A Hybrid Visual Dataflow Language for Coordination in Mobile Ad Hoc Networks

Andoni Lombide Carreton, Theo D'Hondt

A Hybrid Visual Dataflow Language for Coordination in Mobile Ad Hoc Networks

Andoni Lombide Carreton* and Theo D'Hondt

Software Languages Lab
Vrije Universiteit Brussel, Pleinlaan 2 1050 Brussel, Belgium
{alombide} {tjdhondt}@vub.ac.be

Abstract. Because of the dynamic nature of mobile ad hoc networks and the applications running on top of them, these applications have to be conceived as event-driven architectures. Such architectures are hard to program because coordination between concurrent and distributed mobile components has to be expressed by means of event handlers or callbacks. Applications consisting of disjoint event handlers that are independently triggered (possibly by their environment) exhibit a very implicit control flow that is hard to grasp. This paper presents a visual dataflow language tailored towards mobile applications to express the interaction between mobile components that operate on data streams. By using a visual dataflow language as a separate coordination language, the coarse grained control flow of a mobile application can be specified visually and separately from the fine grained control flow. In its turn, this allows a very explicit view on the control flow of the entire mobile application.

Key words: dataflow programming, coordination languages, visual programming, mobile ad hoc networks

1 Introduction

When developing pervasive applications for mobile ad hoc networks, the programmer has to deal with a number of characteristics of both the underlying network infrastructure and the applications running on top of them.

1. Devices in mobile networks often experience intermittent connectivity with nearby peers. Because devices are mobile, they can move out of and back into range of each other at any point in time. Hence, connections between devices are volatile.
2. Applications deployed on mobile ad hoc networks cannot rely on fixed infrastructure such as servers.
3. Mobile applications moving through different ad hoc networks should be able to discard unavailable services and find replacement services at runtime. Applications should remain functional on roaming devices.

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

4. Services offered by nearby devices in the network should be discovered at runtime and trigger the appropriate actions without requiring prior knowledge about these devices or services.
5. The data that is interchanged between different parties often takes the form of a stream, for example a stream of sensor readings or a stream of scanned RFID tags.

These characteristics make it impossible to structure pervasive applications as monolithic programs which accept a fixed input and compute it into some output. Instead, to allow responsiveness to changes in the mobile ad hoc network, programming paradigms targeting pervasive applications propose the adoption of event-driven architectures [1–4]. In such event-driven architectures, the programmer no longer steers the application's control flow explicitly. Rather, control is handed over to the application logic whenever an event is detected by means of callbacks. By adopting such an event-driven architecture, the application logic becomes scattered over different event handlers or callbacks which may be triggered independently [5]. This is the phenomenon known as *inversion of control* [6]. Control flow among event handlers has to be expressed implicitly through manipulation of shared state. Unlike subsequent function calls, code triggered by different event handlers cannot use the runtime stack to make local variables visible to other executions (*stack ripping* [7]), such that these variables have to be made instance variables, global variables, etc. This is why in complex systems such an event-driven architecture can become hard to develop, understand and maintain [8, 9]. Coordination of distributed and concurrent activities can be done on a higher level by means of a separate coordination language. However, current coordination languages lack support to deal with all the characteristics of mobile ad hoc network applications pointed out above.

The visual dataflow language presented in this paper is geared towards applications running on mobile ad hoc networks in the following ways:

1. Application components that move out of range are either treated as temporarily or permanently disconnected. In the latter case, replacement components can be discovered and automatically plugged into the distributed application.
2. Mobile application components are dynamically discovered based on broadcasted role names - acting as topics in a decentralized publish/subscribe architecture - that describe their behavior, and require no additional infrastructure to discover each other.
3. Distributed application components interact by means of reactive scripts that propagate events to dataflow variables and depend on their own set of dataflow variables. This is a straight-forward interface that allows such distributed components to be made dependent on changes in their environment without relying on an explicit callback-style that would introduce the problems mentioned above.
4. The basic dataflow coordination model is extended with infrastructure to allow different strategies of dataflow event propagation tweaked towards a mobile ad hoc network setting, offering one-to-one, one-to-many, many-to-one,

and many-to-many communication among a volatile set of communication partners.

1.1 Coordination in Mobile Ad Hoc Networks

Gelernter and Carriero [10] argue that a complete programming model consists of both a *computation model* and a *coordination model*. The computation model allows programmers to build a single computational activity (e.g., a process, a thread, an actor in an actor language). The coordination model is the glue that binds separate activities into a concurrent application. An ordinary computation language embodies a computation model. Different concurrent languages provide in addition a coordination model that ranges over different levels of abstraction, from manual thread creation and locking to event-based communication among distributed processes. An example of the latter will be discussed in Section 2. A coordination language embodies a coordination model; it provides operations to *create* computational activities and to support *communication* among them. We require the coordination model to be applicable in mobile ad hoc networks, meaning that the model should be resilient to network partitioning and reactive to network topology changes. In the remainder of this section, we discuss some related work in the form of existing coordination and visual (dataflow) languages.

