perception) directly depends on collocation of devices holding these tuples. In this model, the perceived context of a device is equivalent to being in range of γ. The context delivery solely based on γ makes two groups of devices to perceive wrong context information (depicted as black dots). The first group consists of devices contained in the set $\gamma \setminus \tau$. In our example, these are all devices within communication range of the context provider but outside the meeting room. These devices will perceive to be in the meeting room while they are actually not. The second group consists of devices contained in the set $\tau \setminus \gamma$. In our example, these are all devices out of communication range of the context provider (possibly due to an intermittent disconnection) but in the meeting room. These devices will perceive not to be in the meeting room while they are actually.

Other tuple space systems have adopted a *replication* model where tuples are replicated amongst collocated devices in order to increase data availability in the face of intermittent connectivity [9, 12]. In replication-based models, devices in $\tau \setminus \gamma$ will not perceive wrong context information. However, in these systems tuple perception is equivalent to have been once within reach of γ, possibly in the past. This means that devices which have been connected to the context provider once (τ) and are currently in $\omega \setminus \tau$ (depicted as black dots with a R) will perceive to be in the meeting room even though they are no longer there.

2.1 Summary

Using current tuple space approaches the context perception is correct in certain cases (the white dots) but, in many cases it is wrong (black dots). There are three main reasons for these erroneous context perceptions. First, there is a connectivity-context mismatch making context sharing based *solely* on connectivity unsuitable for the development of context-aware applications deployed in a mobile setting. Second, the observed context is affected by intermittent connectivity: temporal disconnections with the context provider result in an erroneous

context perception. Third, when using replication-based models to deal with intermittent connectivity, a permanent disconnection leads devices to perceive that they are in the target area forever.

In order to solve these issues, programmers are forced to manually verify that every tuple (and thus context information) is valid for the application' context. More concretely, programmers have to manually determine tuple perception at the application level after a tuple is read from the tuple space. Manually determining the applications context and adapt accordingly leads to context-related conditionals (`if` statements) being scattered all over the program [1], hindering modularity. Additionally, the content of the tuples have to be polluted with meta data in order to infer tuple perception at application level, decreasing reusability of tuples. For example, a `Room` tuple indicating that a person is currently located in the meeting room should also contain the location information. Finally, programmers need to write application level code that deals with context-awareness in order to compensate for the lack of expressiveness of underlying model.

As the complexity of context-aware applications increases, manually computing tuple perception can no longer be solved using ad hoc solutions. Instead, the coordination model should be augmented with abstractions for context-awareness that allow developers to describe tuple perception in the coordination model itself. Context-aware tuples provides an alternative approach that keeps the simplicity of the tuple space model where interactions and context information are defined by means of tuples, while allowing tuples themselves to determine the context in which a receiving application should be in order to perceive a tuple.

3 Context-Aware Tuples

Context-aware tuples is a novel tuple space approach for mobile ad hoc networks tackling the tuple perception issues described. We introduce the notion of a *context rule* prescribing when a tuple should be perceived by the application, and a *rule engine* to infer when a tuple is perceivable and when it is not. Unlike existing tuple space approaches, *only* the subset of tuples which should be perceivable, is made accessible to applications.

The Core Model The model underlying context-aware tuples gathers concepts from both federated and replication tuple spaces, and extends them with a declarative mechanism to express context information with tuples. Figure 2 depicts our model. A device in the network is abstracted as a virtual machine (VM) carrying one or more context-aware tuple space (CAT) systems. CAT systems are connected to other CAT systems by means of a mobile ad hoc network. Several interconnected CAT systems form a *CAT network*. The composition of a CAT network varies according to the changes on the network topology as the hosting device moves about with the user. A CAT system consists of a tuple space and a rule engine.

The tuple space serves as the interface between applications and the CAT system. It supports *non-blocking* Linda-like operations to insert, read and re-

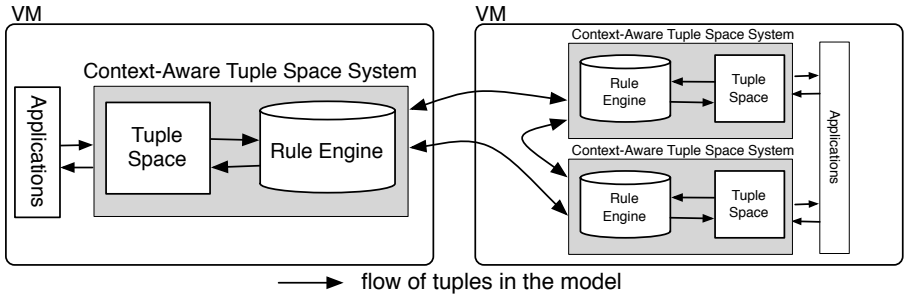

Fig. 2. Context-aware tuple space model

move tuples. The main reason for the strict non-blocking operations is that it significantly reduces the impact of volatile connections on a distributed application[1]. As an alternative to blocking operations, we provide the notion of a *reaction* to a tuple (similar to a LIME reaction [13]): applications can register an observer that is asynchronously notified when a tuple matching a certain template is found in the tuple space.

The tuple space of a CAT system contains two types of tuples. *Public tuples* denote tuples that are shared with remote CAT systems, and *private tuples* denote tuples that remain local to the tuple space in which they were inserted and thus, will not be transmitted to other CAT systems. Applications can insert private and public tuples in the tuple space by means of the `out` and `inject` operation, respectively. As in LIME, applications can access tuples coming from the network without knowing the different collocated CAT systems explicitely.

The rule engine infers when a tuple should be perceived by applications (i.e. when its context rule is satisfied). Before further detailing the role of the rule engine, we describe how tuples are spread across the network.

