Science of Computer Programming 78 (2013) 168–194

Contents lists available at SciVerse ScienceDirect

Science of Computer Programming

journal homepage: www.elsevier.com/locate/scico

NOW: Orchestrating services in a nomadic network using a dedicated workflow language

E. Philips *, R. Van Der Straeten, V. Jonckers

Software Languages Lab, Vrije Universiteit Brussel, Brussels, Belgium

ARTICLE INFO

Article history:
Received 29 November 2010
Received in revised form 2 October 2011
Accepted 7 October 2011
Available online 11 November 2011

Keywords:
Workflow language
Nomadic network
Service orchestration
AmbientTalk

ABSTRACT

Orchestrating services in nomadic or mobile ad hoc networks is not without a challenge, since these environments are built upon volatile connections. Services residing on mobile devices are exposed to (temporary) network failures, which must be considered the rule rather than the exception. This paper proposes a dedicated workflow language built on top of an ambient-oriented programming language that supports dynamic service discovery and communication primitives resilient to network failures. The proposed workflow language, NOW, has support for high level workflow abstractions for control flow, rich network and service failure detection, and failure handling through compensating actions, and dynamic data flow between the services in the environment. By adding this extra layer of abstraction, the application programmer is offered a flexible way to develop applications for nomadic networks.

1. Introduction

We are surrounded by all kinds of wireless communication facilities that enable mobile devices to be connected in a mobile ad hoc network. Nomadic networks[1] fill the gap between traditional and mobile ad hoc networks since these nomadic environments consist of both a group of mobile devices and some fixed infrastructure [1]. As these kinds of networks are omnipresent (for instance, in shopping malls, airports, …), an abundance of interesting applications can be supported. However, the development of such applications is not straightforward as special properties of the communication with mobile devices, such as connection volatility, have to be considered. These complex distributed applications can be developed using technologies such as service-oriented computing. The composition of services can be achieved using the principles of workflow languages. In stable networks, workflow languages are used to model and orchestrate complex applications. The workflow engine is typically centralized and the interactions between the different services are synchronous. Although workflow languages such as WS-BPEL [2] are suited to orchestrate (web)services, they are not suited for nomadic networks where services are not necessarily known a priori. Additionally, services in a nomadic network must be dynamically discovered at runtime, since they can be hosted by mobile devices. Moreover, services that become unavailable should not block the execution of an entire workflow. Distributed engines for workflows exist and more recently mobile ad hoc networks [3,4] and nomadic networks [5] have also been targeted by the workflow community. However, these workflow languages have almost no support for handling the high volatility of these kinds of networks. For instance, there is no support for the reconnections of services, something which happens frequently.

In order to overcome these issues, we developed a workflow language NOW [6]. This nomadic workflow language is implemented on top of AmbientTalk [7,8], a distributed object-oriented programming language for mobile ad hoc networks. The patterns for control flow and failure handling NOW provides are introduced as a library for AmbientTalk.

* Corresponding author.
E-mail addresses: ephilips@vub.ac.be (E. Philips), rvdstrae@vub.ac.be (R. Van Der Straeten), vejoncke@soft.vub.ac.be (V. Jonckers).

[1] Not to be confused with the term *nomadic computing*.

0167-6423/$ – see front matter © 2011 Elsevier B.V. All rights reserved.
doi:10.1016/j.scico.2011.10.012

Our earlier paper [6] introduced the nomadic workflow language NOW and outlined some features of this language. This paper extends our previous work by providing implementation details related to control flow (Section 4.3), dynamic data flow mechanism (Section 5), and failure detection and compensation (Section 6). Moreover, this paper presents a validation (Section 7) of our work by conducting two types of experiments. The first experiment compares the implementations of example applications written in AmbientTalk and NOW. We also show the results of experiments that compare the performance of a direct-style implementation in AmbientTalk and an implementation using the patterns introduced by our workflow language.

This paper is organized as follows: we first introduce in Section 2 a motivating example in which we specify the requirements a nomadic workflow language must fulfil. In Section 3 we present AmbientTalk, an ambient-oriented programming language upon which we build our language. The concepts of control flow are presented in Section 4, together with their implementation description. Subsequently we describe a dynamic data passing mechanism in Section 5 and compensating actions in Section 6 before describing a validation of our work in Section 7. Finally, we discuss related work in Section 8 and conclude with our contributions and future work in Section 9.

2. Motivation

In this section we describe an example scenario which emphasizes requirements that must be supported by a nomadic workflow language. We also introduce a graphical representation depicting the workflow description of this application and afterwards highlight the requirements our workflow language must fulfil.

2.1. Example

Peter lives in Brussels and wants to spend his holidays in New York city. His plane leaves Brussels International Airport at 13:50 and he makes a transit at the airport of Frankfurt. Ten minutes before boarding he has not yet entered the boarding area. At the airport, Peter is announced as a missing passenger and a flight assistant is informed to start looking for him. When the flight assistant does not receive this message or does not respond within a certain time period, he/she is reminded again to look for Peter. Peter also receives a reminder on his smart phone. After ten minutes, the personnel responsible for boarding closes the gates and informs Aviapartner (the company that takes care of the luggage) to remove Peter's suitcase from the plane. When Aviapartner cannot be contacted because of some technical difficulties, the airport of Frankfurt is informed that Peter's luggage must be removed before the plane takes off to New York. Brussels' airport also ensures that the Frankfurt airport is notified of the free seat, so a last minute offer from Frankfurt to Newark becomes available. Peter gets notified that he can return home and catch another flight later.

This example introduces some of the concepts of a nomadic system. First of all, we can identify different kinds of participants in this scenario: mobile devices, mobile services and stationary services. Mobile devices represent the communication with visitors and passengers in the airport building, whereas mobile services are part of the infrastructure of the airport (for instance, flight assistants, guides). The stationary services (such as a departure screen or check-in desk) are also part of the infrastructure, but use a more reliable connection.

Fig. 1 gives a representation of this example using a notation loosely based on the BPMN [9]. This diagram and the other NOW diagrams in the remainder of the paper are used to help the reader follow the examples we present.

The circles in the picture represent *events*: in the beginning of the workflow there is a start event and at the end there is a termination event. The figure also contains several diamonds that depict *gateways* that can merge or split the control flow. There are two different kinds of gateways used in this workflow description: the diamond with a + sign represents a logical *and*, whereas the empty one symbolizes an *xor*. Rounded rectangles represent *activities*, which are a type of work that can be performed by either mobile (dashed line) or stationary services (solid line). A dashed rectangle that wraps an activity or a subworkflow represents a failure pattern, which specifies compensating actions for certain failure types.

As described in our scenario, compensating actions are used when a failure occurs in (or with) a service residing on a mobile device. For instance, when the flight assistant does not respond on time, he/she is reminded again to look for the missing passenger. As network failures are to be considered the rule rather than the exception in nomadic networks, mobile activities have default compensating actions for timeout and disconnection errors. In case of a timeout, the activity is restarted, and in case of a service disconnection, a service of the same type is rediscovered. However, sometimes more precise compensations need to be specified (when Aviapartner cannot be contacted, the airport of Frankfurt must be informed). The dashed rectangle surrounding the mobile activity is used for specifying these compensating actions, which can override and/or extend the default compensating actions of NOW.

The different control flow patterns [10] needed to describe a scenario as a workflow are:

1. *parallel split*: "The divergence of a branch into two or more parallel branches each of which executes concurrently [10]".
2. *exclusive choice*: "The divergence of a branch into two or more branches such that when the incoming branch is enabled, the thread of control is immediately passed to precisely one of the outgoing branches based on a mechanism that can select one of the outgoing branches [10]".
3. *structured discriminator*: "The convergence of two or more branches into a single subsequent branch following a corresponding divergence earlier in the process model such that the thread of control is passed to the subsequent branch

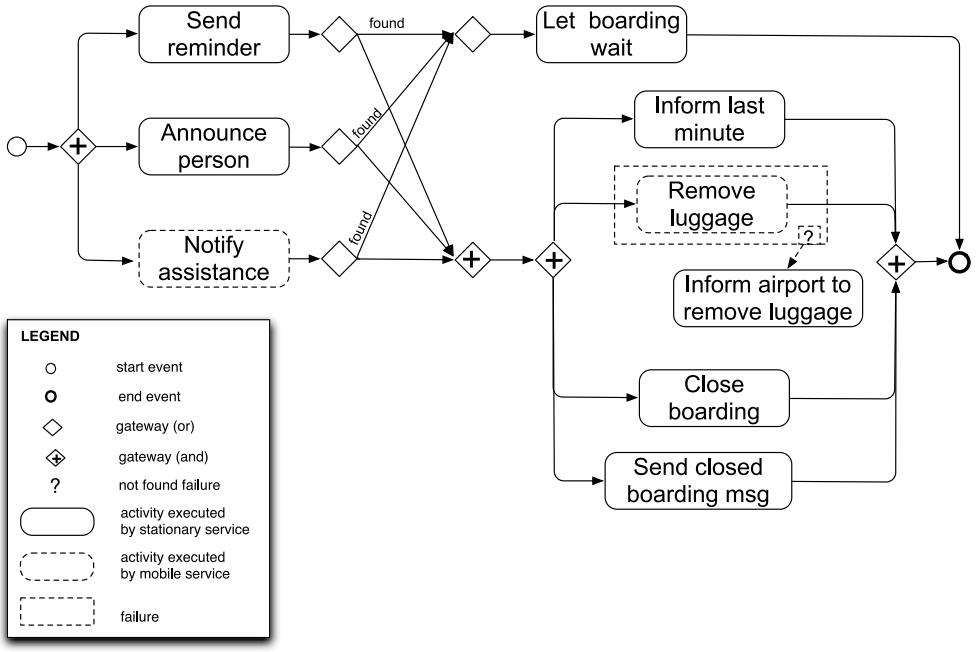


Fig. 1. Workflow description of the motivating example.

when the first incoming branch has been enabled. Subsequent enablements of incoming branches do not result in the thread of control being passed on [10]".

4. *synchronize*: "The convergence of two or more branches into a single subsequent branch such that the thread of control is passed to the subsequent branch when all input branches have been enabled [10]".

2.2. Requirements

As this motivating example shows, a workflow language for nomadic networks must fulfil the following requirements:

1. support two kinds of services: stationary and mobile.
2. support basic control flow patterns.
3. support data flow such that information can be passed between services.
4. automatic handling of failures in communication with mobile services (for example, transparently rediscovering a service of the same type).
5. ability to specify more precise compensating actions for specific kinds of failures by overriding and/or extending the default behaviour.

It is important to note that the description and execution of the workflow reside 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 and thus can serve as a reliable central node for orchestration and failure handling. This is the most important benefit of a nomadic network compared to a mobile ad hoc network.

3. AmbientTalk

The workflow language NOW we developed is built on top of the ambient-oriented programming language AmbientTalk [7,8]. In this section we first briefly present AmbientTalk and some of its language features necessary to comprehend the remainder of the paper. Afterwards we motivate why another language is needed for orchestration in nomadic networks.

AmbientTalk is an object-oriented distributed programming language specifically aimed at mobile ad hoc networks. The programming language is implemented as an interpreter on top of Java. AmbientTalk is developed on top of the J2ME platform and runs on mobile devices.[2] Although the language is implemented on top of Java, these languages differ in the way they deal with concurrency and network programming. Java is a multithreaded language which has a low-level socket API and a high-level RPC API (Java RMI). In contrast, AmbientTalk is a fully event-driven programming language which provides both event-loop concurrency and distributed object communication (which is explained later in Section 3.1).