Coordination languages Thanks to their decoupling both in space and time of the different communicating processes, coordination based on tuple spaces is quite popular for mobile ad hoc network applications. Both the LIME [3] and TOTA [11] middleware implement variations on the original tuple space model targeted towards mobile ad hoc network applications. In both systems however, devices respond to the appearance of such tuples by registering *reactions*. Reactions are an advanced form of callbacks, where the callback gets executed as soon as a tuple in the tuple space is successfully pattern-matched.

Reo [12] is a glue language that allows the orchestration of different heterogeneous, distributed and concurrent software components. Reo is based on the notion of mobile channels and has a coordination model wherein complex coordinators, called connectors, are compositionally built out of simpler ones (where the simplest ones are channels). These different types of connectors hierarchically coordinate their constituent connectors, which eventually coordinate the software components that they interconnect. The mobility of the channels refers in this case to a user-invoked migration of a software component along with all its connected channels to a different host. This is not the automatic runtime adaptation to the frequently changing network topology in a mobile ad hoc network that we require.

Visual languages LabVIEW was the first software program to include graphical, iconic programming techniques to make programming more transparent and the sequence of processing visible to the user [13]. LabVIEW is based on the G visual dataflow language and the concrete implementation in the LabVIEW

environment is primarily used for data acquisition, processing and monitoring in a lab setting. LabVIEW does not use dataflow for expressing distribution and/or parallelism, but for the graphical composition of software components that interact with lab hardware.

Another language that does use dataflow for expressing distribution and concurrency is Distributed Prograph [14]. Distributed Prograph is very similar to our approach in the sense that the program code of dataflow operators is dynamically sent to remote processing units. The scheduling of the execution of these operators happens at runtime, but the processing units themselves have to be known at compile time, which is unrealistic in mobile ad hoc networks. In our approach, the processing units are dynamically discovered at runtime in the mobile ad hoc network.

NesC nesC [15] is a programming language for networked embedded systems (such as sensor networks) offering event-driven execution, a flexible concurrency model, and component-oriented application design. These components, however, are statically linked to each other via their interfaces. This makes nesC programs better statically analyzable, but restricts the language to networks of static devices instead of entirely mobile networks.

2 Fine-grained Programming with AmbientTalk

The visual dataflow language that we propose in this paper is a hybrid language: it uses a host language to implement the dataflow operators and allows expressing the coordination among these operators visually. In our case, this host language is AmbientTalk [16, 17], a distributed scripting language embedded in Java that can be used to compose Java components which are distributed across a mobile ad hoc network. The language is developed on top of the J2ME platform and runs on handheld devices such as smart phones and PDAs. Even though AmbientTalk is embedded in Java, it is a separate programming language. The embedding ensures that AmbientTalk applications can access Java objects running in the same JVM. These Java objects can also call back on AmbientTalk objects as if these were plain Java objects. The most important difference between AmbientTalk and Java is the way in which they deal with concurrency and network programming. Java is multithreaded, and provides both a low-level socket API and a high-level RPC API (Java RMI) to enable distributed computing. In contrast, AmbientTalk is a fully event-driven programming language. It provides only event loop concurrency [18] and distributed object communication by means of asynchronous message passing, which are briefly illustrated below. Event loops deal with concurrency similar to GUI frameworks (e.g. Java AWT or Swing): all concurrent activities are represented as events which are handled sequentially by an event loop thread.

To be synchronized with changes in one's environment, AmbientTalk uses a classic event-handling style by relying on closures to function as event handlers. This has two advantages: firstly closures can be used in-line and can be nested and secondly closures have access to their enclosing lexical scope. Event handlers

are (by convention) registered by a call to a function that starts with **when**. The following code snippet illustrates how AmbientTalk can be used to discover a `LocationService` and `WeatherService` in the ad hoc network.

```
1  when: LocationService discovered: { |locationSvc|
2   when: locationSvc<-getLocation(gpsModule.getCoordinates())
3    becomes: { |myLocation|
4     when: WeatherService discovered: { |weatherSvc|
5      when: weatherSvc<-getWeather(myLocation)
6       becomes: { |weatherInfo|
7        GUI.updateWithWeatherInfo(weatherInfo);
8       }}}}
```

Once the `LocationService` is discovered, it is sent a message along with the current GPS coordinates to determine the current location of the user. As soon as a reply is received, the lookup for the `WeatherService` starts. When such a service is discovered, it is sent the `getWeather` message along with the current location that was received from the `LocationService`.