Distribution of tuples in the network. When two CAT systems discover each other in the network, the public tuples contained in their tuple spaces are cloned and transmitted to the collocated CAT system. Hence, our model replicates tuples to remote CAT systems. Related work has shown that replication increases availability in such a highly disconnected environment allowing parties not to have to be connected at the same time to communicate [12]. In this work, replication is used to support context-aware computation even though context providers and receivers are not connected at the same time. When a CAT system disconnects from the CAT network, the interchanged tuples are still stored in each CAT system allowing applications to perform some computation based on the stored context information despite being disconnected.

Tuples are propagated from CAT to CAT system when they see each other on the network according to a *propagation protocol* (similar to a TOTA propagation rule [9]). Tuples themselves carry a propagation protocol that allows a tuple

[1] From previous work, we have found that a loosely-coupled communication model mitigates the effects of hardware characteristics inherent to mobile ad hoc networks [3].

itself to check whether it should be propagated to a certain CAT system. Such propagation protocol is triggered *before* a tuple is being physically transmitted to a new CAT system avoiding unnecessary exchange of tuples. In this work, the propagation protocol is limited to one-hop neighbours. A description of this scoped propagation mechanism can be found in [16]. Note that other replication techniques could be used [12], and the mechanism for replication is orthogonal to the abstractions for context-awareness introduced in this work.

Managing Tuple Perception. The rule engine is a central component in our model which ensures that applications can only see those tuples that they should perceive. Each tuple inserted in a CAT system carries a *context rule*. A context rule defines the conditions that need to be fulfilled for a tuple to be perceivable. Such context rule is defined by the creator of the tuple and gets transmitted together with the tuple when the tuple is injected in the network.

When a tuple is inserted at a certain CAT system, the tuple is first handed in to the rule engine which installs the necessary machinery to evaluate the tuple's context rule. When the rule engine infers that the conditions on a context rule are satisfied, the tuple's context rule is triggered and said to be *satisfied*. Only when the context rule of a tuple is satisfied, the tuple is inserted in the tuple space of the CAT system. At that moment, the applications are able to read the tuple. The rule engine takes care of reflecting the changes to the receiver' context so that applications cannot perceive those tuples whose context rule is not satisfied. The rule engine combined with the context rule solve the erroneous context perception problems from which replication-based approaches suffer.

As explained in the introduction, context information in tuple space approaches is represented by the ability to read certain tuples from the tuple space. These tuples can be either received from the environment or inserted locally. An example of a locally inserted tuple is a tuple which indicates the user location (e.g. GPS coordinates). As this information is always true, the associated context rule of such a tuple is always satisfied independently of the context. In contrast, a tuple received from the ambient indicating that a user is located in the meeting room needs a custom rule. The rule could specify that, e.g., there should be a location tuple in the receiving tuple space whose coordinates are within the boundaries of the meeting room. Our model conceives a context rule as a set of conditions defined in terms of the presence of certain tuples in the receiving tuple space. The rule engine thus observes the insertion and removal of the tuples in the tuple space to infer which context rules are satisfied. Defining context rules in terms of tuples allows the application to abstract away from the underlying hardware while keeping the simplicity of the tuple space model.

The rule engine incorporates a truth maintenance system built on top of a RETE network [5]. A RETE network optimizes the matching phase of the inference engine providing an efficient derivation of context rule activation and deactivation. The network has also been optimized to allow constant time deletions by applying a *scaffolding* technique [15]. A full description of the engine and its performance is out of the scope of this paper and can be found in [17].

The lifespan of a context-aware tuple. Context rules introduce a new dimension in the lifespan of a tuple. Not only can a tuple be inserted or removed from the tuple space, but it can also be perceivable or not for the application. Figure 3 shows a UML-state diagram of the lifespan of a context-aware tuple. When an application inserts a tuple in a CAT system [2], the tuple is not perceivable and its context rule is asserted in the rule engine. The rule engine then starts listening for the activation of that context rule (CR activation in the figure).

A tuple will become perceivable depending on whether the context rule is satisfied. If the context rule is satisfied, the tuple is perceivable and it is subject to tuple space operations (and thus becomes accessible to the application). If the tuple is not perceivable, the tuple is not subject to tuple space operations but its context rule remains in the rule engine. Every time a tuple's context rule is not satisfied, the out of context listeners for a tuple (OC listeners in the figure) are triggered. Applications can install listeners to be notified when a tuple moves out of context and react to it.

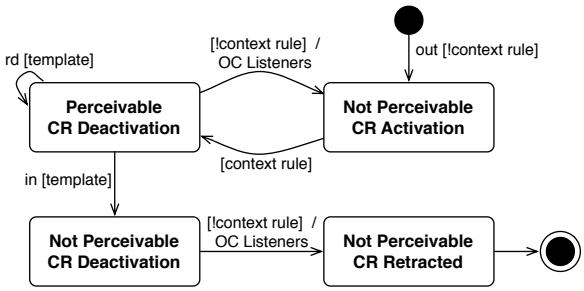

Fig. 3. Lifespan of a context-aware tuple

Upon performing an `in` operation, the tuple is removed from the tuple space but its context is not modified. As such, the tuple is considered not to be perceivable (as it is out of the tuple space) and its context rule remains in the rule engine. Once out of the tuple space, the rule engine listens for the deactivation of the context rule. Once the context rule is no longer satisfied, the context rule is retracted from the rule engine, and the tuple will be eventually garbage collected (once it is no longer referenced by the application).

