

Orchestrating Nomadic Mashups using Workflows

Eline Philips^{*}
ephilips@vub.ac.be

Andoni Lombide
Carreton[†]
alombide@vub.ac.be

Niels Joncheere
njonchee@vub.ac.be

Wolfgang De Meuter
wdmeuter@vub.ac.be

Viviane Jonckers
vejoncke@vub.ac.be

Software Languages Lab
Department of Computer Science
Vrije Universiteit Brussel, Belgium

ABSTRACT

Middleware for mashups is currently not able to compose the services residing in a nomadic network. Its transient connections and connection volatility result in a highly dynamic environment where services can appear and disappear at any point in time. The consequence is that these services must be discovered at runtime in an ad hoc fashion and must execute asynchronously to prevent a disconnected service to block the execution of an entire mashup. Orchestrating loosely coupled asynchronously executing services calls for a process-aware approach. This paper proposes the use of workflow patterns to enable a high level specification of the interactions between the mobile services constituting a nomadic mashup.

1. INTRODUCTION

Today's society is characterised by the ubiquity of mobile devices such as mobile phones, PDAs and handhelds. The omnipresence of wireless communication facilities, for instance WiFi, 3G and Bluetooth, enable us to connect these devices in a mobile ad hoc network (MANET). Nomadic networks fill the gap between traditional networks and mobile ad hoc networks as these nomadic environments consist of both a group of mobile devices and a fixed infrastructure. Mobile devices in a nomadic network can move around dynamically while trying to maintain a connection to the fixed infrastructure. In such networks the heavy computations are performed by the fixed infrastructure [6]. For these kind

of networks, an abundance of interesting applications can be supported as these networks are omnipresent (for instance in hospitals, airports, shopping malls, ...). However, the development of such applications is not straightforward as special properties of the communication with mobile devices have to be taken considered.

A first characteristic that has to be taken into account when composing a nomadic mashup is the connection volatility which is inherent to the environment. Nomadic mashups consist of mobile mashup sources which are residing in the physical world. While in conventional mashups the services constituting the mashup are known beforehand (e.g. by a fixed URL to a RSS feed) and are assumed to execute in a reliable network infrastructure (e.g. a Web 2.0 environment), the services constituting a nomadic mashup have to be dynamically discovered at runtime, since these services can be hosted by mobile devices that can connect and disconnect at any point in time. Furthermore, a mobile service that became unavailable should not block the execution of an entire mashup. Nomadic mashup sources can take many forms, ranging from small sensors on mobile devices up to complex services running on a fixed backbone infrastructure. As in mobile ad hoc networks the challenge is making the high heterogeneity of devices co-operate and deal with their transient and permanent failures.

Although there exists middleware [6] and programming languages, like AmbientTalk (explained in section 3.1), which are developed to meet the specific properties of dynamically changing environments, the composition between the different services is still programmed in an ad hoc way. In order to orchestrate the large heterogeneity of services we encounter, reusable composition patterns are needed to specify the interactions at a higher level. In classic networks, this orchestration can be achieved by using workflow languages which support a parallel execution, the composition of services and the description of control flow. However, these existing workflow languages are not really suited to describe mashups for nomadic networks, because they do not take their specific characteristics into account.

As AmbientTalk is specifically sculpted to deal with the phenomena in mobile networks, such as connection volatility, autonomy, natural concurrency and ambient resources [4], we have used this scripting language and its runtime to build workflow patterns on top of. Hence, enabling the specification of the control flow for mashups at a higher level and

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

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

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

Mashups'09, October 25, 2009, Orlando, FL, USA

Copyright 2009 ACM 978-1-60558-364-8/08/12 ...\$10.00.

incorporating the specific requirements of the environment.

The problem of working with mobile devices in a dynamically changing environment is general to all applications in both nomadic and mobile ad hoc networks. Our proposed solution focusses on nomadic mashups which are a concrete example of architectures that are affected by the phenomena of mobile networks.