The above code consists of four event handlers. The first event handler, registered by means of the **`when:discovered:`** control structure, is invoked when the language runtime discovers a `LocationService` component. Here, `LocationService` refers to a Java interface. The discovered object is accessible via the `locationSvc` variable, which denotes a remote AmbientTalk object that wraps a Java component implementing the weather service. The syntax `obj<-msg()` denotes an asynchronous message send and is used here to query the `LocationService` object for the current location of the user (e.g., city) given her GPS coordinates.

When the query message is received by the remote `locationSvc` object, that object's `getLocation` method is invoked. The return value of this method is used as the reply to the query. This reply is signaled asynchronously to the caller. The **`when:becomes:`** control structure is used to install an event handler that can process this reply. The return value is passed to this event handler (cf. the `myLocation` variable in the example). As soon as this value is received, this event handler registers two new event handlers (following the same pattern) to query a `WeatherService` about the weather at `myLocation`. Therefore, as soon as the `WeatherService` signals a reply, the user interface is updated.

As can be seen from the above example, service discovery and replies to remote queries (to causally connect the program with the outside world) are represented in AmbientTalk as events that trigger the appropriate event handlers. Care must be taken when coordinating and synchronizing asynchronous invocations: nesting callbacks (like in the example presented above) introduces simple synchronization (the discovery of the `WeatherService` only starts when a `LocationService` is discovered and has replied to the `getLocation` message), but more complex synchronization and coordination patterns require more complicated structures (the lookup of a `WeatherService` could happen, for example, in parallel without waiting for the `LocationService` to reply). While in this sim-

ple example the control flow remains apparent enough to understand, the control flow of large-scale event-driven applications can quickly become puzzling.

3 Coordinating Distributed AmbientTalk Components Using a Visual Dataflow Language

Before the Von Neumann architectures took over the parallel programming world as well, dataflow languages were popular to program massively parallel systems that used a dataflow hardware architecture [19]. By making data dependencies explicit, these languages and hardware architectures allowed a high degree of parallelization while preventing race conditions and other problems arising when parallelizing programs intended for Von Neumann architectures. Moreover, the resulting dataflow graphs are easy to visualize, allowing a visual representation of the data dependencies and the coordination they imply on the different parallel components of the application. The dataflow coordination model can be informally described as follows:

- Dataflow programs consist of *dataflow operators* that take a number of input values and return a single output value. These dataflow operators are best compared to functions or procedures in functional or imperative programming languages that always run in parallel.
- Dataflow operators communicate with each other over *dataflow edges*. These edges represent data dependencies and always flow from the output of a dataflow operator (corresponding to its output value) to one of the inputs of a dataflow operator (corresponding to one if its input values).
- When a dataflow operator is fired depends on the concrete coordination model used. Some languages only fire dataflow operators once as soon as *all* its input values have received a value. Other languages repeatedly fire the dataflow operator as soon as *one* of its input values received a *new* value

Such a coordination model allows that different dataflow operators in the dataflow graph can execute in parallel as long as their data dependencies are satisfied. For example a number of operators in a pipeline execute in parallel when the first operator is fed a stream of data. In such a pipeline the first operator is being applied to new data from the stream while operators later in the sequence are being applied to data already processed by earlier dataflow operators in the pipeline.

Currently, the dataflow paradigm is mostly used in the form of the *coarse-grained* dataflow model, as can be seen from the systems discussed in Section 1.1. In such models, the dataflow paradigm is used to orchestrate the control flow between different modules (possibly running in parallel) that can be of an arbitrary level of abstraction, usually implemented in a conventional programming language. When looking at the characteristics and requirements of mobile networks and the applications running on top of them (discussed earlier in Section 1), we have observed that the dataflow model may provide a very suitable coordination model for this kind of applications. These applications consist of

different distributed components running in parallel that in many cases have to be invoked whenever some external data is fed to them (event-driven architectures). Hence, the driving force for program execution in such applications is not the control flow, which is explicitized by the order of statements in an imperative textual program, but the data flow, which is implicit in an imperative textual program. Furthermore, in many cases these data come in the form of streams, such as continuous sensor readings. These observations, combined with the basic coordination model of the AmbientTalk programming language, led us to the integration of AmbientTalk with a coarse grained dataflow coordination model, that explicitizes the data flow, and a graphical language embodying this model. The main advantage above implementing everything in plain AmbientTalk, is that the coarse-grained control flow, which in AmbientTalk would become very implicit in a complex interplay of different event handlers, is now represented in a very explicit visual notation based on the dataflow coordination model. In this section, this visual dataflow language is explained.