2 The AmbientTalk interpreter is available as an application on the Android market.

AmbientTalk offers direct support for the different characteristics of the ambient-oriented programming paradigm [11]:

- In a mobile network (mobile ad hoc or nomadic), objects must be able to discover one another without any infrastructure (such as a shared naming registry). Therefore, AmbientTalk has a *discovery engine* that allows objects to discover one another in a peer-to-peer manner.
- In a mobile network (mobile ad hoc or nomadic), objects may frequently disconnect and reconnect. Therefore, AmbientTalk provides *fault-tolerant asynchronous message passing* between objects: if a message is sent to a disconnected object, the message is buffered so it can be resent when the object reconnects.

3.1. Distributed programming in AmbientTalk

AmbientTalk employs an actor-based concurrency model [12]. In order to allow communication among multiple devices, actors must be able to discover one another. Furthermore, actors need to be resilient to intermittent disconnections of the actors they are communicating with.

Actors are containers of objects. In an actor-based language programs are written exclusively in terms of objects. Hence, actors must export their objects in order to make them available to remote actors. This allows objects on different devices to address each other directly. Each object is exported with a certain *type tag* which can be used for discovering a certain service. Listing 1 shows how to create an AmbientTalk object implementing a service.

Listing 1. Implementation of a service in AmbientTalk.

```
deftype Aviapartner;
def service := object: {
    def companyName := nil;
    ...  /*other fields */
    def init(cn) { self.companyName := cn; /* assignment of other fields */ };
    def removeLuggage(flight, name, surname, id) {
        /* Remove luggage from flight,
           and return the location where it's  stored */ };
    ... /* other methods */
};
export: service as: Aviapartner;
```

This example shows the implementation of an object in AmbientTalk which is bound to the variable `service`. This object implements a service responsible for the luggage transportation at an airport. The object has some fields, a constructor (always called `init`) and several methods amongst which is a method `removeLuggage`. This object is exported with the *type tag* `Aviapartner`, meaning that from now on this object can be discovered by remote actors and their objects using this tag.

As we already mentioned, AmbientTalk differs from Java in the way the language deals with concurrency. AmbientTalk is an event-driven programming language which provides event-loop concurrency and distributed object communication. The latter is achieved by means of *asynchronous message* passing (expressed as `obj<-msg()`).

```
def location := luggageS<-rmvLuggage("BA472", "Peter", "Walker", "BA54282");
```

In this small example, the asynchronous message `rmvLuggage` is sent to the discovered Aviapartner service (this service is bound to the variable `luggageS`).

An asynchronous message send immediately returns a *future*, which is a placeholder for the actual return value. Once this return value is computed, the future is said to be *resolved* with this value which "replaces" the future. Futures are objects to which asynchronous messages can be sent. These messages are accumulated as long as the future has not been resolved. Once the future is resolved, these accumulated messages are forwarded to its resolved value.

AmbientTalk uses a classic event-handling style by relying on blocks of code that are triggered by event handlers. These in-line event handlers are required when access to the actual return values of message sends are required. Event handlers are registered using functions that, by convention, start with `when`.

For example, a passenger must be notified of where he can retrieve his/her luggage that was removed from the plane. Hence, it is necessary that the value of `location` (from the previous code example) is resolved such that it contains the actual location.

```
// When a service classified as Aviapartner is discovered, it is accessible via "luggageS"
when: Aviapartner discovered: { |luggageS|
   // Send asynchronous message "rmvLuggage" to the discovered object
   when: luggageS<-rmvLuggage("BA472", "Peter", "Walker", "BA54282") becomes: { |location|
      /* When a reply is received, the value of the variable "location" equals the location
         where the luggage is stored and can be retrieved by the passenger */
   }};
```

In this example, two event handlers are registered. The first event handler, `when: discovered:`, is triggered when a service of type `Aviapartner` is discovered in the network. The `when: becomes:` event handler is triggered when a reply is received from the asynchronous message send.

3.2. Features of AmbientTalk

In this section we present some language features of AmbientTalk that are used when describing the implementation of our workflow language.

- **block**
  `{ | <parlist> | <body> }`
  This is an anonymous closure (also known as a lambda).
- **annotated messages**
  `obj<-msg()@annotation`
  Annotations can be used to override the default behaviour of futures. For instance, when a message send is annotated with `@Due`, the future is expected to resolve before a given deadline. When the timeout elapses before the future was resolved, the future is ruined with a `TimeoutException`.
- **failure detection**
  `when:becomes:catch: when[ever]:disconnected:`
  `when[ever]:reconnected: when:elapsed`
  When an asynchronously invoked method raises an exception, this exception can be caught asynchronously by using `when: becomes: catch:` instead of the `when: becomes:` construct. In order to deal with transient failures, `when: disconnected:` or `when: reconnected:` event handlers can be specified. These event handlers are triggered when the service disconnects or reconnects. However, in order to let these event handlers trigger every time a service of the same type disconnects or reconnects, the language also supports the constructs `whenever: disconnected:` and `whenever: reconnected`. The `when: elapsed:` event handler is used to invoke a code block when a timeout is elapsed.

3.3. Motivation for an AmbientTalk library

Although AmbientTalk is a distributed programming language targeted towards mobile ad hoc networks and deals with volatile connections at the heart of its programming model, developing large applications in this language is not straightforward.

In AmbientTalk, the application logic is divided amongst several event handlers which can be triggered independently of one another [13]. For small examples it is still manageable to understand the control flow. However, the control flow of large-scale event-driven applications can become very hard to understand. Applications implemented in AmbientTalk are not easily manageable as the control flow and the fine-grained application logic are interwoven. This makes AmbientTalk applications hard to maintain and difficult to reuse.

In the following sections we discuss how to add a 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 workflow abstractions. The control flow patterns and patterns for failure handling NOW introduces are available as a library for AmbientTalk. To experiment with the language, you can download a stand-alone version of the AmbientTalk interpreter with the NOW library [14].

4. Control flow patterns

Before presenting the different control flow patterns our workflow language supports, we first explain what a NOW program is. A NOW program consists of two main parts: a workflow description and an implementation of the connected services. The workflow description can use control flow and failure patterns and the data flow mechanism we explain in Section 5. This description resides on the backbone of the network such that it cannot become unavailable during its execution. A second part of the NOW program is the implementation of the services. These services reside on devices that are either connected via a reliable or unreliable communication link. Services are implemented as distributed AmbientTalk objects and are invoked by sending asynchronous messages (as shown in Section 3.1).

Our workflow language NOW has built-in support for most common control flow patterns. The current implementation consists of 15 control flow patterns of van der Aalst [10]: sequence, parallel split, synchronize, exclusive choice, simple merge, multichoice, structured synchronizing merge, multimerge, structured discriminator, structured partial join, multiple instances without synchronization, static partial join for multiple instances, multiple instances with a priori design-time knowledge, structured loop and implicit termination.

In this section we first show the syntax of these basic control flow patterns and describe how these patterns can be composed. Afterwards we show certain implementation details which highlight how we were able to use these patterns in a nomadic setting.

4.1. Syntax of patterns in NOW

The grammar of control flow patterns in NOW is shown in Backus–Naur form in Listing 2.

Listing 2. Abstract grammar of control flow patterns in NOW.

```
<component>        := <activity> | <pattern>
<activity>         := <service> "." <method> "(" <expression>* ")"
<condition_action> := "[" <block> "," <component> "]"
<pattern>          := <std_pattern> | <sync_pattern>
<sync_pattern>     := "Synchronize(" <component> ")" | "SimpleMerge(" <component> ")" |
                      "StructuredPartialJoin(" <component> ")" |
                      "MultiMerge(" <component> ")" |
                      "StructuredSynchronizingMerge(" <component> ")"
<std_pattern>      := "Sequence(" <component>+ ")" | "ParallelSplit(" <component>+ ")"  |
                      "Connection(" <sync_pattern> ")" |
                      "MultiChoice(" <condition_action>+ ")" |
                      "ExclusiveChoice(" <condition_action>+ ")"  |
                      "StructuredDiscriminator(" <component> ")" |
                      "StructuredLoop(" <component>+ ")" |
                      "MultInstWithoutSync(" <component> ")" |
                      "StaticPartialJoinMultInst(" <component> ")" |
                      "MultiInstWithPrioriDTKnowledge(" <component> "," <integer> ")"
```

As we can derive from the above syntax, each pattern consists of several components which can be either activities or patterns themselves. An activity is represented under the form `<service> . <method> "(" <expression>* ")"` which returns an `Activity` object that has a `start` method. When an activity must be executed, this method is called and it immediately returns a future object. Executing an activity results in discovering and invoking a service, which is implemented as a distributed object in AmbientTalk as shown in Section 3.1. Recall that this invocation is accomplished by sending an asynchronous message to the distributed object. Hence, an event handler needs to be installed to await the result from this message send. The `start` method of the Activity object resembles the next code (more details are given in Section 5):

```
1  /* When the start method is called, it returns a future and stores the futures
2     corresponding resolver, so it can be explicitly resolved at another moment in time */
3  def start() {
4      // Creation of an explicit future
5      def [result, resolver] := makeFuture();
6      when: service discovered: { |s|
7          /* Make a first class asynchronous message object, given a selector (string)
8             and arguments (table) */
9          def msg := createAsyncMessage(selector, args);
10         // Send the message object using the <+ operator
11         when: s<-msg becomes: { |result|
12             // To explicitely resolve the future
13             resolver.resolve(...);
14         };
15     };
16     result;
17 };
```

When the method finishes and a return value is received, the future is resolved (on line 5 in the code excerpt). For instance, when defining an activity

```
def my_activity := Trailer.rmvLuggage(plane);
```

the value of `my_activity` is an `Activity` object. When this object must be executed, its `start` method is called.

```
def result := my_activity.start();
```

The value of the variable `result` is a future object as long as the future is not resolved. When the service has been found and the service invocation has finished, the `start` method resolves the future with a certain value.[3]

Note that in the above syntax the distinction between standard patterns (`std_pattern`) and synchronization patterns `sync_pattern` is made. This is necessary because these two kinds of patterns are handled in a different way when they are composed, as we explain in the subsequent section.

[3] The future is resolved with an environment object as is explained later on in Section 5.

4.2. Composition of control flow patterns

We first describe the composition of standard control flow patterns and then explain how synchronization patterns can be composed.

Fig. 2 depicts the composition of a sequence and a parallel split. This small workflow models the scenario where a plane has landed and activities must be executed in parallel. In this example the flight attendant is reminded to open the door of the plane, and in parallel a trailer must remove the luggage from the plane and transport it to a luggage belt. The implementation of this scenario in NOW is shown in Listing 3. How the variables (`plane` and `belt`) are handled is discussed later on in Section 5 where we present how data is passed between the activities of the workflow.

Fig. 2. Control flow patterns: parallel split with a sequence as its first branch.

Listing 3. Composition of a sequence and parallel split.

```
def seq := Sequence( Trailer.rmvLuggage(plane), Trailer.transport(belt) );
ParallelSplit( seq, FlightAttendant.openDoor() );
```

Composition of patterns becomes more complex when synchronization patterns are involved. In Listing 2 the distinction between standard and synchronization patterns is made as synchronization patterns need to support multiple incoming branches. These incoming branches need to be linked to the synchronization patterns using so-called `connections`.