This paper is organised as follows: first we describe some related work before exhibiting our approach on orchestrating services in a nomadic environment. In this section we first introduce the programming language AmbientTalk, as this language forms the foundation on top of which we are building our nomadic workflows solution. We use the workflow paradigm in order to divide processes into activities where control flows from one activity to another according to a number of patterns that support among others concurrency. Thereafter, the implementation of these workflows is discussed and exemplified. To conclude, some future work and general conclusions are presented.

2. RELATED WORK

Current mashup development tools target a reliable Web 2.0 environment. We can distinguish two approaches: one is using a graphical representation of the service composition, the most representative one is Yahoo! Pipes [10]. The execution engines of such tools assume a stable network infrastructure to allow the server running the engine to coordinate the different services (e.g. Yahoo! Pipes run on a dedicated Yahoo! server and offer no runtime service discovery but assume fixed URLs to reach services). These assumptions cannot be made in nomadic networks where some of the mashup components dynamically join and leave the network while the mashup is executing and disconnections are rather the rule than the exception.

The other approach is by expressing the composition of remote services in web scripting languages using AJAX. Expressing service compositions with AJAX happens using an event-driven paradigm based on callbacks. This causes more complicated event-driven applications to be very hard to understand, as we will show in section 3.1 for the AmbientTalk language, which uses a similar event-driven paradigm but targeted towards mobile ad hoc networks. There exist higher level coordination languages based on Javascript and AJAX, such as Ubiquity [1] and Orc [5]. Orc for instance uses a process calculus to express the coordination of different processes. However, these languages also assume a stable network interconnecting the services.

Other coordination languages are not specifically dedicated to mashup development, but still could be used for this purpose. Reo [2] 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 coordinators dictate the coordination of the simpler connectors, which eventually coordinate the software components that they interconnect. When software components are disconnected, one has to manually invoke the migration of a component to a different node, however the channels connecting the component are automatically rebound.

A workflow language that specifically targets mashups is

Bite [3], which focusses on web-based services. While Bite supports asynchronous messaging and different communication protocols, it operates on services that are known beforehand and assumes fixed URLs to reach those services. Although there exist workflow languages for dynamically changing environments, like mobile ad hoc networks and nomadic networks, they do not cope with all issues inherent to nomadic network applications. CiAN [12] is a workflow engine that was developed to work in a mobile ad hoc network. As a centralised orchestration engine can not be used in MANETs, CiAN decides the services to be invoked a priori. In a dynamic changing environment where services can come in and go out of reach at any possible time, it is not possible to know all services beforehand. An other workflow language, Workpad [7], is developed with nomadic networks in mind. This language also lacks support for disconnections that are inherent to the mobile part a nomadic network consists of. Mobile Business Processes [9] is a workflow engine by Nokia which offers a repository of existing services that can be used to define business processes. Although these workflows are run on mobile devices, they only interface with fixed web services that are known beforehand (by means of a URL).

3. NOMADIC WORKFLOWS

In stable networks, workflows are used to model and orchestrate complex applications. The workflow architecture is typically centralised and the interactions between the different services are synchronous. There also exist distributed engines for workflows and more recently mobile ad hoc networks and nomadic networks are also targeted by the workflow community. However, these workflow languages have almost no support for handling the high volatility of these kind of networks.

AmbientTalk is a programming language which treats disconnections at the very heart of its computational model. Moreover, this language supports dynamic service discovery which is opportune for nomadic networks. Although this language is suited for writing applications for mobile ad hoc and nomadic networks, the orchestration of these applications is still programmed in an ad hoc manner. Complex applications or mashups that consist of asynchronously executing distributed services become hard to develop, understand and reuse. In this section, we first briefly explain AmbientTalk and how it offers support for scripting together mobile services. Subsequently, we introduce our workflow abstractions that we built on top of this language.

3.1 AmbientTalk

In this section, we briefly explain the programming language support that we assume to build our workflow language targeting mashups in nomadic networks. The *ambient-oriented* programming paradigm [4] is specifically aimed at such applications. For this reason we chose to build our workflow language on top of an ambient-oriented programming language. Ambient-oriented programming languages should explicitly incorporate potential network failures in the very heart of their computational model. Therefore, communication between distributed application components should happen without blocking the execution thread of the different components such that devices may continue doing useful work even when the connection with a communication partner is lost.