3.1 Visual Dataflow Programming

Figure 1 shows the general idea behind our visual dataflow language. The languages uses the boxes-and-arrows notation to denote dataflow operators and dataflow dependencies between them respectively. The identifier before the `->`

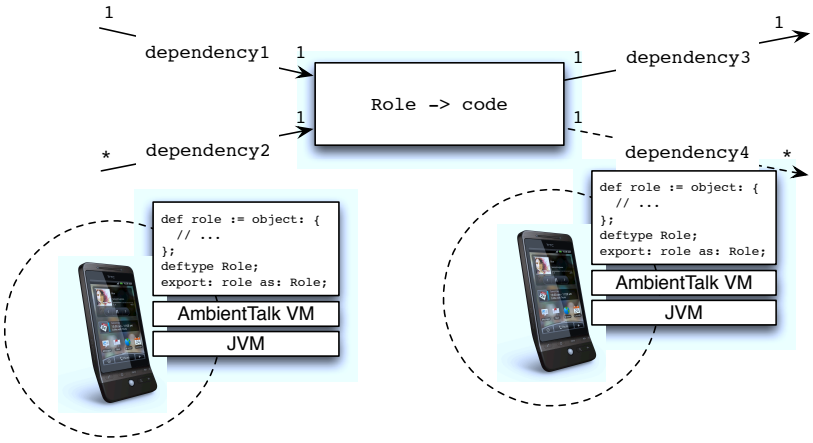

Fig. 1. Basic architecture of a dataflow program

symbol denotes the *role* of the operator (for now, it suffices to think of a role as a procedure name). The code after the `->` symbol can be any list of AmbientTalk expressions and comprises the code of the dataflow operator, which serves as its implementation. This code can be parametrized by variables that are bound to the input values of the dataflow operator. This is achieved by naming the edges, such that these names can be used as the names of the dataflow parameters in the dataflow operator implementation. The dataflow operator firing rule in our

visual languages is the following: *any new value propagated along an incoming dataflow edge results in reapplying the dataflow operator with the new value of the dataflow variable.*

Executing a dataflow program happens by distributing the dataflow operator code to devices that match the roles designated to the operators in the graph and installing communication channels that represent the dependency edges between them. Dependency edges can be either *fixed* (uninterrupted lines) or *rebinding* (dashed lines). The service discovery needed for this is further explained in Section 3.2. For this, these devices should have an AmbientTalk virtual machine running on top of a Java virtual machine, and additionally host the necessary library code to execute their role code. The code associated with a role is mobile AmbientTalk code that can call any AmbientTalk or Java library code that is made visible to it by the device. This is further explained in Section 3.3.

Finally, to cater for group communication in mobile ad hoc networks, we extended the basic dataflow coordination model with dependency arities that allow dataflow dependencies to be one-to-one, one-to-many, many-to-one, many-to-many. This is indicated by the programmer by changing the annotations at the start point or end point of the graph edges and is further explained in Section 3.5.

Before going further into detail, we sketch a small scenario. Envision the shop of the future where every product is tagged with an RFID tag. Customers pick up these products and carry them to certain locations in the shop where the tagged products can be scanned. In a CD shop for example, one could install an RFID reader below the listening stations where customers can listen to the albums they are carrying before buying them. Based on the list of scanned products, the device scanning the products requests a list of recommended products from a remote party. In the CD shop demo application that we have implemented this remote party is the LastFM web service[1]. The resulting list of recommended products is passed to a software service representing the shop and is filtered to only contain the products that are currently in stock, extended with their location in the shop (based on products that are scanned by RFID devices in the shop's shelves). Given these three pieces of information (the list of scanned products carried by the user, the list of recommended products, and the list of recommended products available in the shop), a small application on the user's PDA or smartphone is updated to show this information and help the user in getting the products he wants. As soon as the user removes a product from the range of the device scanning the products (for example by putting it back in the shelves) or brings another product in range of the device, this change is reflected on the application running on his PDA or smartphone.

In this scenario, we consider the user's PDA or smartphone a mobile device moving throughout the shop and being dynamically discovered by multiple RFID readers running the dataflow program. Interaction between the user's mobile device and other devices present in the shop happens entirely spontaneous over wireless connections that can break at any point in time when the user moves

[1] `http://www.last.fm/api/intro`

out of range of one of the components. However, such intermittent connections as the the user roams the shop should be tolerated and not break the application. The continuous flow of RFID data generated by the different components of the application is usually processed by representing this data as data streams [20, 21]. The screenshot shown in figure 2 shows the scenario implemented in our