Connections are necessary for complex compositions where not all outgoing branches, for instance, of a parallel split pattern, are connected to the same synchronization pattern. When all branches of a split pattern are joined by one synchronization pattern, it would be possible to write the following code

```
Sequence( ParallelSplit( branch1, branch2, ...), Synchronize( ... ) )
```

which would mean that first a parallel split is executed and afterwards all its branches are merged by a synchronize. However, it is also possible that the branches of the parallel split are merged by several different synchronization patterns. Hence, for each branch it must be specified which synchronization pattern is responsible for its merging process.

We explain the way synchronization patterns are composed using a small workflow example. Consider a scenario where several things need to happen before a plane can take off. First of all, catering must supply the plane with the necessary food and beverages, and the luggage of the passengers must be loaded. Furthermore, before passengers can board, the plane must be cleaned and a flight attendant must check whether all personnel are on board. Fig. 3 shows a workflow diagram for this example scenario.

This workflow example can be written in NOW as shown in Listing 4.

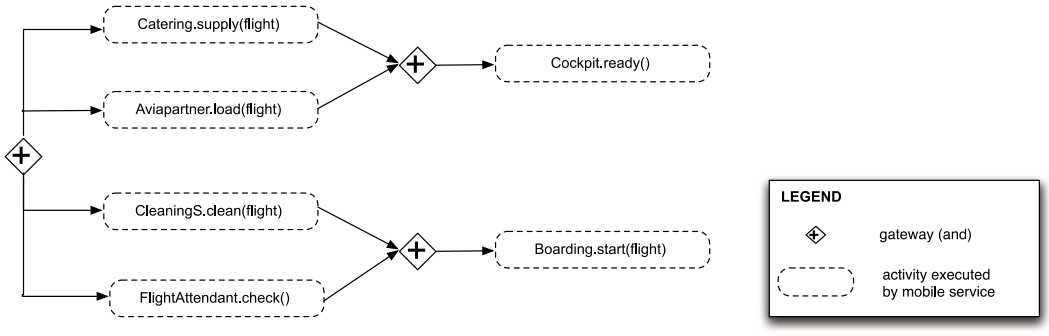


Fig. 3. Control flow patterns: parallel split followed by multiple synchronization patterns.

Listing 4. Composition of a parallel split and synchronize.

```
1  def sync1 := Synchronize( Cockpit.ready() );
2  def sync2 := Synchronize( Boarding.start(flight) );
3  ParallelSplit( Sequence( Catering.supply(flight), Connection(sync1) ),
4                 Sequence( Aviapartner.load(flight), Connection(sync1) ),
5                 Sequence( CleaningS.clean(flight), Connection(sync2) ),
6                 Sequence( FlightAttendant.check(), Connection(sync2) ) );
```

Connection patterns are used by NOW to wrap a synchronization pattern and inform it of its number of incoming branches. In this example, not all outgoing branches of the parallel split pattern are connected to the same synchronize pattern. This is achieved by writing, for each branch of the parallel split, to which synchronization pattern it must be connected. The first two branches of the parallel split are connected to the first synchronize pattern as we can see on lines 3 and 4 of the code excerpt. A second synchronize pattern is connected to the other two outgoing branches of the parallel split (line 5,6).

4.3. Implementation description

In this section we show the implementation details of the composition of NOW's control flow patterns. First we present an implementation description for standard patterns and continue with the implementation of synchronization patterns.

Each control flow pattern is implemented as an AmbientTalk function that returns an object which is instantiated with the arguments of the function call. Each resulting object has a `start` method that is executed when the pattern is started and returns a future, similar to the execution of an activity. Hence, when the actual return value of the pattern is needed, an event handler needs to be installed:

```
def seq := Sequence( ... );          // Returns an object with a start method
def result := seq.start();           // Returns a future
when: result becomes: { |r| ... };
```

As was already clarified in Section 4.1, each pattern consists of several components which can be either activities or patterns themselves. Executing an activity results in the invocation of a service. Since we want to interact with activities in the same way as patterns in our implementation, the execution of a pattern must also return a future. Hence, a necessary event handler needs be installed to wait for its termination (the last line in the code snippet).

Listing 5 shows the implementation of a standard workflow pattern, in particular a sequence pattern. The implementation of the other standard patterns resembles this code, but differs in the way the component(s) of the pattern are executed.

Listing 5. Implementation of sequence.

```
1  def Sequence(@args) {
2       object: {
3            def components := args;
4
5            def start() {
6                 def [result, resolver] := makeFuture();
7                 execute(1, resolver);
8                 result;
9            };
10
11           def execute(idx, resolver) {
12                def component := components[idx];
13                if: (idx < components.length()) then: {
14                     run(component, {| _ | execute(idx+1, resolver)});
15                } else: {
16                     run(component,  {| _ | resolver.resolve(_) });
17                }; };
18      } taggedAs: [Pattern];
19 };
```

When a sequence pattern is defined, the function `Sequence` is called with a variable number of components. These components are grouped in a table (`@args`) and an object is returned where the field `components` is bound to that table (line 3 in Listing 5). When the pattern is started, a *future* is defined (line 6) and returned (line 8). When the future is resolved, the future object is replaced by the actual return value.

The execution of each pattern is specific to the behaviour of that pattern. In case of a sequence, every component must await the completion of its previous component. The execution of a component is implemented by the `run` method, which is common for all patterns (both standard and synchronization patterns). The code snippet in Listing 6 shows the implementation of that method.

Listing 6. Run method of control flow patterns.

```
1  def run(component, continue := { | _ | _ }) {
2       if: (is: component taggedAs: Activity) then: {
3            // Component is an activity, wrapping a service object
4            when: component.service.tag discovered: { |service|
5                 // Make a first class asynchronous message object, given a selector (string)
6                 def msg := createAsyncMsg(component.selector);
```

```
7                // Send the message object using <+
8                when: component.service <+ msg becomes: { |reply|
9                    continue(reply); } };
10       } else: {
11            // Component is a pattern, the engine must start it and then await its completion
12            when: component.start() becomes: { |reply|
13                 continue(reply);  }
14             } };
```

The `run` method of a pattern has one argument, a component, and an optional argument `continue` which is an AmbientTalk block. The `run` method starts the component and when its execution is finished it will proceed to call the `continue` block. This block represents the next execution step of the current pattern.

When the component is an activity (line 3–7), a service of the given type tag must be discovered (line 4) and when one is found an asynchronous message must be sent to that service. Given that an asynchronous message send returns a future, an event handler must be installed to await this future to be resolved (line 6). When the service has returned a result, the future is resolved and the statements inside the event handler are executed (line 7).

When the component that must be executed is a workflow pattern, its execution is started. This is achieved by calling its `start` method (on line 10), which returns a future. When the workflow pattern has finished its execution, this future is resolved and the `continue` block is called with the resolved value.

As we already mentioned, the execution of each control flow pattern differs in the way its components are executed (e.g. sequential, in parallel, etc.). This specific behaviour is implemented by the `execute` method of the pattern (line 11–17 in Listing 5). For the implementation of the sequence pattern, the difference between the execution of the all-but-last and last component has to be made since only after the last component has finished its execution can the future (returned by the pattern when started) be resolved. Hence, the second argument (`continue`) of the `run` method is different when the last component of a sequence is executed. For the all-but-last components of the sequence, the value of `run`'s second argument is an AmbientTalk block which calls the `execute` method of the sequence with an increased index (line 14 of Listing 5). In contrast, on line 16 we see that this value is an AmbientTalk block which resolves the future, as at that moment all components of the sequence have finished their execution.

We now describe the implementation of a synchronize pattern whose implementation is shown in Listing 7. Note that this implementation has some extra functionality compared to the one presented in our earlier paper [6], for example, making the pattern reentrant.

Listing 7. Implementation of synchronization.

```
1  def Synchronize(cmp) {
2       object: {
3            def incomingBranches := 0;
4            def started := 0;
5            def component := cmp;
6
7            def addSync() {
8                 incomingBranches := incomingBranches + 1;
9            };
10
11           def start() {
12                def [result, resolver] := makeFuture();
13                started := started + 1;
14                execute(resolver);
15                result;
16           };
17
18           def execute(resolver) {
19                if: (incomingBranches == started) then: {
20                     run(component, { | _ | resolver.resolve(_) });
21                }; };
22      } taggedAs: [SyncPattern];
23 };
```

Recall that synchronization patterns must be composed using a `connection` pattern. Upon instantiation, such a connection notifies its enclosed synchronization pattern that there is another incoming branch. This is achieved by calling the `addSync` method (line 7–9 in Listing 7). Another difference between the standard control flow patterns and the implementation of the synchronization patterns is the field `started` on line 4. Each time the method `start` is called, the

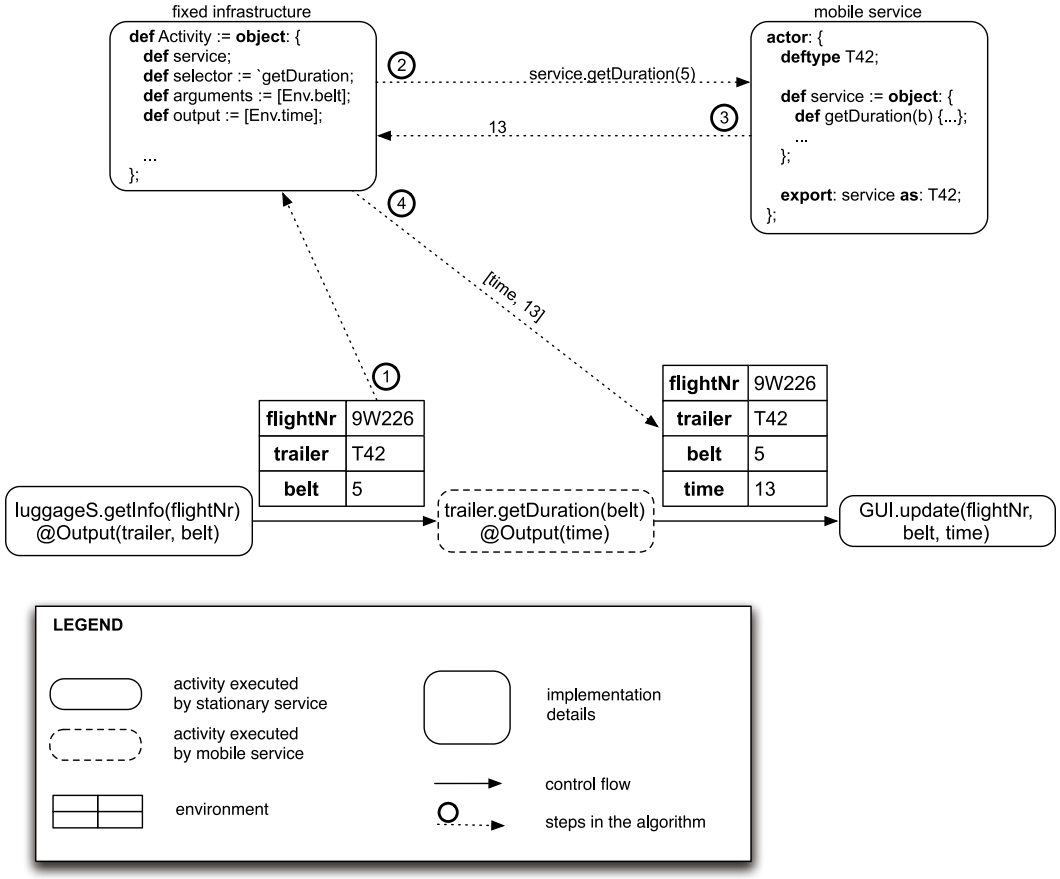


Fig. 4. Passing environments in a sequence pattern.

value of this field is incremented (line 13). This is done in order to ensure that the component is executed only when all incoming branches are resolved (line 19, 21). Note that the run method of synchronization patterns is the same as the one given previously in Listing 6.