Ambient-oriented languages also deal with the dynamically changing network topology in nomadic and mobile ad hoc networks. The fact that in such networks devices spontaneously join with and disjoin from the networks means that the services these devices host cannot be discovered using a fixed, always available name server, but instead require dynamic service discovery protocols (e.g. broadcasting advertisements to discover nearby services).

Both the runtime discovery of and the non-blocking communication between distributed application components in nomadic and mobile ad hoc networks give rise to an event-driven architecture, where there is a natural form of concurrency among the distributed application components. Such architectures can greatly benefit from process-aware technologies such as workflows to allow a separate and higher level orchestration of the concurrent processes in the mashup.

AmbientTalk [14, 13] is a distributed programming language embedded in Java¹. The language is designed as a distributed scripting language that can be used to compose Java components which are distributed across a nomadic or even 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 either a low-level socket API or a high-level RPC API (i.e. Java RMI) to enable distributed computing. In contrast, AmbientTalk is a fully event-driven programming language. It provides only event loop concurrency [8] and distributed objects communicate by means of asynchronous message passing. 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.

AmbientTalk offers direct support for the different characteristics of the ambient-oriented programming paradigm described above.

1. In an ad hoc network, objects must be able to discover one another without any infrastructure (such as a shared naming registry). Therefore, AmbientTalk has a service discovery engine that allows objects to discover one another in a peer-to-peer manner. Java interfaces act as the common pieces of information by means of which objects are advertised and discovered (publish/subscribe service discovery engine).
2. In an ad hoc network, objects may frequently disconnect and reconnect because of network partitions. Therefore, AmbientTalk provides fault-tolerant asynchronous message passing between objects: if a message is sent to a disconnected object, the message is buffered and resent later, when the object becomes reconnected. Other advantages of asynchronous message passing over standard RPC is that the asynchrony hides latency and that it keeps the application respon-

sive (i.e. the event loop is not blocked during remote communication and is free to process other events).

3.2 Distributed Programming in AmbientTalk

AmbientTalk uses a classic event-handling style by relying on blocks of code that are triggered by event handlers. 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. 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`.

```
// when a service classified as LocationService is discovered,
// this object will be accessible via locationSvc
when: LocationService discovered: { |locationSvc|
  // send asynchronous message getLocation to discovered object
  when: locationSvc<-getLocation(gpsModule.getCoordinates())
    becomes: { |myLocation|
      // when reply is received,
      // discover an object of type WeatherService
      when: WeatherService discovered: { |weatherSvc|
        // send asynchronous message to the discovered WeatherService
        when: weatherSvc<-getWeather(myLocation)
          becomes: { |weatherInfo|
            // update weather information in
            // the user interface
          }
        }
      }
    }
}
```

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 location 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 his 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 signalled 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`, and as soon the reply to this query is received update the user interface.

As can be seen from the above example, service discovery and replies of remote queries 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, but more complex synchronization and coordination patterns require more complicated structures (e.g. the lookup of a

¹The language is available at prog.vub.ac.be/amop

WeatherService could happen in parallel without waiting for the LocationService to reply). While in this simple example the control flow remains apparent enough to understand, the control flow of large-scale event-driven applications can quickly become puzzling. In the following sections we discuss how to add a process-aware layer of abstraction on top of AmbientTalk (which uses messages/events as the level of abstraction) such that the asynchronously executing processes can be orchestrated by means of workflows.

3.3 Workflow Patterns in AmbientTalk

This section describes the implementation of some workflow patterns on top of AmbientTalk². Consider the example that was given in section 3.1 where a user interface is updated with the current weather at the user's location. This example could be expressed by a sequence pattern as depicted in figure 1. The circle at the end of the pattern denotes a stop node, whereas the rectangles represent the used services.

The description of the workflow resides on the backbone of the nomadic system, whereas the different services are either located on fixed devices or on mobile devices that move around through the environment. By situating the workflow description on the fixed infrastructure, we ensure that the workflow itself cannot disconnect and become unavailable during its execution.