Coordination. Our model combines replication of tuples for read operations while guaranteeing atomicity for remove operations. Atomicity for remove operations is an essential feature to support synchronization between applications. In our model, applications cannot remove tuples coming from a remote CAT system which is no longer connected. In order for a remove operation to succeed, the CAT system which created and injected the tuple in the network (called the *originator* system) needs to be connected. This means that a remove operation in our approach is executed *atomically* as defined in Linda [7]: if two processes

[2] To keep the figure concise `out` denotes the insertion of a private or a public tuple.

perform a remove operation for a tuple, only one removes the tuple. When an originator is asked to remove one of its (stored) tuples by another CAT system, it removes the tuple and injects an *antituple* in the network for the removed tuple. For every tuple there is (conceptually) a unique antituple with the same format and content, but with a different sign. All tuples injected by an application have positive sign while their antituples have a negative sign. Whenever a tuple and its antituple are stored in the same tuple space, they immediately *annihilate* one another, i.e. they both get removed from the tuple space. By means of antituples, CAT systems can "unsend" tuples injected to the network.

Garbage Collection of Tuples. In our model, a public tuple gets replicated to collocated CAT systems. Some of these tuples may not be used by the receiving CAT system, resulting in accumulation of obsolete tuples. We use two mechanisms to garbage collect tuples in the CAT network. First, all tuples are injected in the network with an associated timeout. Such a timeout denotes the total lifespan of a tuple and it is determined by the application that creates and injects the tuple to the network. A tuple is transmitted together with its timeout to another CAT system. When the time period has elapsed, independently of the state in which a tuple is, the tuple becomes candidate for garbage collection in the CAT system. This means that the tuple context' rule is retracted from the engine, and the tuple is removed from the tuple space if necessary. The transitions for garbage collection have been omitted from figure 3 to keep it clear and concise. Secondly, when a public tuple gets removed, an antituple is sent to those systems that received the removed tuple. If a CAT system cannot be reached, the removal of the tuple is delayed until its timeout elapses.

3.1 Operations

In this section we describe context-aware tuples from a programmer's perspective. Context-aware tuples have been implemented as part of AmbientTalk[3], a distributed object-oriented programming language specifically designed for mobile ad hoc networks [18]. We introduce the necessary syntax and features of the language along with our explanation.

In order to create a CAT system programmers can call the `makeCatSystem` operation as follows:

```
def cat := makeCatSystem();
```

This operation initializes a CAT system (including the rule engine and the tuple space) and publishes it to the *ambient*, i.e. the CAT network. It returns the tuple space of the newly created CAT system. Variables are defined with the keyword `def` and assigned to a value with the assignment symbol (:=).

As mentioned before, all operations for interacting with the tuple space of a CAT system are non-blocking. We provide the `rdp(template)` operation to check if a tuple matching the `template` is present in the tuple space(without

[3] Context-aware tuples is available with AmbientTalk at http://soft.vub.ac.be/amop

removing it), and the `out(tuple)` operation to insert a private `tuple` in the tuple space. In order for applications to insert a public tuple, thereby making it available to other collocated CAT systems, the `inject:` operation is provided:

```
cat.inject: tuple  inContext: contextRule timeout: timeInterval;
```

This operation takes as parameter a tuple and its context rule and a time interval denoting the timeout value for the tuple. A context rule is defined by an array containing the set of templates and constraints that need to be satisfied for the tuple to be perceivable. Constraints are conceived as logical conditions on the variables used in a template. As a concrete usage example, consider again the scenario sketched in the motivation. In order to model that a device is within the meeting room, the context provider could inject a public tuple as follows:

```
cat.inject: tuple(inRoom, meetingRoom)
    inContext: [tuple(location,?loc), withinBoundary(roomArea,?loc)];
```

A tuple is created by means of the `tuple` operation which takes as parameter a list of fields. As usual, the first field of a tuple is its type name. In this case, we create a `inRoom` tuple for the meeting room whose context rule consists of two terms that need to match. First, there must be a tuple in the tuple space matching the `tuple(location,?loc)` template [4]. The `?` operator indicates a variable in a template, i.e. the template matches *any* `location` tuple in the tuple space. Secondly, the `location` tuple needs to satisfy a constraint: its coordinates have to be within the area of the meeting room. The `withinBoundary` function returns such a constraint given the coordinates stored in the `?loc` variable and the meeting room area stored in `roomArea` variable.

Programmers can use reactions to register a block of code that is executed when a tuple matching a template is inserted in the tuple space. Our approach extends a LIME reaction with the notion of *context*: a reaction can only be triggered when the tuple matching the pattern is perceivable. Programmers can also react to a tuple moving out of context by installing an `outOfContext` listener. In what follows, we describe the different kinds of reactions supported.

```
cat.when: template read: closureIn outOfContext: closureOut;
```

The `when:read:` operation takes as parameter a template to observe in the tuple space, and two closures that serve as event handlers to call when the tuple is perceivable and when it is not, respectively. When a perceivable tuple matches the template, the `closureIn` handler is called binding all variables of the template to the values of the matching tuple. As this operation performs a reaction to a `rd` operation, the tuple is not removed from the tuple space. When the context rule of the matching tuple is not satisfied, the `closureOut` handler is called. The `when:read:` operation only triggers the event handlers once for a matching tuple. If several perceivable tuples match the template, one is chosen non-deterministically. The `whenever:read:` operation works analogously but it

[4] A template is created by means of the `tuple` operation as well. However, only templates can take variables as fields.

triggers the event handlers for *every* perceivable tuple matching the template. The code snippet below shows the usage of this operation in our scenario.

```
cat.whenever: tuple(inRoom,?name) read: {
   display("You are in room" + name);
 } outOfContext: {
    display("You moved out of room" + name);
 };
```

In the example, each time an `inRoom` tuple is matched, the application notifies that the user moved in a certain room. Once the user moves out of the boundaries of that room, the `inRoom`'s context rule is not satisfied and the `outOfContext` closure is applied notifying the user that he moved out of the room.