5. Data flow in nomadic networks

In this section we describe the data flow mechanism that enables passing information between (possibly non-consecutive) activities (requirement 3 in Section 2.2). We present the mechanism used for parameter bindings when invoking a service (executing an activity). The formal syntax for an activity is extended as shown below:

```
<activity> := <service> "." <method> "(" <expression>* ")" [ "@Output(" <symbol>* ")" ]
```

When an activity is executed, the service's method is invoked with its formal parameters bound to their values, which are looked up in the *environment*. The environment is an object with a unique identifier and a dictionary associating variables with values. The invoked method can return a number of result values which are bound to the corresponding variables specified using the `@Output` syntax. Returning the wrong number of values or parameters that are not found in the current environment results in an error. In the example below, a luggage service is invoked in order to retrieve the belt number where the luggage can be collected and the trailer that is responsible for bringing the luggage to that belt.

```
LuggageService.getInfo(flight)@Output(trailer, belt)
```

Executing this activity results in the discovery of a `LuggageService` and the invocation of the method `getInfo` on the found service. This method call returns a table whose values are bound by the activity to the variables `trailer` and `belt`, respectively.

Instead of using a simple global or static environment for our workflow language, we developed a dynamic system where the environment flows through the workflow graph and is dynamically adapted. Each time a workflow is started, a new environment is instantiated and passed to this workflow instance. Hence, in order to support multiple instances patterns, it suffices to start each instance with a copy of the passed environment with a new unique id.

Fig. 4 shows an example of how an environment gets updated in each step of a sequence pattern. The dotted arrows are annotated with a number specifying the different steps in the algorithm. First, the environment is passed to the activity of the workflow. The rounded rectangle in the upper-left corner of the picture shows the source code of the second activity.

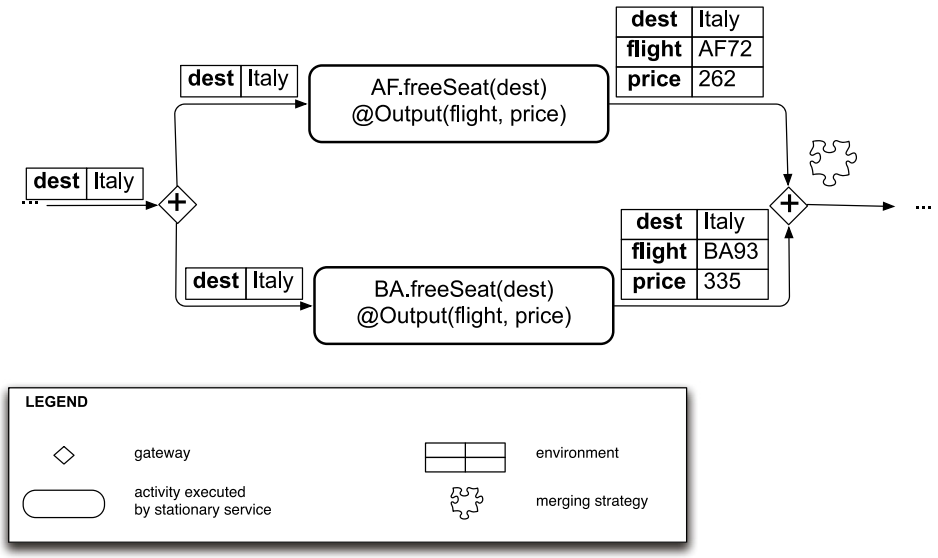


Fig. 5. Environments: merging strategies for synchronization patterns.

Such an activity is an AmbientTalk object that resides on the fixed infrastructure of the nomadic network. The activity looks up the arguments in the environment and passes these values to the service invocation (step 2). The service resides on a (mobile) device and its implementation is depicted in the upper-right rounded rectangle. Note that first a service of the right type tag must be discovered before it can be invoked. When the service execution finishes, a result is sent to the activity (step 3) and the activity adds a new variable binding to the environment (step 4), which is sent to the subsequent activity in the sequence.

This mechanism can be thought of as dynamic scoping but with special semantics for patterns such as a synchronization, which merges multiple incoming branches, as is illustrated in Fig. 5. If part of a workflow can be reached by more than one path, it is possible that the environments are (completely) different.

As is shown in the example in Fig. 5, when a *parallel split* is encountered, the environment is conceptually duplicated for each outgoing branch. Further adaptations of the environment are local to each branch. However, when a synchronization point is reached, the environments from all incoming branches need to be merged. We have identified several possible merging strategies that might be useful in different cases:

- Prioritize one of the incoming branches when resolving conflicts.
- Pick the environment of one incoming branch and ignore the others.
- Merge conflicts into a table containing the different values (with or without duplicates).
- Remember the "scope" from before splitting and restore it (not always desirable).
- Employ a user provided function to resolve conflicts.

The small workflow depicted in Fig. 5 represents a scenario where a booking agent looks for a free seat on a flight to Italy. This is achieved by invoking two airline companies (Air France and British Airways) services. When both results are received, the passed environments need to be merged. When the first merging strategy is used, and Air France has a higher priority than British Airways, the resulting environment is the environment that is passed from the Air France activity to the synchronize pattern. The second merging strategy will just randomly pick one of those two environments. When the third merging strategy is used, the resulting environment contains the following bindings: `[flight, [AF72, BA93]]` and `[price, [262, 335]]`. The fourth strategy results in the environment with only one variable binding `[dest, Italy]`. When the fifth merging strategy is used and the user provided function is to *take the flight with the lowest price*, the resulting environment is the one where price equals 262.

Synchronization patterns can be initiated with a merging strategy that specifies how the incoming environments need to be merged. For instance, when the last merging strategy is used for the example above the following code needs to be written in NOW:

```
1  def my_strategy := { | envs| def env := envs[1];
2                               def price := envs[1].find(`price);
3                               envs.each: { |e| if: ( e.find(`price) < price ) then: {
4                                                    env := e;
5                                                    price := e.find(`price) }; };
6                               env; }
7  def sync := Synchronize( ..., my_strategy);
8  def parSplit := ParallelSplit( Sequence( AF.freeSeat(dest)@Output(flight, price),
9                                           Connection(sync) ),
```

```
10                  Sequence( BA.freeSeat(dest)@Output(flight, price),
11                            Connection(sync) ) );
```

So, as we can see on the seventh line in this piece of code, the synchronize pattern is initiated with an extra argument. Hence, the grammar we gave earlier in Listing 2 is adapted for synchronization patterns such that they have a second argument (which is an AmbientTalk block).

Depending on the merging strategy chosen for a synchronization pattern upon creation of a new workflow, it should be taken into account that the services used after that synchronization point must be able to cope with values that are tables instead of atoms. This does not pose any restrictions on the workflow. Sometimes this behaviour is even wanted. For instance, when a booking agent wants to show all available flights to a certain destination to his/her costumers. Hence, in examples where information from several services needs to be collected, this third merging strategy is useful.

By introducing this *environment passing style*, we satisfy the key requirements of a data flow mechanism in a workflow model. As Sadiq et al. stated [15], a data flow model must have the ability to:

- manage both the input and output data of activities;
- ensure that data produced by one activity is available for other activities; and
- ensure consistent flow of data between activities.

The first requirement is fulfilled, since the formal parameters are looked up in the environment before starting the execution of an activity. After this execution, the output values are associated with their variable names and added to this dictionary. By introducing the notion of an environment, which flows through the entire workflow, we ensure that the second requirement is satisfied. The last requirement is fulfilled because of the straightforward passing of data in sequence patterns, and the merging strategies we provide in case of multiple branches.

The dynamic approach we propose provides a much richer and more powerful mechanism than a globally shared environment or simple flow of output values (from one activity to the next) could achieve. As in our approach the data flow is attached to the control flow, it is possible that local changes to the environment can occur in different branches. The dynamic mechanism we offer eases the burden of manually arranging static scopes of variables.

Implementation description

In this section we explain how the implementation of the control flow patterns needs to be adapted in order to support the dynamic data flow mechanism we presented above.

When an activity is started, it receives the current environment, looks up its parameters, discovers a service of the right type, invokes that service, and binds the resulting values to the specified output variables. In order to do so, the `start` method of a control flow pattern must have the incoming environment as its argument (as is shown on line 12 in Listing 8). Note that looking up parameters in the environment, the method invocation itself, and the extension of the environment with the new variable bindings, are handled by the run method (of which the basic implementation is shown in Listing 6).

In order to let synchronization patterns support the data flow mechanism, some more adjustments to the implementation have to be made. Recall that when a workflow is (re)started the only action that must be performed is passing a new environment (with a unique identifier) to that workflow instance. The execution of the different workflow instances share the same objects that implement the different patterns and activities of the workflow's description. So, when multiple instances of the same workflow are executed there is no guarantee of the order in which the incoming branches of a synchronization pattern are enabled. It is possible that the first incoming branch is enabled for each workflow instance before any other branch is enabled. For instance, consider the following code snippet where a parallel split is followed by a synchronize pattern. In this example we use the notation `[branch, sync]`, which is syntactic sugar for `Sequence(branch, Connection(sync))`.

```
def sync := Synchronize(...)
def parSplit := ParallelSplit( [branch1, sync], [branch2, sync], [branch3, sync] )
```

When this small workflow is started two times (by executing `parSplit.start( env )` twice), it is possible that three enablements of outgoing branches of the parallel split are originated from different workflow instances. Hence, for the implementation of synchronization patterns it is not sufficient to count the number of incoming branches that have terminated, as the `sync` pattern can be started three times where two of them result from a different workflow instance, which must not result in the success of the synchronize pattern. To deal with this problem, the implementation of the synchronize patterns is extended (as is shown in Listing 8).

Listing 8. Synchronize with support for data passing.

```
1  def Synchronize(cmp, _strategy) {
2      object: {
3          def incomingBranches := 0;
4          def component := cmp;
5          def strategy := _strategy;
```

```
6           def envs := [];
7
8           def addSync() {
9                incomingBranches := incomingBranches + 1;
10          };
11
12          def start(env) {
13               def [result, resolver] := makeFuture();
14               envs := envs + [env];
15               execute(env.id, resolver);
16               result;
17          };
18
19          def execute(idValue, resolver) {
20               def envsOfId := envs.filter: {|env| env.id == idValue};
21               if: (envsOfId.length() == incomingBranches) then: {
22                    def nEnv := Environment.new();
23                    nEnv.merge(envsOfId, strategy);
24                    run(component, nEnv, { | _ | resolver.resolve(_) });
25               }; };
26     } taggedAs: [SyncPattern];
27 };
```

In this extended implementation, environments passed to the pattern are stored (line 6, 14) and only when the number of environments of the same id equals the number of incoming branches, synchronization has succeeded and the next component may be executed (lines 22–25). Also note that the pattern has two arguments; the second argument is used for determining how incoming environments need to be merged (line 23).