First, we explain how these services are implemented in AmbientTalk and thereafter we describe how these workflow patterns, with an emphasis on the sequence pattern that is used in the example, are implemented and can be used as an abstraction layer for describing the control flow. Afterwards we describe how more complex workflows can be expressed by combining several workflow patterns.



Figure 1: Workflow representation for a mashup consisting of three services.

In AmbientTalk, services are implemented as distributed objects that advertise themselves by means of a service *type tag*. Currently, services have a fixed interface and must implement a *start* method which performs the actual execution of the service. This *start* method has one argument which allows passing data between the different services. The code snippet below illustrates the implementation of the WeatherService in AmbientTalk. Important to note is that this is code running on the service host. The mashups making use of this service are oblivious to its implementation, they should only match on the service type tag under which the service is advertised.

```

deftype WeatherService;

def service := object: {
  def start(args) {
    // Check if args is a location, if not throw an error.
    // Otherwise, determine the weather
    // at this location and return it.
  };
};
export: service as: WeatherService;

```

²The implementation is available at <http://code.google.com/p/ambienttalk/>

In order to retrieve the forecast information of the user's location, we need to compose the two services, LocationService and WeatherService, by making use of a sequence pattern. This sequence pattern must first send the asynchronous *start* message to the first service, and afterwards invoke the WeatherService by passing the result of the invocation of the LocationService to it. The result of the WeatherService invocation is afterwards passed to the GUI service where also the *start* method is invoked. The reply of this invocation is then passed to the stop pattern, which ends the workflow. Hence, the implementation of this small example uses two control flow patterns, namely sequence and stop. These workflow patterns are implemented as AmbientTalk objects which are tagged. These tags are used to distinguish between normal patterns, patterns that signal multiple replies (such as a *simple merge*) and service type tags (that denote yet to be discovered services). In order to enable a transparent nesting of patterns and services, we need to make sure that these interfaces match. Therefore, workflow patterns also implement a *start* method.

The code below presents the implementation of a sequence pattern. This pattern is initialised with a table of components *componentsTable*. When invoking its *start* method, all components of that table (service type tags or workflow patterns) are invoked sequentially and the last component of the sequence is returned. Remember that services can be executed on a mobile device residing in the nomadic environment, and hence the connections found here can be transient. The workflow patterns keep this transient and permanent failures into account by incorporating timeouts, after which a new service with the same type tag is looked up.

```

def Sequence := object: {
  def componentsTable;

  // Constructor of the Sequence pattern.
  def init(table) {
    componentsTable := table;
  };

  def start(args) {
    // Creates a notification object on which when-callbacks
    // can be registered.
    def result := makeFuture();

    def execute(idx, args) {
      def component := componentsTable[idx];
      if: (is: component taggedAs: Service) then: {
        // Send service the asynchronous start message.
        // After service signalled a reply, check if the
        // sequence has ended.
        // If so, return the notifier object that will
        // signal the reply event from the service
        // invocation, such that other patterns can be
        // notified when the sequence is done executing.
        // If not, call execute with an increased index (idx).
      } else: {
        // Invoke the start message of the pattern.
      }
    }
  }
} taggedAs: [Pattern];

```

Important to note is that this sequential execution can be achieved by explicitly waiting for the output of a service before starting the following one, without having to manually synchronize asynchronously executing process by nesting callbacks in the correct way. After the workflow pattern's execution has finished, the stop pattern will use this last component to listen for its reply and eventually

end the workflow when this reply (which is asynchronously computed) is received.

As can be seen in the abstract implementation (the full implementation is out of the scope of this paper) of the `execute` method, a test is performed to check the type of the sequence's current component. By hiding the different implementations of components (services or patterns) in the patterns itself, we allow composition of nested workflow patterns at an abstract level. Furthermore, the introduction of workflow patterns on top of AmbientTalk makes the orchestration of services explicit whereas the control flow in the language AmbientTalk itself is programmed in an ad hoc manner. By adding this extra layer of abstraction we enable composable, reusable services.