The following two operations work analogously to the previous ones but, they perform a reaction to an `in` operation rather than a `rd` operation.

```
cat.when: template in: closureIn outOfContext: closureOut;
cat.whenever: template in: closureIn outOfContext: closureOut;
```

Those operations remove the tuple from the tuple space before calling the `closureIn` handler. Note that if the tuple to be removed comes from another CAT system, the underlying CAT system contacts the originator CAT system to atomically remove the original tuple. If that removal fails, the replicated tuple is not removed from the local tuple space and `closureIn` is simply not triggered.

4 Semantics

We now formalize the context-aware tuples model by means of a calculus with operational semantics based on prior works in coordination [19, 20]. The syntax of our model is defined by the grammar shown in table 4. k identifies the type of the tuple: $+$ for a public tuple, $\oplus$ for a private tuple and $-$ for an antituple. A context-aware tuple c is specified as a first order term τ. $\tau^k_{x,t}\langle r\rangle$ indicates that the tuple with content τ, type k and timeout t, originates from a tuple space with identifier x and is only perceivable when its context rule r is satisfied. The context rule is considered optional and the notation $\tau^k_{x,t}$ should be read as $\tau^k_{x,t}\langle 1\rangle$, i.e. the context rule is always true. The antituple of a tuple $\tau^k_{x,t}$ is denoted by $\tau^-_{x,t}\langle 0\rangle$, i.e. its context rule is always false.

Table 1. Context-Aware Tuples: Grammar

$k ::= + \mid \oplus \mid -$	Tuple Types
$c ::= \tau^k_{x,t}\langle r\rangle$	Context-Aware Tuple
$S ::= \emptyset \mid c, S$	Tuple Set
$P ::= \emptyset \mid A.P$	Process
$C ::= \emptyset \mid (\llbracket S\rrbracket_x \mid C) \mid (P \mid C)$	Configuration
$A ::= out(x,\tau,r,t) \mid inject(x,\tau,r,t) \mid rd(x,\nu) \mid in(x,\nu) \mid$ $outC(x,\nu) \mid whenRead(x,\nu,P_a,P_d).P \mid whenIn(x,\nu,P_a,P_d).P$	Actions

A process P consists of a sequence of tuple space operations A. Tuples are stored in S which is defined as a set of tuples composed by the operator $(,)$. A tuple space with content S and identifier x is denoted by $[\![S]\!]_x$. A system configuration C is modeled as a *set* of processes P and collocated tuple spaces $[\![S]\!]_x$ composed by the operator $|$. An application consists of all $P \in C$.

Next to the grammar, we assume the existence of a matching function $\mu(\nu, \tau)$ that takes a template ν and a tuple content τ, and returns θ. θ is a substitution map of variable identifiers from the template ν to the actual values from τ. A concrete value in this map can be accessed by θ_z that returns the actual value for z. The matched tuple can be accessed by θ_τ. We also assume the existence of a function *time* which returns a numeric comparable value indicating the current time. $r(S)$ indicates that the context rule r is satisfied in the tuple set S.

The semantics of the context-aware tuples model is defined by the transition rules shown in table 2. Every transition $C \xrightarrow{\lambda} C'$ indicates that a configuration C can be transformed into a configuration C' under the condition λ.

Table 2. Operational Semantics

$$out(x,\tau,r,t).P|[\![S]\!]_x|C \xrightarrow{t'=time()+t} P|[\![\tau^{\oplus}_{x,t'}\ \langle r\rangle, S]\!]_x|C \quad \text{(OUT)}$$

$$inject(x,\tau,r,t).P|[\![S]\!]_x|C \xrightarrow{t'=time()+t} P|[\![\tau^{+}_{x,t'}\ \langle r\rangle, S]\!]_x|C \quad \text{(INJ)}$$

$$[\![\tau^k_{x,t}\ \langle r\rangle, S]\!]_x|[\![S']\!]_y|C \xrightarrow[(x\neq y)]{\tau\notin S'\wedge(k\neq\oplus)\wedge} [\![\tau^k_{x,t}\ \langle r\rangle, S]\!]_x|[\![\tau^k_{x,t}\ \langle r\rangle, S']\!]_y|C \quad \text{(RPL)}$$

$$rd(x,\nu).P|[\![\tau^k_{y,t}\ \langle r\rangle, S]\!]_x|C \xrightarrow[r(S)]{\mu(\nu,\tau)=\theta\wedge(k\neq-)\wedge} P\theta|[\![\tau^k_{y,t}\ \langle r\rangle, S]\!]_x|C \quad \text{(RD)}$$

$$[\![\tau^-_{y,t}\ \langle 0\rangle, \tau^k_{y,t}\ \langle r\rangle, S]\!]_x|C \xrightarrow{(k\neq-)} [\![\tau^-_{y,t}\ \langle 0\rangle, S]\!]_x|C \quad \text{(KILL)}$$

$$[\![\tau^k_{y,t}\ \langle r\rangle, S]\!]_x|C \xrightarrow{t\leq time()\wedge(k\neq-)} [\![\tau^-_{y,t}\ \langle 0\rangle, S]\!]_x|C \quad \text{(TIM)}$$

$$in(x,\nu).P|[\![\tau^k_{x,t}\ \langle r\rangle, S]\!]_x|C \xrightarrow[(k\neq-)]{\mu(\nu,\tau)=\theta\wedge r(S)\wedge} P\theta|[\![\tau^-_{x,t}\ \langle 0\rangle, S]\!]_x|C \quad \text{(INL)}$$