6. Compensating actions

In a dynamically changing environment, the challenge is to make the large heterogeneity of services co-operate and deal with their transient and permanent failures. AmbientTalk already has built-in support to handle both disconnections and reconnections with the event handlers `when: disconnected:` and `when: reconnected:`. In NOW, we provide a *Failure* pattern which wraps a part of a workflow and imposes compensating actions and strategies (requirement 5 in Section 2.2). Since we want to handle (transient) failures at different levels of granularity, the failure pattern can be used on one specific service or wrap an entire subworkflow. Russell et al. [16] already classified workflow exception patterns used by workflow systems. The exceptions he discusses are, for instance, *constraint violation*, *deadline expiry* and *work item failure*. The failures we support are specific to the (temporary) network failures that can arise, although some basic exception handling can be achieved by using the `exception` failure. Examples of events we capture in a failure pattern are disconnections, reconnections, timeouts and possibly service exceptions/errors. Possible compensating actions include retrying, rediscovery (potentially yielding a different service), skipping, waiting or executing a specific subworkflow to handle the event.

Listing 9. Abstract grammar of failures and compensations.

```
<component>        := <activity> | <pattern>
<pattern>          := <sync_pattern> | <std_pattern> | <failure_pattern>
<failure_pattern>  := "Failure(" <component> ", [" <failure>+ "] )"
<failure> := "Timeout(" <integer> "," <compensation> ")" |
             "Disconnect(" <compensation> ")" |
             "Exception(" <symbol> "," <compensation> ")" |
             "NotFound(" <compensation> ")"
<compensation>  := "Retry(" [ <integer> "," <compensation> ] ")" |  "Skip()" |
                   "Rediscover(" [ <integer> "," <compensation> ] ")" |  <component> |
                   "Wait(" <integer> "," <compensation>* ")" |
                   "Restart(" [ <integer> ","  <compensation> ] ")"
```

Listing 9 shows the abstract grammar of the *failure* pattern and its possible compensating actions. A compensating action is not always successful, hence we provide a way of limiting the number of times each compensating action is tried. When a compensating action has reached its maximum attempts, another (more drastic) one can be tried.

The compensating action *retry* tries to invoke the failed activity (not the entire wrapped workflow) a number of times. *Rediscover* will (re)discover a service with the same type tag (which might result in invoking the same service when only one is available). The *skip* compensation just skips the *entire* wrapped part of the workflow, whereas *restart* restarts it. We also provide a *wait* compensating action, which will simply wait for a specified time before trying the next compensating action. Another possible compensating action is the execution of a subworkflow (activity or a pattern).

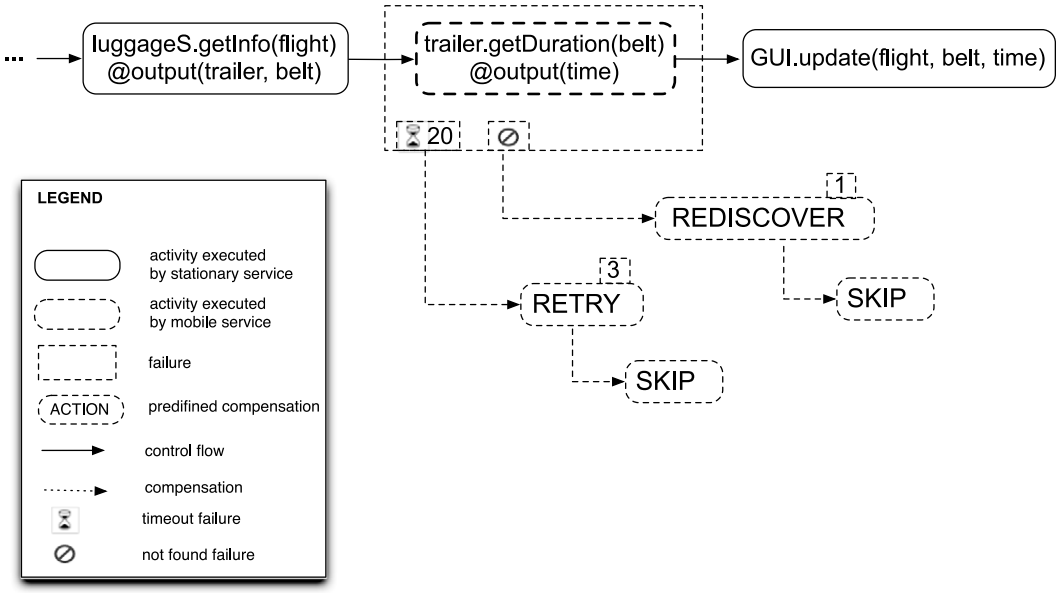


Fig. 6. Compensating actions specified for different kinds of failures.

Note that in some compensating actions the parameters are optional, indicating that the compensating action will be tried until it succeeds.

Consider the example of updating a screen at the airport with the necessary information (flight number, belt number, estimated duration before luggage is on the belt) so that passengers can retrieve their luggage. By using the failure pattern (drawn as a dashed rectangle wrapping part of a workflow), we can specify compensating actions, as is shown for the trailer service in Fig. 6.

When the second activity has a timeout (after 20 s no reply is returned), we try to resend the message three times. If this is still unsuccessful we move on to the next compensating action, which just skips this activity (so no time gets displayed on the screen). In case of a disconnection, however, a service of the same type must be (re)discovered. After trying to rediscover a trailer service one time, the wrapped activity is also skipped. The implementation of this workflow in NOW can be seen in the following code snippet.

```
Sequence( luggageS.getInfo(flight)@Output(trailer, belt),
          Failure( trailer.getDuration(belt)@Output(time),
                   [ Timeout( 20, Retry(3, Skip()) ),
                     Disconnect( Rediscover(1, Skip()) ) ]),
          GUI.update(flight, belt, time) );
```

Failure patterns can be nested, so different strategies can be formulated on different levels of granularity. A whole workflow can be surrounded by a failure pattern specifying "after a disconnection, wait 20 s and then try to rediscover" and smaller parts of this workflow can be wrapped with more specific failure patterns, which possibly override (shadow) the behaviour imposed by the outermost failure pattern. In NOW an inside-out policy is used to determine the compensations that must be executed for an occurred failure event.

As the first requirement in Section 2.2 stated, a workflow language targeting nomadic networks must support both stationary and mobile services. Mobile services are wrapped with a `Failure` pattern such that these services can be rediscovered in case of a disconnection. Transparently compensating possible disconnections of mobile services is hence also satisfied (requirement 4).

Implementation description

In order to support the detection of failures and to compensate for them, we need to extend the implementation of the `run` method we showed earlier in Listing 6. In order to do so, our implementation makes use of the event handlers supported by AmbientTalk, such as `when: disconnected:`. The compensations that are specified by the failure pattern are stored in the environment. The environment contains at any point in time bindings for the variables `notFound`, `exception`, `timeout`, and `disconnection`. These variables contain the active compensating actions for their respective failures. Such a compensating action is an object (`Retry`, `Rediscover`, `Restart`, `Skip`) that contains the component that must be executed, the maximum number of times the compensating action can be tried, and the number of times the compensating actions has already been executed.

Listing 10 shows an extended implementation of the `run` method we showed in Listing 6.

Listing 10. Run method with support for failure detection and compensation.

```
1  def run(component, env, continue := { | _ | _ }) {
2       if: (is: component taggedAs: Activity) then: {
3            // Component is an activity, wrapping a service object.
4            execute(component, env, continue);
5       } else: {
6            // Component is a pattern, the engine must start it and then await its completion
7            when: component.start(env) becomes: { |reply|
8                 continue(reply); }} };
9
10 def execute(activity, env, continue) {
11      when: activity.service.tag discovered: { |service|
12           def timeout := env.find(`timeout);
13           def output := activity.output;
14           def args := activity.parameters;
15           def method := activity.selector;
16            // Make a first class asynchronous message object
17           def msg := createAsyncMsg(method, env.bind(args));
18           if: (is: timeout taggedAs: TimeoutType) then: {
19                def d := timeout.duration;
20                 // Make a first class asynchronous message object annotated with @Due
21                 msg := createTimeoutMsg(method, env.bind(args), d); };
22           // Send the message object using <+
23           when: service <+ msg becomes: { |reply|
24                if: (is: reply taggedAs: Table) then: {
25                     1.to: reply.length+1 do: { |idx|
26                         env.insert(output[idx].variable, reply[idx])} };
27                continue(reply);
28           } catch: TimeoutException using: { |e|
29                def comp := timeout.compensation;
30                compensate(activity, comp, env, continue); };
31           whenever: service disconnected: {
32                def disconnection := env.find(`disconnection);
33                def comp := disconnection.compensation;
34                compensate(activity, comp, env, continue); };
35      };
36     when: seconds(20) elapsed: {
37           def notFound := env.find(`notFound);
38           def comp := notFound.compensation;
39           compensate(activity, comp,  env, continue)} };
```

We now describe how each of these failures are detected and handled. The detection of a *not found* failure is implemented by making use of a timer. After a period of time, we conclude that a certain service has not been found. On line 33 AmbientTalk's `when: elapsed:` event handler (explained in Section 3.2) is used for this functionality. When the timeout is elapsed, first the failure object is retrieved from the environment (line 34) and then its compensating action is executed (line 35–36). The `compensate` function on lines 27, 31, and 36 is responsible for executing the compensation that was specified for the detected failure.

Disconnection failures are handled in a similar way, these failures are captured using the `whenever: disconnected:` event handler (line 28). In this case, the disconnection failure object is retrieved from the environment and its compensating action is executed (line 29–31).

Timeout failures require a slightly different approach since the asynchronous message that is sent to the service must expire after some time. When no compensating action is specified for a timeout failure, a regular asynchronous message is sent: `service <- selector( args )` (line 16, 20). In case a compensation is specified, the service invocation needs to look like `service <- selector( args ) @Due( time )` (as can be seen on lines 17–19). When the asynchronous message send expires, a `TimeoutException` is raised and caught by the `catch` block on lines 25–27. Here again, the compensating action for the timeout failures is executed. When no `@Due` annotation is specified, it is not possible to catch timeout exceptions.

Note that in the implementation we show in Listing 10, the support for data flow is implemented. First of all, the run method has an extra parameter `env` (line 1), and the parameters of an activity are looked up before an asynchronous message is sent to a service (`env.bind( args )` on line 16). Furthermore, the environment is extended with new variable bindings for the output variables of the activity (line 22–23). The AmbientTalk construct `to: do: (Number end, Closure code)` on line 22 iterates from the given value (1 in our code) to the one that is passed as an argument (`reply.length + 1`) applying the given closure at each iteration.

7. Validation

The nomadic workflow language we presented in this paper is related to research domains such as ambient intelligence. These research domains are very active and serve a wide range of applications. In previous sections we used example scenarios that can occur in an airport, which is one possible application domain. Nomadic networks can also be expected in health-care, domotica, and traffic.

Nowadays, in order to implement these kinds of applications for environments with volatile connections, application developers can depend on languages like AmbientTalk, where network failures are tackled at the heart of their programming model. As validation of our work we show that implementing applications for a nomadic network is still not straightforward in a language like AmbientTalk. We highlight some problematic properties (such as maintainability and readability) of programs written in an event-driven style as in AmbientTalk and show how workflow abstractions can be used to overcome these.

We first show the implementation of the scenario at the airport, where we focus on the composition of control flow patterns. In a second example we illustrate how the detection and handling of failures is handled by using an example application at a warehouse. Afterwards, we present the results of experiments used to measure the overhead of introducing workflow abstractions on top of AmbientTalk and also measure the overhead introduced by failure detection.