The stop pattern and its rationale is explained later on. The code below shows the implementation of our small example. The last line of the code fragment invokes the execution of workflow.

```
sequence := Sequence.new([LocationService, WeatherService, GUI]);

stop := Stop.new(sequence);
stop.start();
```

3.3.1 Combining Workflow Patterns

We introduce a more advanced mashup example which combines several workflow patterns and can be used to illustrate the composition of patterns. Consider an airport where passengers benefit from a rich airport infrastructure. For instance, passengers get a reminder on their PDA five minutes before the boarding time of their flight. This can be achieved by combining several services, namely a `ETicketService`, `DigitalClock`, `BoardingService` and `ReminderService`. This mashup both uses services residing on the backbone infrastructure and the mobile parts of a nomadic network. The `ETicketService` is a service on the passenger's mobile phone which contains all the information of the electronic ticket he/she bought. The services are part of the fixed infrastructure of the nomadic network and respectively are able to retrieve the current time, information of the boarding time of a flight and send reminders to certain passengers. Figure 2 depicts the workflow representation for this mashup. This workflow uses both a *parallel split* and *synchronize* pattern to describe the orchestration of the different services.

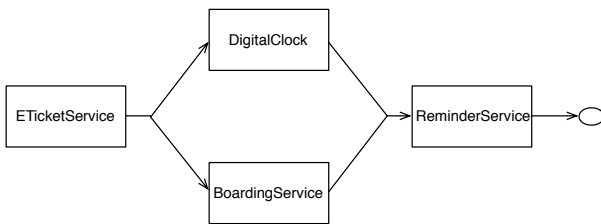


Figure 2: Workflow representation for a mashup using a parallel split and synchronize.

A parallel split diverges a single branch into two or more concurrently executing branches. As can be seen in the figure above, the `DigitalClock` and `BoardingService` are two parallel branches that can be executed independently of each other. Note that the synchronization of the two processes executing in parallel cannot be expressed by nesting callbacks, such as in the example in section 3.2. Although it is possible to

express such a synchronization in AmbientTalk, it requires the extensive use of the reflective and metaprogramming facilities of the language and leads to very complicated and difficult to reuse code. The implementation of this pattern has as input a tag which can be either an intentional description of a service or a pattern. The output of the parallel split is a list of components that can contain intentional descriptions of services by means of a type tag or objects that implement a `start` method, for instance a workflow pattern. A synchronization pattern converges several branches that have all succeeded into one subsequent branch. Concretely, when both the `DigitalClock` and `BoardingService` have terminated, the `ReminderService` is activated. The input of this pattern is a list of components (possibly the output of a parallel split pattern) and has as output a type tag that denotes a service or a pattern. The implementation of our mashup example is given by the following code snippet.

```
def parallelSplit :=
  ParallelSplit.new( ETicketService,
                    [ DigitalClock, BoardingService ] );

def synchronization :=
  Synchronization.new( parallelSplit, ReminderService );

def stop := Stop.new( synchronization );

stop.start();
```

Note that a `stop` pattern is also necessary in order to complete this workflow. The `typeTag` variable of the pattern can be instantiated with an intentional description of a service (for instance a tag `DigitalClock`) or a workflow pattern. The `start` method of the stop pattern needs to start this service or pattern. In case of a pattern, the output of this pattern can be either a type tag of a service or a list (when the pattern was for instance a parallel split). This output also needs to be started in order to have a correct termination of the workflow.

```
def Stop := object: {
  def typeTag;

  // Constructor of the Stop pattern.
  def init(tag) {
    typeTag := tag;
  };

  def start() {
    if: (is: typeTag taggedAs: Service) then: {
      // Start the service.
    } else: {
      // Start the component and after it's completed,
      // invoke the start method of the output (reply).
      when: serviceTag.start() becomes: { |reply|
        if: (is: reply taggedAs: Table) then: {
          // Start each component of the list.
          reply.each: { |cmp| cmp.start(); };
        } else: {
          // Start the service.
        };
      };
    };
  };
} taggedAs: [Pattern];
```

3.4 Discussion

The workflow patterns discussed in this section are just a small selection of the workflow patterns that we have adapted to the characteristics of nomadic networks and mashups running on top of them. Although we only have presented

toy examples of mashups, by making use of this selection of workflow patterns, we have shown that:

- Services constituting the mashup can be hosted on mobile devices and are discovered at runtime in a peer-to-peer manner based on an intentional description.
- Communication among the different services in a mashup happens without blocking other concurrently running services, even if some of them move out of range. This allows services to remain responsive and perform other meaningful tasks when interacting with components of different mashups.
- Communication between services is fault tolerant. The underlying runtime system guarantees message delivery by buffering messages that were not received by the destination service and attempting to resend them when the destination service becomes available again.