$$in(x,\nu).P|[\![\tau^+_{y,t}\ \langle r\rangle, S]\!]_x|[\![\tau^+_{y,t}\ \langle r\rangle, S']\!]_y|C \xrightarrow[(x\neq y)]{\mu(\nu,\tau)=\theta\wedge r(S)\wedge} P\theta|[\![S]\!]_x|[\![\tau^-_{y,t}\ \langle 0\rangle, S']\!]_y|C \quad \text{(INR)}$$

$$outC(x,\tau).P|[\![\tau^k_{y,t}\ \langle r\rangle, S]\!]_x|C \xrightarrow{!r(S)} P|[\![\tau^k_{y,t}\ \langle r\rangle, S]\!]_x|C \quad \text{(OC)}$$

$$whenRead(x,\nu,P_a,P_d).P|[\![S]\!]_x|C \xrightarrow{1} rd(x,\nu).P_a.outC(x,\theta_\tau).P_d|P|[\![S]\!]_x|C \quad \text{(WR)}$$

$$whenIn(x,\nu,P_a,P_d).P|[\![S]\!]_x|C \xrightarrow{1} in(x,\nu).P_a.outC(x,\theta_\tau).P_d|P|[\![S]\!]_x|C \quad \text{(WI)}$$

The (OUT) rule states that when a process performs an `out` operation over a local tuple space x, the tuple is immediately inserted in x as a private tuple with context rule r and timeout t'. The process continuation P is executed immediately. When a tuple is inserted in the tuple space x with an `inject` operation as specified by (INJ), the tuple is inserted in x as a public tuple and is replicated to other tuple spaces as specified by (RPL). This rule states that when a tuple space y moves in communication range of a tuple space x, all tuples $\tau^k_{x,t}$ which are not private and are not already in y will be replicated to y. The (RD) rule states that to read a template ν from a tuple space x, x has to contain a matching $\tau^k_{y,t}$ and the context rule of τ is satisfied in S. (RD) blocks if one of these conditions is not satisfied. When (RD) does apply, the continuation P is invoked with substitution map θ. Note that we do not disallow x to be equal to y in this rule. The (KILL) rule specifies that when both a tuple τ and its unique antituple τ^- are stored in the same tuple space, τ is

removed immediately. The (TIM) rule specifies that when the timeout of a tuple τ elapses, its antituple τ^- is inserted in the tuple space. The `in` operation is *guaranteed* to be atomically executed. In the semantics, it has been split into a local rule (INL) and a remote rule (INR). (INL) works similarly to (RD), but it removes the tuple $\tau^k_{x,t}$ originated by the local tuple space x and inserts its antituple $\tau^-_{x,t}$. (INR) states that when the `in` operation is matched with a tuple published by another tuple space y, y must be one of the collocated tuple spaces (i.e. be in the configuration). Analogously to (INL), the tuple is removed and its antituple is inserted. The (OC) rule states that to move out of context a tuple τ from a local tuple space x, x has to contain τ (possibly its antituple) and its context rule is *not* satisfied. The WR rule states that a `whenRead` operation performed on the local tuple space x with template ν and processes P_a and P_d, is immediately translated into a new parallel process and the continuation P will be executed. The newly spawned parallel process is specified in terms of performing a `rd` operation followed by an `outC` operation. A `rd` operation blocks until there is a tuple matching ν in the local tuple space. The continuation P_a is then executed to subsequently perform an `outC` which blocks until the tuple is no longer perceivable. Finally, the continuation P_d is invoked whereafter the process dies. The WI is specified analogously but as it models a `whenIn` operation, it performs a `in` operation rather than a `rd` one. The `wheneverRead` and `wheneverIn` operations have been omitted as they are trivial recursive extensions of `whenRead` and `whenIn`, respectively.

Note that ($KILL$) does not remove antituples. This has been omitted to keep the semantics simple and concise. By means of (RPL), the antituple of a tuple τ is only replicated to those systems that received τ. In our concrete implementation if a system cannot be reached, the removal of the antituple is delayed until the timeout of its tuple elapses (which inserts an antituple as specified by (TIM)). An antituple can only be removed once there are no processes in the configuration which registered an `outC` operation on the original tuple.

5 Flikken: Programming with context-aware tuples

We demonstrate the applicability of context-aware tuples by means of the implementation of *Flikken* [5]: a game in which players equipped with mobile devices interact in a physical environment augmented with virtual objects. The game consists of a dangerous gangster on the loose with the goal of earning 1 million euro by committing crimes. In order to commit crimes a gangster needs to collect burgling equipment around the city (knives, detonators, etc). Policemen work together to shoot the gangster down before he achieves his goal. Figure 4 shows the gangster's and a policeman's mobile device at the time the gangster has burgled the local casino. The gangster knows the location of the places with big amounts of money (banks, casinos, etc). When a gangster commits a crime, policemen are informed of the location and the amount of money stolen. Policemen can see the position of all nearby policemen and send messages to each

[5] Flikken (which means *cops* in Dutch) is also included in the AmbientTalk distribution

other to coordinate their movements. The gangster and policemen are frequently informed of each other positions and can shoot at each other.

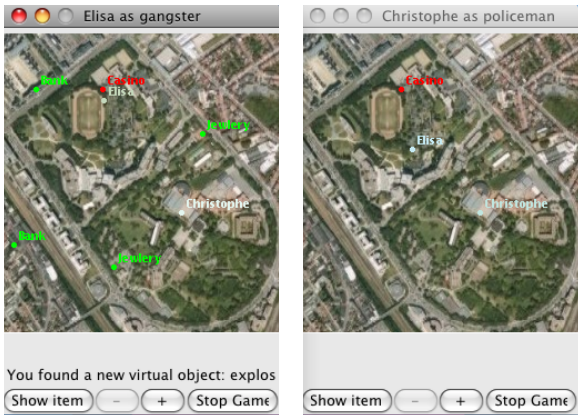

Fig. 4. Flikken GUI on the gangster device (left) and a policeman device (right).