7.1. Example application stressing control flow

Our first example application implements the scenario at the airport described in Section 2. This application focuses on the composition of control flow patterns. For conciseness we omit the detection and handling of failures, which is explained in detail in the subsequent example application (Section 7.2). We first describe the implementation in NOW and afterwards show parts of the direct-style implementation in AmbientTalk. The interested reader can find the entire implementation at our dedicated website [14].

Listing 11 shows the implementation of the example application in our nomadic workflow language.

Listing 11. Implementation of the airport scenario in NOW.

```
1  def seq := Sequence( BoardingResponsible.waitForPassenger(passenger),
2                       ControlTower.takeoff(flight) );
3  def structDiscr := StructuredDiscriminator( seq );
4
5  def sync2 := Synchronize( ControlTower.takeoff(flight) );
6  def parSplit2 := ParallelSplit( [ LastMinute.freeSeat(flight, passenger), sync2 ],
7                                   [ LuggageS.removeLuggage(flight, passenger), sync2 ],
8                                   [ BoardingResponsible.closeGate(flight), sync2 ],
9                                   [ MessagingS.gateClosed(flight, passenger), sync2 ] );
10 def sync1 := Synchronize( parSplit2 );
11
12 def exc1 := ExclusiveChoice( [{ |found| found }, structDiscr],
13                               [{ |found| ! found }, Connection(sync1)] );
14 def exc2 := ExclusiveChoice( [{ |found| found }, structDiscr],
15                               [{ |found| ! found }, Connection(sync1)] );
16 def exc3 := ExclusiveChoice( [{ |found| found }, structDiscr],
17                               [{ |found| ! found }, Connection(sync1)] );
18
19 def parSplit := ParallelSplit( [ MessagingS.reminder(passenger)@Output(found), exc1],
20                                 [ Announcement.missing(passenger)@Output(found), exc2 ],
21                                 [ Assistance.missing(passenger)@Output(found), exc3 ]);
```

This implementation uses five standard control flow patterns of van der Aalst (sequence, structured discriminator, synchronize, parallel split, and exclusive choice). The workflow starts with a *parallel split* pattern (`parSplit` on lines 19–21). All branches of this *parallel split* are connected to an *exclusive choice* pattern (defined on lines 12–17) where the control flow either continues to a *structured discriminator* or a *synchronize* pattern. In case the control flow is passed to the *structured discriminator* (line 3), two more activities must be executed in sequence (line 1–2). When the value of the variable `found` equals false, the *synchronization* pattern `sync1` is executed. This gives rise to the execution of another *parallel split* (named `parSplit2` on line 6–9) which invokes four services and continues with a *synchronization* pattern. When all four services have terminated their invocation, this *synchronization* pattern `sync2` continues the execution of the workflow by invoking the last activity (line 5).

We now show part of the implementation of the example application in AmbientTalk in Listing 12. We present the implementation of lines 12–21 of the code shown in Listing 11 which only implements the first *parallel split* pattern with its three outgoing branches which each contain an exclusive choice pattern.

Listing 12. Implementation of part of the airport scenario in AmbientTalk.

```
1  def missingPerson(person, flight) {
2       def results := [];
3
4       def check(reply) {
5            def nbrTrue := 0;
6            def nbrFalse := 0;
7            results.map: { |e| if: (e) then: {nbrTrue := nbrTrue + 1}
8                              else: {nbrFalse := nbrFalse + 1} };
9            if: ( (reply).and: {nbrTrue == 1} ) then: {
10                // Only the first time "true" is retrieved, call "personFound" function
11                personFound(person, flight);
12           } else: {
13             if: (nbrFalse == 3) then: {
14                 /* Only when all branches have resulted in "false", the function
15                    "personNotFound" is called */
16                 personNotFound(person, flight) } };
17      };
18
19      // First branch of the parallel split pattern
20      // Discover the service of type MessagingService
21      when: MessagingService discovered: { |service|
22           // Invocation of the found service by sending an asynchronous message
23           when: service<-reminder(person) becomes: { |reply|
24               results := results + [reply];
25               check(reply) } };
26
27      // Second branch of the parallel split pattern
28      // Discover the service of type AnnouncementService
29      when: AnnouncementService discovered: { |service|
30            // Invocation of the found service by sending an asynchronous message
31           when: service<-missingPerson(person) becomes: { |reply|
32               results := results + [reply];
33               check(reply) } };
34
35      // Third branch of the parallel split pattern
36      // Discover the service of type Assistance
37      when: Assistance discovered: { |service|
38           // Invocation of the found service by sending an asynchronous message
39           when: service<-missingPerson(person) becomes: { |reply|
40                results := results + [reply];
41                check(reply) } }
42 };
```

In order to implement the parallel split pattern in plain AmbientTalk, the result of the service invocation of each branch must be stored (in the table `results` on line 2). Each time a future is resolved, we check if the reply is true (which means the missing person is found) and if so, we check that this is the first time this answer is retrieved. This is necessary since three services are invoked in parallel and we want to model a *structured discriminator* pattern, which means we can continue only once with the function call `personFound( person, flight )`, namely the first time true is retrieved as a result. This implementation can be found on lines 9–11 of Listing 12. In case the future is resolved with the value false, the result is added to the `results` table and the function `personNotFound( person, flight )` is executed only when this table contains as many times false as the number of branches of the *parallel split* (three in this specific case). The implementation can be found on lines 13–16.

We claim that the implementation of this example application is more readable in our nomadic workflow language compared to a direct style implementation in AmbientTalk. In order to confirm/validate this claim however further user tests with a representative group of people need to be conducted. Not only is the implementation in NOW more compact (21 compared to 89 lines of code), it is also better structured (a result of employing patterns).

An unwanted property of AmbientTalk is the fact that the application logic is divided amongst several event handlers that can be triggered independently of one another [13], since the language has an event-driven architecture. The control flow of an application is thus no longer determined by the programmer but by external events. This phenomenon is known as *inversion of control*. This makes the event-driven programming style not always straightforward or suitable for large-scale applications [17]. As a library of this ambient-oriented programming language, NOW hides these event handlers for the application developers. In the implementation of the workflow language, these event handlers can be found, amongst others, in the `run` method whose implementation is shown in Listing 10.

Moreover, as we can see in the code excerpt in Listing 12, the implementation of the three branches of the parallel split are very similar. The only difference between them is the name of the service that must be discovered and the asynchronous message send (line 21–25, 29–33, and 37–41 in Listing 12). There is no way this can be abstracted easily in AmbientTalk, since this is conceptually the asynchronous version of initializing a reference and sending it a message.

From this we can conclude that the implementation in NOW is more reusable since the control flow and fine-grained application logic are not interwoven, which makes it also more maintainable.

7.2. Example application stressing the detection and handling of failures

The second example application we show, focuses on the detection and handling of failures. We show that although AmbientTalk has built-in support for both permanent and transient failures, the implementation of such an application using failures is not straightforward.

Carol uses her iPhone to store her shopping list for IKEA. When she enters the warehouse, her shopping list is transmitted and the warehouse infrastructure checks whether all her articles are in stock. At the same time, Carol receives recommendations of articles she might also be interested in. If all articles of Carol's shopping list are available, a map of the warehouse is depicted on her iPhone showing where she can find all her articles. In case not all articles are present, Carol receives a message on her mobile phone that states that she needs to go to the customer service where they will try to find a solution to the problem. But since Carol is standing in the basement of IKEA, it is possible that her iPhone has no connection and an announcement is made that she needs to go to the customer service.

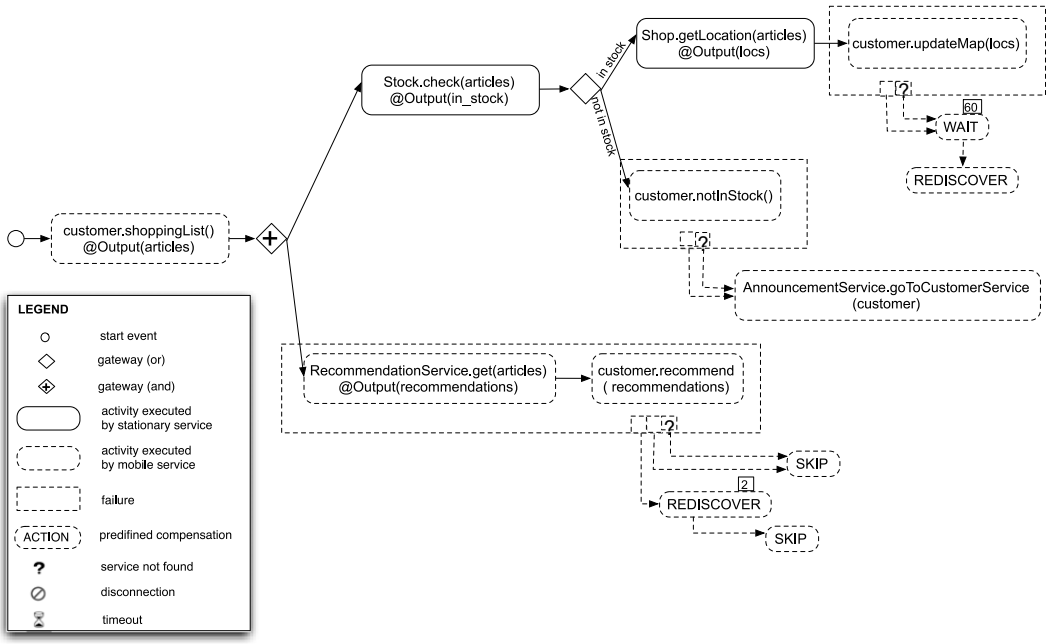


Fig. 7. Workflow description of the scenario at IKEA.

Fig. 7 gives a representation of this example. For each activity denoted with a dashed rectangle, compensating actions for certain kinds of failures are defined. Recall that our nomadic workflow language automatically handles failures in communication with mobile services (requirement 4 in Section 2.2). Hence, the activities `customer.shoppingList( ) @Output(articles)` and `AnnouncementService.goToCustomerService( customer )` have the default compensating actions *retry* in case of a timeout failure and *rediscover* in case of a disconnection.

Again, first we show the implementation of this example application in NOW, followed by an implementation in AmbientTalk.

Listing 13 shows the implementation of this example application in our nomadic workflow language. As we already mentioned, NOW has built-in support for the automatic detection and handling of failures in its communication with mobile services. Hence, in the code in Listing 13 there are only 3 failure patterns used, namely for the ones that need more specific behaviour.

Listing 13. Implementation in NOW.

```
1  def exc := ExclusiveChoice( { |in_stock| in_stock },
2              Sequence( Shop.getLocations(articles)@Output(locs),
3                        Failure( Customer.updateMap(locs),
4                                 [ Disconnect(Wait(10, Rediscover())),
5                                   NotFound(Wait(60, Rediscover())) ]) ),
```

```
6                       Failure( Customer.notInStock(),
7                                [ Timeout(120, AnnouncementS.goToCustomerService(customer)),
8                                  NotFound(AnnouncementS.goToCustomerS(customer)) ] ) );
9
10  def parSplit := ParallelSplit(
11              Sequence( Stock.check(articles)@Output(in_stock), exc ),
12              Failure( Sequence( RecommendationS.get(articles)@Output(recommendations),
13                       Customer.recommend(recommendations) ),
14                       [ Timeout(120, Rediscover(1, Skip())),
15                         Disconnect(Skip()),
16                         NotFound(Skip()) ] ) );
17
18  def seq := Sequence( Customer.shoppingList()@Output(articles), parSplit);
```

The first pattern of this workflow is the *sequence* pattern which is defined on line 18 in the code in Listing 13. Although this *sequence* uses mobile service invocation, no *failure* pattern is used since the default compensating actions for timeout and disconnection failures are sufficient in this case. After this service invocation, the control flow is split over two branches by using a *parallel split* pattern. The second branch of the *parallel split* (line 12–16 in Listing 13) is a *sequence* which is being wrapped by a *failure* pattern since more specific compensating actions are needed. For instance, when a timeout occurs, the system needs to rediscover the service that caused the timeout two times before skipping the entire wrapped *sequence*. The first branch of the *parallel split* first executes the activity to check the availability of the articles and then either warns the customer that not all articles are available (lines 6–8), or updates a map on the users mobile phone (lines 3–5). Since in both cases communication with a mobile service is needed, and more specific behaviour for failures is modelled (see Fig. 7), two more *failure* patterns are used in the implementation.