These implemented properties allow us to use standard reusable workflow patterns to describe the coordination between concurrently running distributed application components in nomadic networks without having to manually coordinate the interactions among these components using explicit callbacks.

We have deployed AmbientTalk applications on a number of mobile devices such as the HTC Touch Cruise and are currently in progress of deploying them on newer platforms such as Google Android and the Apple iPhone. Since our workflow abstractions simply assume an AmbientTalk virtual machine, they are directly usable on these platforms as well.

4. CONCLUSION

Complex distributed applications (such as mashups) running in nomadic networks have to be conceived as concurrently running activities to allow the different application components to remain responsive and keep doing useful work in the face of the frequent network partitions inherent to these kinds of networks. The orchestration of these concurrent activities into meaningful applications currently happens in an ad hoc way, usually by means of a callback-based paradigm. Workflows and workflow patterns provide an additional layer of abstraction such that interaction patterns among application components can be specified on a higher level and be reused because of their loose coupling with the fine-grained application logic. Unfortunately, current workflow systems do not meet all the requirements for the kinds of applications that we envision in nomadic networks. In this paper, we have presented the implementation of workflow patterns on top of a runtime system that does allow the orchestration of distributed services in a nomadic network, thanks to both a peer-to-peer and dynamic service discovery mechanism and communication primitives resilient to the volatile connections inherent to such networks. Now that a number of workflow patterns are implemented, we have hinted at some future work, most notably the introduction of new workflow patterns specifically designed for nomadic networks and a graphical workflow language that allows to chain together these workflow patterns together graphically and providing a visual view on the orchestration of the different application components.

5. FUTURE WORK

At this moment, the research of workflow patterns for nomadic networks is in its earliest phase of development, hence we were already able to define some issues that have to be handled in the near future.

Firstly, the current implementation of workflow patterns is restricted to the control flow patterns defined by van der Aalst [11]. We would like to extend them by also supporting data flow patterns and enable more complex ways of passing data between services. By enabling this data flow, we would be able to express more advanced mashups.

Currently, the naming and discovery of services happens via Java interfaces (wrapped in AmbientTalk type tags). Although this already allows describing services intentionally (by means of simple type tags), the assumption is made that these type tags represent a unique service and that it is known by all mashup participants. The discovery mechanism for instance does not take versioning into account. For example, if the `WeatherService` from the example in section 3.3 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 for checking versioning constraints.

A shortcoming of today's status is that the services have a restricted interface. At the moment, services are implemented as distributed objects which implement a start method. To allow more flexible service compositions, we are working towards patterns where services can have their own interface. Currently, calls to such an interface have to be wrapped in the single `start` method (e.g. an AmbientTalk object delegating calls to Java components in its `start` method).

Additionally, we would like to come up with some more advanced patterns that cope with some specific properties of the dynamic changing environment. Van der Aalst [11] describes a synchronisation pattern which succeeds when all branches have succeeded. In a dynamically changing environment, like nomadic and mobile ad hoc networks, 100% synchronization will not always be possible. Although van der Aalst presents some synchronisation patterns (like static partial join for multiple instances), these patterns are not sufficient. For instance, we would like to let synchronisation succeed when a selection of the results are available (after a certain percentage of answers is retrieved, after a certain period of time, at a predefined timeout value...).

Furthermore, as disconnections are inherent to nomadic networks it seems appropriate to build in support for compensating actions. As disconnections are the rule rather than the exception, we want to be able to specify for instance a timeout whenever a certain service is no longer available. These compensating actions are tightly coupled to the relaxed synchronisation that can succeed when not all branches of a workflow were realised.