Flikken is an ideal case study for context-aware tuples as it epitomizes a mobile networking application that has to react to context changes on the environment such as changes on player's location, appearance and disappearance of players, and the discovery of virtual objects while moving about. Moreover, how to react to these changes highly depends on the receivers of the context information, e.g. virtual objects representing burgling items should only be perceived by the gangster when he is nearby their location while they should not be perceived at all by policemen. In what follows we describe the coordination and interaction between policemen and the gangster which is fully specified by means of context-aware tuples.

5.1 Implementation

Every player has a CAT system in his mobile device. Once the game starts, policemen and the gangster communicate player to player by means of the CAT network. Throughout the city various context providers (i.e. CAT systems) are placed to inform players about virtual objects or crime locations by injecting the necessary tuples. A special type of context provider is the headquarter (HQ) of the players which signals the start of the chase.

Due to space limitations, this section only describes the set of tuples coordinating the core functionality. Table 3 shows an overview of the tuples used in the game and its context rule. The tuples are divided in five categories depending on the entity that injects them in the environment, i.e. all players, only gangster, only policemen, headquarters and city context providers. As used in the semantics, a tuple is denoted by the term τ and the first element of a tuple indicates its type name. We use capitals for constant values.

Table 3. Overview of the Context-Aware Tuples used in Flikken

Tuple Content	Tuple Context Rule	Tuple Description
	All Players	
τ(TeamInfo, uid, gip)	[true]	Private tuple denoting the player's team.
τ(PlayerInfo, uid, gip, location)	[τ(TeamInfo, ?u, ?team), ?team $\neq$ gip]	Injected to opposite team members every 6 minutes to notify the position of a player. Location is a 2-tuple indicating the (GPS) coordinates of the player.
τ(OwnsVirtualObject, GUN, bullets)	[true]	Private tuple inserted by players when they pick up their gun at their HQ.
	Only The Gangster	
τ(CrimeCommitted, name,location,reward)	[τ(TeamInfo, ?u, POLICEMAN)]	Notifies policemen that the gangster committed a crime.
τ(OwnsVirtualObject, type, properties)	[true]	Private tuple inserted when the gangster picks up a virtual object in the game area.
	Only Policemen	
τ(PlayerInfo, uid, gip, location)	[τ(TeamInfo, ?u, gip)]	Notifies the position of a policemen to his colleagues every time he moves.
	HeadQuarters	
τ(InHeadquarters, location)	[τ(PlayerInfo,?u,?team,?loc), inRange(location, ?loc)]	Notifies that the player entered his HQ. Used to start the chase (when this tuple moves out of context for the gangster's HQ) and to reload policemen's guns.
τ(CrimeTarget, name, location)	[τ(TeamInfo, ?u, GANGSTER)]	Notifies the gangster of the position of crime targets.
τ(CommitCrime,name, location,reward, vobj)	[τ(TeamInfo, ?u, GANGSTER), τ(PlayerInfo,?u, GANGSTER,?loc), inRange(location,?loc), hasVirtualObjects(vobj)]	Notifies the gangster of the possibility of committing a crime. `hasVirtualObjects` takes an array of virtual object ids and checks that the gangster has the required `OwnsVirtualObject` tuples.
	City Context Providers	
τ(VirtualObject, id, location)	[τ(TeamInfo, ?u, GANGSTER), τ(PlayerInfo,?u, GANGSTER,?loc), inRange(location, ?loc)]	Notifies the gangster of the nearby presence of a virtual object. `inRange` is a helper function to check that two locations are in euclidian distance.
τ(Rechargeable-VirtualObject,GUN, BULLETS)	[τ(InHeadQuarters,?loc), τ(OwnsWeaponVO,GUN,?bullets), ?bullets $<$ BULLETS]	Represents the player's gun. The gangster gets only one charge at the start of the game, while policemen's guns are recharged each time they go back to their HQ.

The CAT system on the player's device carries a vital private tuple τ(TeamInfo, uid, gip) indicating to which team he belongs. Every player transmits its location to the CAT network by means of the tuple τ(PlayerInfo, uid, gip, location). These tuples are often used in other tuple's context rules to identify the current whereabouts of a player and his team. For example, the tuple implementing a grenade uses them as follows.

```
cat.inject: tuple(VirtualObject, grenade, location)
inContext: [tuple(TeamInfo, ?u, GANGSTER),
            tuple(PlayerInfo, ?u, GANGSTER, ?loc),
            inRange(location, ?loc) ]
```

The tuple τ(VirtualObject, grenade, location) should be only perceived if the receiver is a gangster whose location (given by `?loc` in the `PlayerInfo` tuple) is physically proximate to the virtual object. The `inRange` function returns the constraint that checks if the gangster location is in euclidian distance with the location of the grenade (stored in `location`). Upon removal of a `VirtualObject` tuple, a private tuple τ(OwnsVirtualObject, object) is inserted in his CAT sys-

tem. `OwnsVirtualObject` tuples are used in the context rule of `CommitCrime` tuples which notify the gangster of a crime that can be committed. As crimes can only be committed when the gangster has certain burgling items, the context rule of the `CommitCrime` tuple requires that certain `OwnsVirtualObject` tuples are present in the tuple space. For example, in order for the gangster to perceive the `CommitCrime` tuple for the `grandCasino`, a τ(OwnsVirtualObject,grenade) tuple is needed as shown below.

```
cat.inject: tuple(CommitCrime, grandCasino, location, reward)
inContext: [tuple(TeamInfo, ?u, GANGSTER),
            tuple(PlayerInfo, ?u, GANGSTER, ?loc),
            inRange(location, ?loc),
            tuple(OwnsVirtualObject,grenade)];
```