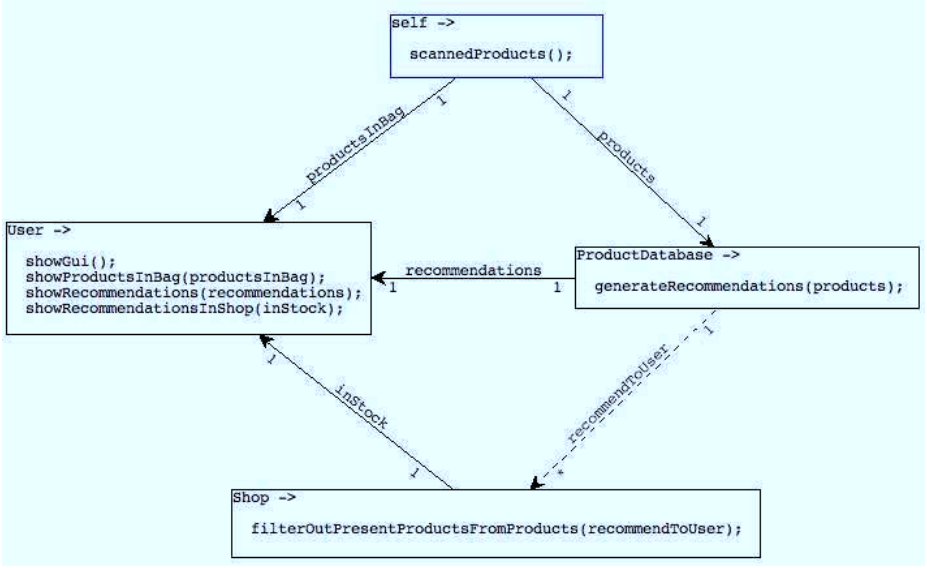

Fig. 2. AmbientTalk visual dataflow example

visual dataflow language. In this example, the Java package paths to the methods that are invoked in the code are omitted for the sake of brevity, but other than that the program shown here is entirely functional. Dataflow operators are represented as boxes while the directed edges connecting these boxes represent data dependencies.

3.2 Discovering Operator Nodes

Our visual dataflow language serves the purpose of coordinating application components running in parallel and distributed over a mobile network. Because the devices hosting these application components can move out of range and back into range of each other at any point in time, our dataflow engine has to discover these application components at runtime without relying on naming servers or other fixed infrastructure. In our visual dataflow language, mobile application components that play a role in a dataflow program are discovered based on their roles. These roles actually have the same use (and in fact are implemented this way) as the Java interfaces that are used by AmbientTalk to discover remote objects and hence act as the *topics* that are being used by the underlying distributed publish/subscribe architecture. The system uses UDP

broadcasting to both advertise dataflow nodes and devices willing to execute one or more of the roles in the dataflow program. This an entirely decentralized approach that does not assume any other infrastructure but the mobile devices themselves.

Depending on what kind of dataflow dependency edge there is between two operators, the discovery mechanism works differently. In case of *fixed* dataflow edges, such as between the `Shop` node and the `User` node, once the `User` node is discovered, the same instance of the dataflow program expects always the same instance of a `User` node. This means concretely that when the connection is lost with the `User` node, the `Shop` node will wait until the connection is restored to resume the execution of the dataflow program. This makes sense if there is a stateful aspect over the different distributed nodes in the dataflow program, for example if users should receive personalized recommendations and not recommended products from different users. To detect a permanent disconnection the dataflow dependency edge can be annotated with a timeout period. Event messages are buffered until the timeout is signaled[2].

On the other hand, in some cases a different operator node encoding the same role can be used as a replacement, typically when there is no state associated with the operator's execution. This is catered for by *rebinding* dataflow edges, which are represented by a dashed line, such as the one between the `ProductDatabase` node and the `Shop` node. In this scenario, the shop may consist of different shelves which are all represented as `Shop` nodes. Concretely, a rebinding dataflow edge allows rebinding the dataflow dependency to another destination operator at runtime, because the network topology has changed for example. Again, a timeout can be specified that determines how long event messages are buffered. In this example, the user might have moved out of range of shelf in the shop and moved into communication range of a different shelf. How the group communication to the different shelves is handled is discussed later in Section 3.5.

3.3 Executing Mobile AmbientTalk Code

Dataflow operators operate on data streams. In most cases, it makes sense to process the stream on the device, through which the stream flows, to reduce communication overhead and to avoid a performance bottleneck when all the processing happens on a single device. In such scenarios, it is thus cheaper to move the processing code towards the data than the data stream itself. In the shopping assistant example, the scaling of album cover images can happen directly on the server hosting the images which is better suited for such a CPU-intensive job than a mobile device. Furthermore, in the face of intermittent network connections, long-lasting computations can continue while the network connections with other nodes is temporarily broken, and flush buffered

[2] Buffer overflows can happen in theory for very large timeout periods and will raise a Java exception.

return values when the network connection is restored. This is why the implementation of dataflow operators is in fact mobile AmbientTalk code that is sent to application components playing a role in the dataflow program. To be able to execute this mobile code, these application components need to provide some already present infrastructure in the form of some pre-implemented AmbientTalk or Java methods. Currently, this means that services playing a role in our coarse-grained dataflow model are usually implemented as objects that provide an interface that can be called by the mobile code (as shown in our example where the `filterOutPresentProducts` method is assumed to be implemented by the `Shop` node[3]). To allow objects to be easily extended with the necessary methods for accepting and executing their mobile dataflow operator code in the correct way, we provide a basic `OperatorHostInterface` object. Custom implementations specific to the device hosting the service can be used as well (e.g., to impose restrictions on the received mobile code to prevent security issues) by overriding specific methods on the `OperatorHostInterface`.