The code snippets below show the implementation of part of this example application in AmbientTalk. The implementation only captures the wrapped *sequence* pattern that models the retrieval and sending of recommendations.

In order to improve the structure of the code, we implemented a `discoverService` function which discovers a service of a certain type tag and executes the next function that must be executed. In case a service of the given type tag cannot be found, another action is performed.

```
1  def discoverService(typeTag, next, notFound := { |_| _ }) {
2      when: typeTag discovered: { |service| next(service) };
3      when: seconds(120) elapsed: { notFound() };
4  };
```

This function takes two required arguments (the `typeTag` and an AmbientTalk block called `next`) and one optional argument (an AmbientTalk block called `notFound`). When a service is discovered, the AmbientTalk block is executed with this service (line 2). Otherwise the `notFound` block is executed (line 3).

We now describe the implementation of the wrapped *sequence* pattern in plain AmbientTalk. First a recommendation service needs to be discovered and, afterwards, we need to retrieve the list of recommendations of that service before sending them to the customer. The implementation is given in Listing 14.

Listing 14. Implementation in AmbientTalk.

```
1  def recommendToCustomer(customer, recommendations) {
2      def ctr := 0;
3
4      def recommend(customer) {
5         if: (ctr == 2) then: {
6             system.println("skipped");
7         } else: {
8            when: customer<-recommend(recommendations)@Due(seconds(120)) becomes: { |reply|
9                 reply;
10           } catch: TimeoutException using: { |e|
11                ctr := ctr + 1;
12                discoverService( Customer,
13                                 { |c| recommendToCustomer(c, recommendations) },
14                                 { |_| system.println("skipped") } ) };
15           whenever: customer disconnected: {
16                system.println("skipped") } } };
17
18     recommend(customer);
19 };
20
21 def getRecommendations(recommendationS, articles) {
22     def ctr := 0;
```

```
23
24      def retrieveList(recommendationS) {
25          if: (ctr == 2) then: {
26              system.println("skipped");
27          } else: {
28              when: recommendationS<-get(articles)@Due(seconds(120)) becomes: { |recs|
29                  discoverService( Customer,
30                      { |c| recommendToCustomer(c, recs)},
31                      { |_| system.println("skipped") });
32              } catch: TimeoutException using: { |e|
33                  ctr := ctr + 1;
34                  discoverService( RecommendationService,
35                      { |r| retrieveList(r) },
36                      { |_| system.println("skipped") } ) };
37              whenever: recommendationS disconnected: {
38                  system.println("skipped") } } };
39
40      retrieveList(recommendationS);
41  };
42
43  discoverService( RecommendationService,
44                   { |s| getRecommendations(s)},
45                   { |_| system.println("skipped")} );
```

When the last line of the code snippet in Listing 14 is executed, a service of type `RecommendationService` is being discovered (by using the function `discoverService` we described earlier). The second argument of that function call is the AmbientTalk block `{|s| getRecommendations(s)}`, whereas the `NotFound` argument is bound to the block `{|_| system.println( "skipped" )}` (since the compensating action for a service-not-found failure is skip).

The execution of the function `getRecommendations` results in the execution of the function `retrieveList` (line 40). Since there is a limit on the number of times the rediscover compensating actions may be executed (2 times in this scenario), we need to keep track of the number of times the service is being rediscovered (the variable `ctr` on line 22). In case we tried to rediscover the service already twice, the end of the application is reached (lines 25–26). Otherwise, we invoke the found service and when a result is retrieved in time, we send the recommendations to the customer (lines 29–31). In case a timeout is reached, we execute the specified compensating action and increase the variable `ctr` (lines 33–36).

The sending of the recommendations to the users is implemented by the `recommendToCustomer` function (lines 1–19). The implementation of this function is similar to the implementation of the `getRecommendations` function.

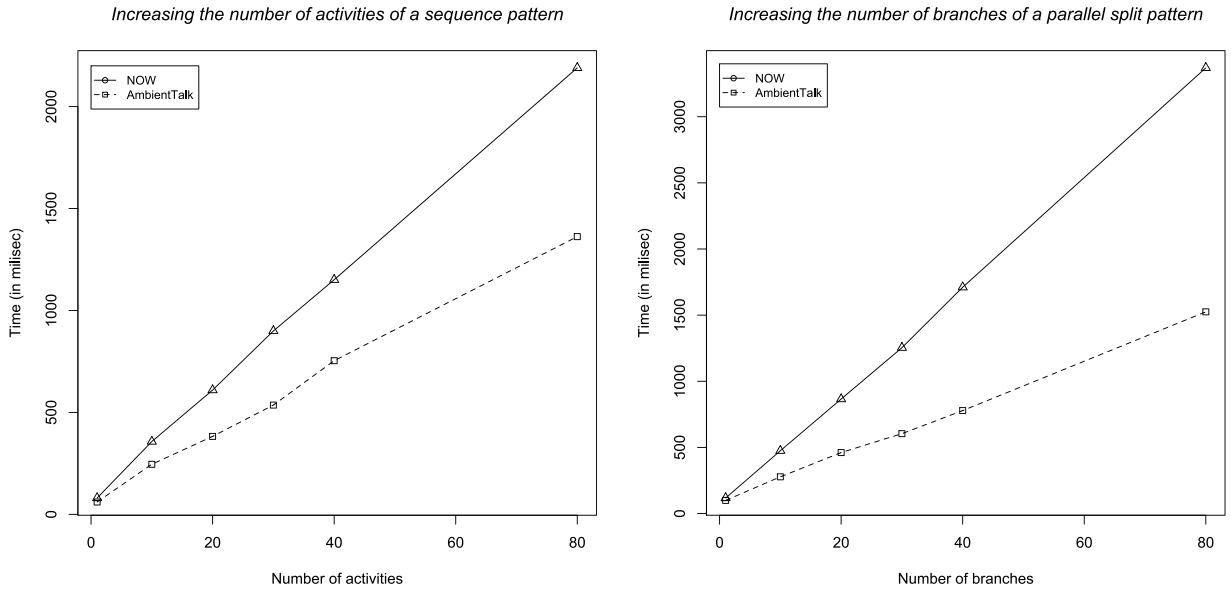
Again, we observe code duplication in the implementation that is hard to factor out in a clean way. Also note that since AmbientTalk has no support for default compensating actions in the communication with mobile services, more code (event handlers) is needed for the detection and handling of failures. Although this small part of the entire implementation is still readable, the nesting of the event handlers becomes 8 levels deep which makes the control flow of the application very hard to follow.

7.3. Scalability results

In this section we present the performance measurements of NOW compared to AmbientTalk. NOW is a proof-of-concept implementation as a library for AmbientTalk which we use to illustrate that the introduction of workflow abstractions make the code more understandable, reusable, and maintainable. When absolute performance is a necessity, NOW could be implemented in the AmbientTalk interpreter itself. At this moment, the scalability of our language is our main concern. In this section we show that the NOW library is scalable and also measure the overhead introduced by the workflow patterns.

We show the results of three different experiments. In the first experiment, presented in Section 7.3.1, we measure the overhead of the control flow patterns introduced by NOW. For this experiment we implement two basic control flow patterns frequently used in workflows, namely a sequence and parallel split, and compare the execution time of the implementation in NOW and AmbientTalk. In Section 7.3.2 we show the overhead introduced by failure detection in both NOW and AmbientTalk and compare the execution times of both implementations using the same two control flow patterns. As a third experiment, we compare the performance of implementations of the airport and IKEA application presented in Sections 7.1 and 7.2. Those applications use more control flow patterns, and in order to make the example more complex we wrap the application with a multiple instances pattern.

The experiments are executed on a MacBook with a 2.4 GHZ Intel Core 2 Duo processor and 4 GB of 1066 MHz DDR3 RAM. The used software includes OS X 10.6.6, JVM 1.6.0_22 and AmbientTalk 2.19.1. For each experiment two virtual machines are started (with a maximum heap size of 2 GB). A first virtual machine is used for the execution of the example application (in our experiments the sequence or parallel split pattern), whereas the services are executed by a second VM. Each experiment was executed 10 times and the average execution time was computed. Note that all activities used in the first experiments presented in Sections 7.3.1 and 7.3.2 have the same execution time.



(a) Execution time of a sequence pattern.

(b) Execution time of a parallel split pattern.

Fig. 8. Measurement of overhead introduced by patterns. Results for sequence pattern (a) and parallel split pattern (b).

7.3.1. Language scalability

We conducted two experiments: one where we increased the number of activities of a sequence and one where we augmented the number of branches of a parallel split pattern. We compared the execution time of these example applications in plain AmbientTalk and in NOW. The implementation in AmbientTalk behaves in the same way as the way the control flow patterns are defined. However, the implementation of those patterns in AmbientTalk uses a more direct style of programming using event handlers and hence does not have as many layers of abstraction as the implementation in NOW.

The first experiment we performed consists of increasing the number of activities in a sequence pattern and comparing the execution time of its implementation using plain AmbientTalk and using NOW. We measured the execution time of a sequence with 1, 10, 20, 30, 40 and 80 activities. The results of this experiment are shown in Fig. 8(a).

We can conclude that for each of the implementations there exists a linear correlation with the following coefficients:

	NOW	AmbientTalk
R^2	0.9993	0.9971
a	26.534	16.3902
b	80.700	62.2802

where R^2 is the squared correlation coefficient, and the linear model is defined as $ax + b$.

The linear correlation between the number of activities and the implementation in our workflow language shows that the implementation of NOW is scalable, as is the case for AmbientTalk as well. From the results we can also conclude that there is a small overhead when the NOW application is executed compared to the core AmbientTalk application. For instance, the execution time of a sequence of 80 activities is 2.2 s in our workflow language, whereas the execution takes 1.4 s when the implementation written in (plain) AmbientTalk is used. This overhead (an average factor of 1.62) is a result of, amongst other things, the management of the environment implementing the data flow through the workflow. The corresponding implementation in AmbientTalk uses local variables in event handlers which are assigned when a return value of a service is retrieved. Moreover, NOW introduces patterns and activities which are all implemented as AmbientTalk objects. So, the execution in our workflow languages uses more objects (for each activity, for the pattern itself, environment, ...), and there is a higher number of futures, used to chain patterns and activities together, that must be managed. We refer to Section 7.3.4 for a more elaborate discussion on the overhead introduced by NOW.

As a second experiment, we increase the number of branches of a parallel split pattern (where each branch consists of one activity which has identical execution time in all the branches). The measured application consists of a parallel split of 1, 10, 20, 30, 40, and 80 branches, followed by a synchronization pattern which merges all branches of the parallel split pattern. The different execution times of its implementation in NOW and AmbientTalk are shown in Fig. 8(b). Again, there is a linear correlation between the execution time and the number of branches. The table below shows the coefficients for each implementation.