Each player also registers several reactions to (1) update his GUI (e.g. to show the `OwnsVirtualObject` tuples collected), and (2) inject new tuples in response to the perceived ones, e.g. when a gangster commits a crime, he injects a tuple τ(CrimeCommitted, name,location,reward) to notify policemen. The code below shows the reaction on `PlayerInfo` tuples installed by the application.

```
cat.whenever: tuple(PlayerInfo, ?uid, ?tid, ?location) read: {
   GUI.displayPlayerPosition(tid, uid, location);
} outOfContext: {
   //grey out a player if there exists no update of his coordinates.
   def tuple := cat.rdp(tuple(PlayerInfo, uid, tid, ?loc));
    if: (nil == tuple) then: { GUI.showOffline(uid) };
};
```

Whenever a `PlayerInfo` tuple is read, the player updates his GUI with the new location of that player. As `PlayerInfo` tuples are injected with a timeout, they are automatically removed from the tuple space after its timeout elapses triggering the `outOfContext:` handler. In the example, this handle greys out the GUI representation of a player if no other `PlayerInfo` tuple for that player is in the CAT system. If the `rdp` operation does not return a tuple, the player is considered to be offline as he did not transmit his coordinates for a while.

5.2 Discussion

Flikken demonstrates how context-aware tuples aid the development of context-aware applications running on mobile ad hoc networks and address the tuple perception issues described in section 2 by introducing two key abstractions: (i) the context rule of a tuple which determines the context in which a receiving CAT system should find itself in order to perceive the tuple, and (ii) the rule engine which takes care of inferring tuple perception before applications are exposed to the tuple. A tuple space model with such abstractions has the following benefits:

1. Decomposing a tuple into content and context rule leads to separation of concerns, increasing modularity.
2. Since context rules can be developed separately, it enables programers to reuse the rules to build different kinds of tuples, increasing reusability. For example, in Flikken, we used a `inRangeOfGangster(?loc)` function to build the rule for the different `VirtualObject` tuples which was also reused to build the three first conditions of `CommitCrime` tuples.

3. Programmers do not need to add computational code to infer tuple perception as the rule engine takes care of it in an efficient way, making the code easier to understand and maintain.

Flikken is a significant subset of an augmented reality game inspired by the industrial game The Target[6]. The game functionality counts 11 tuples (8 of which carry a custom context rule) and 7 reactions. Currently, the game only provides one kind of virtual object for the player's defense, namely a gun. As future work, we would like to extend it with more complex virtual objects like mines, bombs, bulletproof vests, and radio jammers (to disrupt the communication with nearby players). We would also like to enhance the game interaction by incorporating compass data from the mobile device to be able to aim and kill players in a certain trajectory rather than using a certain radius of action.

The AmbientTalk language and context-aware tuples run on J2SE, J2ME under the connected device configuration (CDC), and Android 1.6 Platform. The current discovery mechanism is based on multicast messaging using UDP. Our current experimental setup for Flikken is a set of HTC P3650 Touch Cruise phones communicating by means of TCP broadcasting on a wireless ad hoc WiFi network. As future work, we aim to port the current Java AWT GUI of Flikken to the Android platform to deploy the game on HTC Hero phones.

6 Related Work

In this section, we discuss related systems modeled for context-awareness and show how context-aware tuples differs from them. Most of the tuple space systems designed for a mobile environment can be divided into federated and replication tuple space models. Both suffer from the problems shown in Section 2.

In TOTA, tuples themselves decide how to replicate from node to node in the network. As tuples can execute code when they arrive at a node, they can be exploited to achieve context-awareness in an adaptive way. But, programming such tuples has proven to be difficult [9]. TOTA, therefore, provides several basic tuple propagation strategies. None of these propagation strategies addresses the perception problem tackled by our approach. Writing context-aware tuples in TOTA would require a considerable programming effort to react on the presence of an arbitrary combination of tuples as it only allows reactions on a single tuple.

GeoLinda [14] augments federated tuple spaces with a geometrical read operation `read(s,p)`. Every tuple has an associated shape and the `rd` operation only matches those tuples whose shape intersects the addressing shape `s`. GeoLinda has been designed to overcome the shortcomings of federated tuple spaces for a small subset of potential context information, namely the physical location of devices. As such, it does not offer a general solution for context-aware applications. In contrast, we propose a general solution based on context rules, which allows programmers to write application-specific rules for their tuples. Moreover, in GeoLinda the collocation of devices still plays a central role for tuple perception which can lead to erroneous context perception.

[6] http://www.lamosca.be/en/the-target

EgoSpaces provides the concept of a *view*, a declarative specification of a subset of the ambient tuples which should be perceived. Such views are defined by the *receiver* of tuples while in context-aware tuples it is the other way around. Context-aware tuples allow the *sender* of a tuple to attach a context rule dictating the system in which state the receiver should be in order to perceive the tuple. EgoSpaces suffers from the same limitations as federated tuple spaces since at any given time the available data depends on connectivity [8].

Publish/subscribe systems are closely related to tuple spaces as they provide similar decoupling properties [4]. Context-aware publish subscribe (CAPS) [6] is the closest work as it allows certain events to be filtered depending on the context of the receiver. More concretely, the publisher can associate an event with a *context of relevance*. However, CAPS is significantly different from context-aware tuples. First, CAPS does not allow reactions on the removal of events, i.e. there is no dedicated operation to react when an event moves out of context. Moreover, it is not a coordination abstraction, i.e. atomic removal of events is not supported. And last, their context of relevance are always associated to physical space.

The Fact Space Model [11] is a LIME-like federated tuple space model that provides applications with a distributed knowledge base containing logic facts shared among collocated devices. Unlike context-aware tuples, rules in the Fact Space Model are not exchanged between collocated devices and are not bound to facts to limit the perception of context information.