The code snippet below shows how a certain host can advertise itself as a `User` by simply exporting a service object implementing the `User` role (hence, this is local code present on the user's machine). The `userService` object extends from the `OperatorHostInterface` discussed above and simply implements the necessary methods to fulfill its role in the application. The last two lines declare a Java interface that will be used for service discovery and publish the user service into the network such that it can be discovered by other remote application components.

```
1  def userService := extend: OperatorHostInterface with: {
2    // ... Private fields...
3    def showGui() {
4      // Show the user interface
5    };
6    def showProductsInBag(products) {
7      // Update GUI with new shopping bag contents
8    };
9    def showRecommendations(products) {
10     // Update GUI with new recommendations
11   };
12   def showRecommendationsInShop(products) {
13     // Update GUI with new recommendations in stock
14   };
15 };
16 deftype User;
17 export: userService as: User;
```

Note that the same device hosting a number of application components can play a role in different dataflow programs. However, race conditions cannot occur because the communication between all dataflow operator nodes happens by

[3] One can designate a dedicated namespace that is visible to the mobile code and organize the library code to be called by the mobile code in Java packages.

means of asynchronous AmbientTalk messages that are scheduled in the event loop of each host, and are sequentially executed by a single thread (causing the sequential execution of the operator code as well).

3.4 Propagating Events and Reacting to Events

Until this point, we have not elaborated yet on how the actual dataflow program is executed by our dataflow engine. This is based on the reactive programming paradigm [22–24], which requires a reactive version of the AmbientTalk interpreter [25]. Concretely, the dataflow variables in the operator code denote reactive values. These reactive values are updated each time their respective input value changes (by new data objects flowing over the dataflow edges corresponding to the input values). Analogous to the reactive programming paradigm, a dataflow operator is re-executed as soon as one of the reactive values it depends on changes. Dataflow updates are signaled simply by executing a dataflow operator, which results in a new return value for the executed dataflow operator and which is propagated over all the outgoing dataflow edges. For example in the **`self`** role, the scanned products could be periodically updated by the RFID reader by for example periodically scanning its surroundings for tags and updating a reactive value. The events signaled by this reactive value will in this case be propagated along the `products` and `productsInBag` dataflow dependencies, invoking the rest of the dataflow graph. In the other direction, the `User` node will receive updates to its `productsInBag`, `recommendations` and `inStock` dataflow variables, which will result in the re-execution of the dataflow operator, leading to the necessary updates to the user interface.

The names identifying the dataflow dependency edges are again used for service discovery, but this time only for enabling the communication (initiated behind the scenes by the dataflow engine) to update the reactive values used in the dataflow operator code: no additional semantics are attached to them (in contrast to the role names). Important to note is that the propagation of dataflow events happens by means of the underlying reliable asynchronous messages of AmbientTalk. This means that intermittent connections between dataflow operators do not cause errors, instead the event messages are buffered and resent when the same dataflow operator host comes back in range or a replacement host is found. Per dataflow dependency, a timeout can be specified that determines how long these messages are buffered. In case of a timeout, an exception is raised on both disconnected application components, which can trigger cleanup actions in response.

3.5 Dependency Arities

In mobile ad hoc networks, it is in many cases necessary to gather information from and/or propagate information to a multitude of peers (that can move out of range and back into range at any point in time). For example, a mobile application may want to query other mobile applications about their GPS coordinates to show the location of their users on a map. On the other hand, an application

may also want to periodically broadcast the new GPS coordinates of its host device to nearby peers. To cater for this kind of remote communication with a number of equivalent peers, we have extended the original dataflow coordination model with *dependency arities*. These dependency arities can be one-to-one, one-to-many, many-to-one or many-to-many. This is depicted graphically in our visual dataflow language on the end points of edges (i.e., `1-1`, `1-*`, `*-1` and `*-*`, respectively).