Finally, we would like to extend our framework by building a graphical interface on top of our implementation. Most workflow languages have such an interface which facilitates the usage of the workflow patterns. By introducing a graphical interface we provide a graphical representation of the mashup which will make it easier to express the orchestration of services in a dynamically changing environment.

Acknowledgements

The research presented in this paper is partly funded by Alcatel-Lucent Bell and the Institute for the Promotion

of Innovation through Science and Technology in Flanders through the DIY-SE project.

6. REFERENCES

- [1] Ubiquity, 2005-2009.
<http://labs.mozilla.com/blog/2008/08/introducing-ubiquity/>.
- [2] Farhad Arbab. Reo: a channel-based coordination model for component composition. *Mathematical Structures in Comp. Sci.*, 14(3):329–366, 2004.
- [3] Francisco Curbera, Matthew Duftler, Rania Khalaf, and Douglas Lovell. Bite: Workflow composition for the web. In *ICSOC '07: Proceedings of the 5th international conference on Service-Oriented Computing*, pages 94–106, Berlin, Heidelberg, 2007. Springer-Verlag.
- [4] Jessie Dedecker, Tom Van Cutsem, Stijn Mostinckx, Theo D'Hondt, and Wolfgang De Meuter. Ambient-Oriented Programming. In *OOPSLA '05: Companion of the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*. ACM Press, 2005.
- [5] David Kitchin, Adrian Quark, William Cook, and Jayadev Misra. The orc programming language. In *FMOODS '09/FORTE '09: Proceedings of the Joint 11th IFIP WG 6.1 International Conference FMOODS '09 and 29th IFIP WG 6.1 International Conference FORTE '09 on Formal Techniques for Distributed Systems*, pages 1–25, Berlin, Heidelberg, 2009. Springer-Verlag.
- [6] Cecilia Mascolo, Licia Capra, and Wolfgang Emmerich. Mobile computing middleware. In *In Advanced lectures on networking*, pages 20–58. Springer-Verlag, 2002.
- [7] Massimo Mecella, Michele Angelaccio, Alenka Krek, Tiziana Catarci, Berta Buttarazzi, and Schahram Dustdar. Workpad: an adaptive peer-to-peer software infrastructure for supporting collaborative work of human operators in emergency/disaster scenarios. In *CTS '06: Proceedings of the International Symposium on Collaborative Technologies and Systems*, pages 173–180, Washington, DC, USA, 2006. IEEE Computer Society.
- [8] Mark Miller, Eric Dean Tribble, and Jonathan Shapiro. Concurrency among strangers: Programming in e as plan coordination. In R. De Nicola and D. Sangiorgi, editors, *Symposium on Trustworthy Global Computing*, volume 3705 of *Lecture Notes in Computer Science*, pages 195–229. Springer, April 2005.
- [9] Lasse Pajunen and Suresh Chande. Developing workflow engine for mobile devices. In *EDOC '07: Proceedings of the 11th IEEE International Enterprise Distributed Object Computing Conference*, page 279, Washington, DC, USA, 2007. IEEE Computer Society.
- [10] Mark Pruett. *Yahoo! pipes*. O'Reilly, 2007.
- [11] Nick Russell, Arthur H.M. ter Hofstede, Wil M. P. van der Aalst, and Natalya Mulyar. Workflow control-flow patterns: A revised view. Technical report, BPMcenter.org, 2006.
- [12] Rohan Sen, Gruia-Catalin Roman, and Christopher D. Gill. Cian: A workflow engine for manets. In *COORDINATION*, pages 280–295, 2008.
- [13] Tom Van Cutsem, Stijn Mostinckx, and Wolfgang De Meuter. Linguistic symbiosis between event loop actors and threads. *Computer Languages Systems & Structures*, 35(1), 2008.
- [14] Tom Van Cutsem, Stijn Mostinckx, Elisa Gonzalez Boix, Jessie Dedecker, and Wolfgang De Meuter. Ambienttalk: object-oriented event-driven programming in mobile ad hoc networks. In *Proceedings of the XXVI International Conference of the Chilean Computer Science Society (SCCC 2007)*, pages 3–12. IEEE Computer Society, 2007.