7 Conclusion

We have introduced a novel tuple space approach in which a tuple itself carries a predicate, called a *context rule*, that determines when the receiving application is in the right context to perceive the tuple. The novelty of our approach lies in the use of context rules combined with the introduction of a rule engine in the tuple space system which takes care of inferring when a context rule is satisfied, to control which tuples present in the tuple space should be actually accessible by applications. This decouples tuple reception from tuple perception solving the context perception problems exhibited by existing tuple space systems. By decomposing a tuple into content and context rule we allow the programmer to separate concerns. Since context rules can be developed separately, it enables programers to reuse the rules to build different kinds of tuples. Programmers do not longer need to infer tuple perception manually as the rule engine takes care of it in an efficient way, making the code easier to understand and maintain.

Acknowledgments The authors would like to thank Amy L. Murphy for her helpful comments on earlier versions of the paper, and Bruno De Fraine for his invaluable help with the formal semantics of our model.

References

1. P. Costanza and R. Hirschfeld. Language constructs for context-oriented programming: an overview of contextl. In *DLS '05*, pages 1–10, NY, USA, 2005. ACM.

2. N. Davies, A. Friday, S.P. Wade, and G.S. Blair. L2imbo: a distributed systems platform for mobile computing. *ACM Mob. Netw. and Appl.*, 3(2):143–156, 1998.
3. J. Dedecker, T. Van Cutsem, S. Mostinckx, T. D'Hondt, and W. De Meuter. Ambient-oriented Programming in Ambienttalk. In *the European Conf. on Object-oriented Progr. (ECOOP)*, volume 4067 of *LNCS*, pages 230–254. Springer, 2006.
4. P. Th. Eugster, Pascal A. Felber, R. Guerraoui, and A.Kermarrec. The many faces of publish/subscribe. *ACM Computing Survey*, 35(2):114–131, 2003.
5. Charles L. Forgy. Rete: A fast algorithm for the many pattern/many object pattern match problem. In J. Mylopoulos and M. L. Brodie, editors, *Artificial Intelligence & Databases*, pages 547–557. Kaufmann Publishers, INC., San Mateo, CA, 1989.
6. D. Frey and G-C. Roman. Context-aware publish subscribe in mobile ad hoc networks. In *Proc. of the 9th Int. Conf. on Coordination Models and Languages (COORDINATION)*, volume 4467 of *LNCS*, pages 37–55. Springer, June 2007.
7. D. Gelernter. Generative communication in Linda. *ACM Transactions on Programming Languages and Systems*, 7(1):80–112, Jan 1985.
8. C. Julien and G.-C. Roman. Active coordination in ad hoc networks. In *Proc. of the 6th Inter. Conf. on Coordination Models and Languages (COORDINATION)*, volume 2949 of *LNCS*, pages 199–215. Springer, February 2004.
9. M. Mamei and F. Zambonelli. Programming pervasive and mobile computing applications with the TOTA middleware. In *IEEE Int. Conf. on Pervasive Computing and Communications (PERCOM)*, page 263. IEEE Computer Society, 2004.
10. C. Mascolo, L.Capra, and W. Emmerich. Mobile Computing Middleware. In *Advanced lectures on networking*, pages 20–58. Springer-Verlag, 2002.
11. S. Mostinckx, C. Scholliers, E. Philips, C. Herzeel, and W. De Meuter. Fact spaces: Coordination in the face of disconnection. In *Conf. on Coord. Models and Lang. (COORDINATION)*, volume 4467 of *LNCS*, pages 268–285. Springer-Verlag, 2007.
12. Amy L. Murphy and G.P Picco. Using lime to support replication for availability in mobile ad hoc networks. In *8th Int. Conf. on Coordination Models and Languages (COORDINATION)*, volume 4038 of *LNCS*, pages 194–211. Springer-Verlag, 2006.
13. Amy L. Murphy, G.P. Picco, and G.-C. Roman. LIME: A middleware for physical and logical mobility. In *Proceedings of the The 21st International Conference on Distributed Computing Systems*, pages 524–536. IEEE Computer Society, 2001.
14. J. Pauty, P. Couderc, M. Banatre, and Y. Berbers. Geo-linda: a geometry aware distributed tuple space. In *Proc. of the 21st Inter. Conf. on Advanced Networking and Applications (AINA)*, pages 370–377. IEEE Computer Society, 2007.
15. Mark Perlin. Scaffolding the RETE network. In *International Conference on Tools for Artificial Intelligence*, pages 378–385. IEEE Computer Society, 1990.
16. Christophe Scholliers, Elisa Gonzalez Boix, and Wolfgang De Meuter. TOTAM: Scoped Tuples for the Ambient. In *Workshop on Context-aware Adaptation Mechanisms for Perv. and Ubiquitous Services*, volume 19, pages 19–34. EASST, 2009.
17. Christophe Scholliers and Eline Philips. Coordination in volatile networks. Master's thesis, Vrije Universiteit Brussels, 2007.
18. T.Van Cutsem, S. Mostinckx, E. Gonzalez Boix, J. Dedecker, and W. De Meuter. Ambienttalk: object-oriented event-driven programming in mobile ad hoc networks. In *Int. Conf. of the Chilean Comp. Science Society*, pages 3–12. IEEE C. S., 2007.
19. M. Viroli and M. Casadei. Biochemical tuple spaces for self-organising coordination. In *Proc. of the Inter. Conf. on Coordination Models and Languages (COORDINATION)*, pages 143–162. Springer-Verlag, 2009.
20. M. Viroli and A. Omicini. Coordination as a service. *Fundamenta Informaticae*, 73(4):507–534, 2006.