	Incoming `1`	**Incoming `*`**
Outgoing `1`	Send one value to a single (rebound or fixed) node	Send one value to all reachable nodes of same role
Outgoing `*`	Send list of values to a single (rebound or fixed) node	Send list of values to all reachable nodes of same role

Table 1. Dependency arity semantics

In the example given above, there is a one-to-many dataflow dependency between the `ProductDatabase` and `Shop` nodes. This will cause the dataflow engine not only to look for a single `Shop` operator host (representing a shelf in our shop scenario), but to all hosts able to play this role in the dataflow program. They will all receive the mobile operator code and will all receive the events propagated along the `recommendToUser` edge. Now it is up to the dataflow programmer to decide what will happen with all the different return values from the replicated `Shop` nodes running in parallel. One could either choose to receive the events propagated by a single `Shop` (although other ones are running in parallel) by installing a one-to-one dependency between the `Shop` and the `User` node. In our scenario this means that the user only receives recommended products from a single shelf, although multiple ones are filtering recommended product lists based on their contents. The alternative would be to install a many-to-one dependency between these components. In that case, the `User` node will receive *all* the events of *all* the replicated `ProductDatabase` nodes that are in communication range. The result here is that the user receives recommended products from all shelves in communication range, i.e., a dynamically changing list of product lists. The programmer can specify a timeout to determine how long values received from non-responding nodes should be kept in the list. Hence, the list changes not only when dataflow values change, but also when values are added (because new nodes were discovered) or removed (when nodes time out). The processing code that operates on this reactive list should of course take into account that the dataflow variable represents such a changing list. Note that declaring a dataflow dependency one-to-many or many-to-many will automatically convert the respective dataflow edge into a rebinding edge (see Section 3.2). The reason is that when broadcasting events to all nodes playing the same role, a fixed dataflow dependency simply makes no sense: events are propagated to a dynamically changing collection of listeners as the network topology changes. This is not always desirable: the dataflow dependency between the `Shop` node and `User` node should clearly be fixed: one instance of the dataflow program

should send personalized recommendations always to the same user. Table 1 summarizes the semantics of the different combinations of dataflow nodes and dependency arities connecting them.

4 Limitations and Future Work

The naming and discovery of dataflow nodes happens via Java interfaces acting as role names. We make the underlying assumption that the name of such Java interfaces represents a unique service and is known by all participating services in the immediate neighbourhood. This discovery mechanism also does not take versioning into account explicitly. For example, if the `ProductDatabase` service from the example in Section 3.1 is updated, older clients may discover the updated service, and clients that want to use only the updated service may still discover older versions. Clients and services are thus themselves responsible to check versioning constraints. A similar issue can be observed with respect to security: currently, dataflow operator hosts are responsible themselves for providing a secure infrastructure to execute the mobile AmbientTalk code that is being sent to them by the dataflow engine. This infrastructure currently has to be implemented by the programmer of the service: there is no ready to use framework for this purpose.

Finally, we are working on a more advanced visual dataflow editor and debugger. The current editor[4] is a very early prototype which we would like to extend with debugging support (currently it only offers syntactic error checking and undefined variables checking of the mobile AmbientTalk scripts in dataflow operators). We are currently working on a prototype that allows the stepwise execution of the dataflow graph, similar to the way one would step through invocation stack frames in a stack-based language. This is not trivial since the execution of the dataflow program is distributed over a mobile ad hoc network, of which the nodes have to communicate with the debugger. The approach we are currently pursuing is to integrate a simulator with the debugger that simulates network communication and also allows simulating failures, for example arbitrary network partitions, during the simulated execution of the dataflow program.

5 Conclusions

In this paper, we have introduced a visual dataflow language for coordinating mobile ad hoc network applications. The motivation for using a dataflow language for coordination is that the language offers a coordination model that is very well suited to the dynamic and inherently parallel nature of mobile ad hoc network applications and allows separating the coarse-grained coordination behavior from the fine-grained application logic. The language represents data

[4] It can be downloaded as a library for the AmbientTalk language from: `http://code.google.com/p/ambienttalk/` ("rfid" SVN branch).

dependencies between distributed mobile application components very explicitly and allows them to be visualized and edited graphically. Since its coordination model is purely based on the satisfaction of these data dependencies, it maps very well on a mobile ad hoc network environment where distributed application components are running in parallel, react to events coming from the outside world, and are interconnected by peer-to-peer connections over which data can only flow when the connection is not broken (which may frequently happen due to the limited communication range and the mobility of the devices). Our implementation of a dataflow coordination language is made resilient to these intermittent connections by either buffering dataflow events between different nodes while communication partners are (temporarily) unavailable, or by allowing new reachable nodes to be dynamically selected. The resulting dataflow graph instance is entirely decentralized and does not rely on a fixed infrastructure, but instead only on peer-to-peer connections. To provide the programmer with abstractions to encode different communication strategies that we deem useful in such a mobile context, we have extended the basic dataflow model with *rebinding* data dependency edges and dependency *arities*.

References

1. Kaminsky, A., Bischof, H.P.: Many-to-many invocation: a new object oriented paradigm for ad hoc collaborative systems. In: 17th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications, New York, NY, USA, ACM (2002) 72–73
2. Meier, R., Cahill, V.: Steam: Event-based middleware for wireless ad hoc networks. In: 22nd International Conference on Distributed Computing Systems, Washington, DC, USA, IEEE Computer Society (2002) 639–644
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. Grimm, R.: One.world: Experiences with a pervasive computing architecture. IEEE Pervasive Computing **3**(3) (2004) 22–30
5. Chin, B., Millstein, T.: Responders: Language support for interactive applications. In: ECOOP, Nantes, France (July 2006)
6. Haller, P., Odersky, M.: Event-based programming without inversion of control. In: Proc. Joint Modular Languages Conference. Volume 4228 of Lecture Notes in Computer Science., Springer (2006) 4–22
7. Adya, A., Howell, J., Theimer, M., Bolosky, W.J., Douceur, J.R.: Cooperative task management without manual stack management. In: USENIX Annual Technical Conference, Berkeley, CA, USA, USENIX Association (2002) 289–302
8. Levis, P., Culler, D.: Mate: A tiny virtual machine for sensor networks. In: International Conference on Architectural Support for Programming Languages and Operating Systems, San Jose, CA, USA. (Oct. 2002)
9. Kasten, O., Römer, K.: Beyond event handlers: programming wireless sensors with attributed state machines. In: 4th international symposium on Information processing in sensor networks, Piscataway, NJ, USA, IEEE Press (2005) 7
10. Gelernter, D., Carriero, N.: Coordination languages and their significance. Commun. ACM **35**(2) (1992) 97–107

11. Mamei, M., Zambonelli, F.: Programming pervasive and mobile computing applications with the TOTA middleware. In: IEEE International Conference on Pervasive Computing and Communications, Washington, DC, USA, IEEE Computer Society (2004) 263
12. Arbab, F.: Reo: a channel-based coordination model for component composition. Mathematical. Structures in Comp. Sci. **14**(3) (2004) 329–366
13. Kalkman, C.: Labview: A software system for data acquisition, data analysis, and instrument control. Journal of Clinical Monitoring and Computing **11**(1) (1995) 51–58
14. Cox, P.T., Glaser, H., Lanaspre, B.: Distributed prograph: Extended abstract. In: International Workshop on Parallel Symbolic Languages and Systems, London, UK, Springer-Verlag (1996) 128–133
15. Gay, D., Levis, P., von Behren, R., Welsh, M., Brewer, E., Culler, D.: The nesc language: A holistic approach to networked embedded systems. In: ACM SIGPLAN conference on Programming language design and implementation, New York, NY, USA, ACM (2003) 1–11
16. Van Cutsem, T., Mostinckx, S., Gonzalez Boix, E., Dedecker, J., De Meuter, W.: Ambienttalk: object-oriented event-driven programming in mobile ad hoc networks. In: XXVI International Conference of the Chilean Computer Science Society, IEEE Computer Society (2007) 3–12
17. Van Cutsem, T., Mostinckx, S., De Meuter, W.: Linguistic symbiosis between event loop actors and threads. Computer Languages Systems & Structures **35**(1) (2008)
18. Miller, M., Tribble, E.D., Shapiro, J.: Concurrency among strangers: Programming in E as plan coordination. In: Symposium on Trustworthy Global Computing. Volume 3705 of LNCS., Springer (April 2005) 195–229
19. Johnston, W.M., Hanna, J.R.P., Millar, R.J.: Advances in dataflow programming languages. ACM Comput. Surv. **36**(1) (2004) 1–34
20. Park, K., Kim, Y., Chang, J., Rhee, D., Lee, J.: The prototype of the massive events streams service architecture and its application. In: 9th ACIS International Conference on Software Engineering, Artificial Intelligence, Networking, and Parallel/Distributed Computing, Washington, DC, USA, IEEE Computer Society (2008) 846–851
21. Jin, X., Lee, X., Kong, N., Yan, B.: Efficient complex event processing over rfid data stream. In: 7th IEEE/ACIS International Conference on Computer and Information Science, Washington, DC, USA, IEEE Computer Society (2008) 75–81
22. Elliott, C., Hudak, P.: Functional reactive animation. In: ACM SIGPLAN International Conference on Functional Programming. Volume 32(8). (1997) 263–273
23. Wan, Z., Taha, W., Hudak, P.: Real-time FRP. In: International Conference on Functional Programming (ICFP'01). (2001)
24. Peterson, J., Hudak, P., Elliott, C.: Lambda in motion: Controlling robots with haskell. In: 1st International Workshop on Practical Aspects of‘ Declarative Languages. (January 1999)
25. Mostinckx, S., Lombide Carreton, A., De Meuter, W.: Reactive context-aware programming. In: Workshop on Context-Aware Adaptation Mechanisms for Pervasive and Ubiquitous Services (CAMPUS 2008). Volume 10 of Electronic Communications of the EASST., DisCoTec (June 2008)