	NOW	AmbientTalk
R^2	0.9995	0.9986
a	41.2915	17.854
b	52.9908	85.649

Executing a parallel split with 40 branches takes around 1.7 s using NOW's implementation, compared to 0.7 s when using AmbientTalk. The implementation in NOW is on average 2.32 times slower than the one in AmbientTalk. The overhead introduced by this workflow pattern is a result from the usage of more futures used to chain patters and activities together. Moreover, no optimizations for NOW have been implemented yet. We discuss this difference in execution time between NOW and AmbientTalk more in Section 7.3.4.

When we compare the coefficients of the regression test ($ax + b$) of the parallel split, with those of the sequence pattern obtained from the previous experiment, we see that the execution time of the sequence pattern is faster for the same number of activities. As we already mentioned, each experiment uses a separate virtual machine for the execution of the services of the application. For the sequence experiment, only one service is provided and this service is invoked sequentially as many times as the number of the pattern's activities. In order to perform the second experiment, we need several services (one for each branch of the parallel split pattern) in order to prevent a bottleneck in one single service. When a parallel split is being executed, all those services are invoked and can execute in parallel. However, the workflow, running in a separate virtual machine, cannot benefit from any parallelism because it is implemented as a single actor and the actor model specifically forbids inter-actor concurrency. Hence, in this experiment (where the execution time of a single activity is small) there is no real benefit of parallelism. When the execution time of the activities would be significantly larger, using a parallel split would be beneficial and the difference between the execution time of the two experiments would be the other way around.

The benchmarks of these two basic control flow patterns hint that our language implementation is scalable, as there is a linear correlation between the number of activities (branches) and the execution time of the application. In Section 7.3.3 we show the benchmarks of two more elaborate experiments that contain more patterns and use more advanced ones, such as multiple instances. The experiments we conducted for these two basic control flow patterns, sequence and parallel split, are executed using two VMs on the same machine, which gives us very small latencies. However, in real-life there will be much more latency involved as services are running on different machines connected by a network. The overhead that is introduced by NOW will be less significant in these real-life applications as the network delays are larger.

7.3.2. Scalability of language with failure detection

The experiments we conducted to test the scalability of the language implementation are repeated in order to test the scalability of languages with support for failure detection. In order to conduct these experiments we used the implementation of NOW with failure detection. So, for these experiments the `run` method shown in Listing 10 is used, whereas for the previous experiments the one of Listing 6 is used. This simulates the behaviour where each activity is wrapped by a failure pattern that can detect all 4 failures: timeouts, disconnections, exceptions and when a service is not found. In order to achieve the same behaviour when testing the implementation in AmbientTalk, we also need to detect those four failures. Instead of only using the `when: discovered:` and `when: becomes:` event handlers in AmbientTalk, we also install the event handlers that can catch exceptions (`catch:`), timeouts (by annotating the asynchronous message send with `@Due` and catching a `Timeout` exception), disconnections (`when: disconnected:`), and detect when a service is not found (`when: elapsed:`).

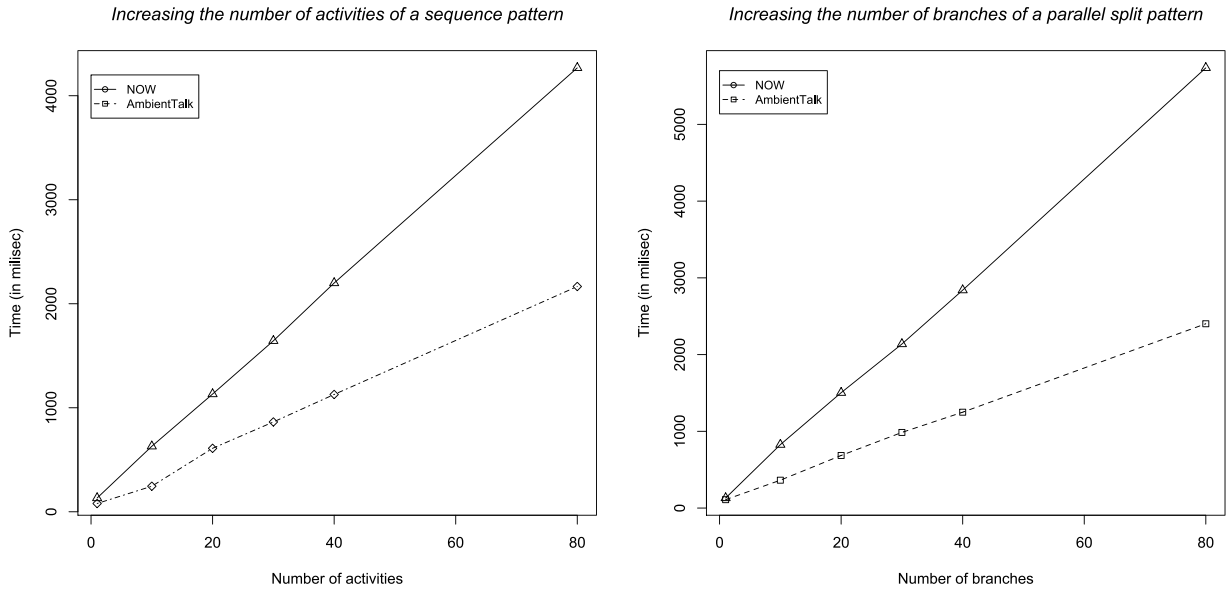
We repeat the experiment where the number of activities, each wrapped by a failure pattern, in a sequence pattern is increased. The results of this experiment are shown in Fig. 9(a).

We can derive that for each of the performance graphs of implementations with failure detection code there exists a linear correlation between the number of activities and the total runtime with the following coefficients:

	NOW	AmbientTalk
R^2	0.9999	0.9973
a	52.268	26.7466
b	90.639	41.8289

where R^2 is the squared correlation coefficient, and the linear model is defined as $ax + b$.

From the results obtained by this experiment, we can deduce that the implementation of a sequence pattern in NOW is slower with a factor of 1.95 than the corresponding code in AmbientTalk. For a discussion on the difference between the execution times of these implementations we refer to Section 7.3.4. Moreover, we can conclude that the execution time of an implementation is increased when introducing failure detection. This phenomenon occurs for applications written in AmbientTalk as well as in NOW. For instance, where previously a sequence of 80 activities had an execution time of 1.4 s in AmbientTalk, it now has a duration of 2.2 s. This factor is 18% higher than the difference in performance measured for the same experiment without failure detection (1.92 compared to 1.62 previously). Recall that for this experiment, each activity is wrapped by a failure pattern that can detect the four types of failures we support. For the corresponding implementation in AmbientTalk, this behaviour is achieved by adding four event handlers, for each service that must be discovered and



(a) Execution time of a sequence pattern.

(b) Execution time of a parallel split pattern.

Fig. 9. Measurement of overhead introduced by failure detection. Results for sequence pattern (a) and parallel split pattern (b).

invoked. These extra event handlers cause the difference in execution time with the previous experiment presented in Section 7.3.1.

We also measured the execution times of the second experiment where the number of branches of a parallel split is increased. The different execution times of those implementations are shown in Fig. 9(b). The table below shows the coefficients of the linear correlation for the implementations both in NOW and AmbientTalk.

	NOW	AmbientTalk
R^2	0.9994	0.9995
a	70.4633	29.0120
b	70.0916	90.5204

Here again we notice that the implementations with failure detection support are slower than the ones without. We can also derive that the execution time of a parallel split in NOW is on average 2.43 times larger than the execution time of the corresponding implementation in AmbientTalk. When comparing this factor with the one obtained from the parallel split-experiment without failure detection (presented in Section 7.3.1), we see that this factor is 4.7% higher (the factor of the previous experiment is 2.32).

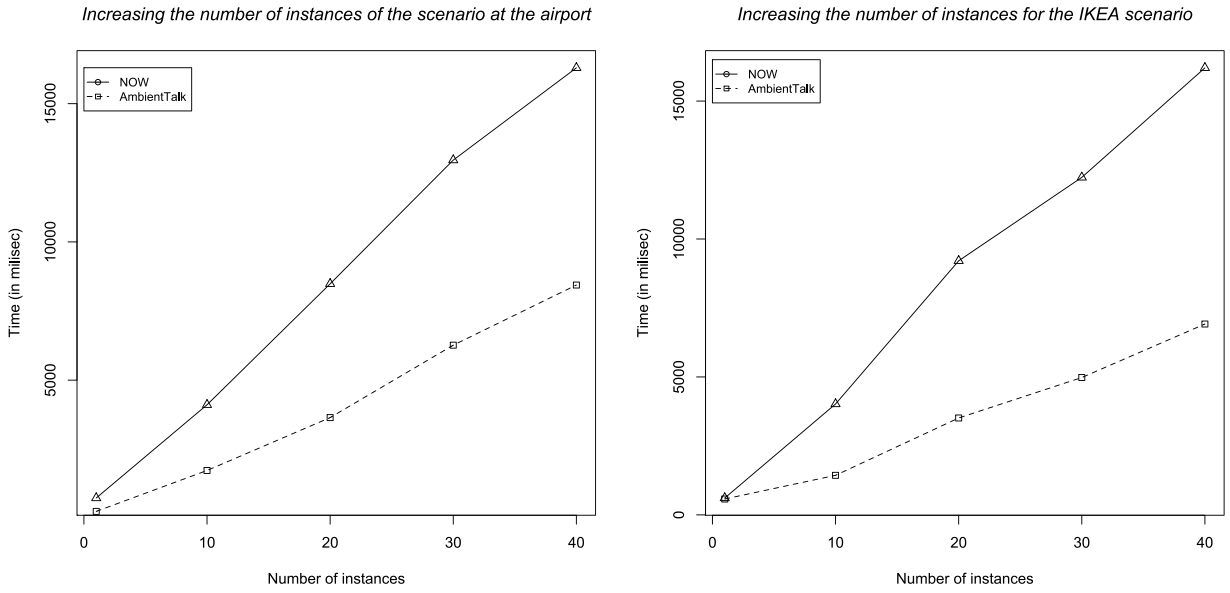
We showed that the introduction of failure detection increases the execution time with a factor (20% for a parallel split). However, we did not measure the overhead that is introduced by executing the actual compensating actions themselves for a certain failure. Executing performance tests for this experiment is meaningless, as the nature of a fault (timeout, disconnection, …) leads to delays orders of magnitudes larger than its compensation. For instance, the compensation of a timeout of 20 s can take a mere 20 ms. We can conclude from these benchmarks that our language implementation with failure detection is scalable for basic control flow patterns (sequence, parallel split), as there is a linear correlation between the number of activities (wrapped by a failure pattern) and the execution time of the application. In Section 7.3.3 we present benchmarks for applications using more, and more complex, workflow patterns.

7.3.3. Scalability of example scenarios

In this section we present benchmarks testing the overhead introduced by NOW measured using the two example applications whose implementations are described in Sections 7.1 and 7.2. The airport example consists of 12 control flow patterns (sequences, parallel splits, synchronizes, connections, exclusive choice patterns, and a structured discriminator). We also want to measure the overhead of more complex control flow patterns, hence, we wrap this workflow with a multiple instances pattern (called *multiple instances with a priori design-time knowledge*). We repeat the experiment by increasing the number of multiple instances. The results are shown in Fig. 10(a).

The second application, the scenario at IKEA, has 9 patterns (failure patterns, sequences, parallel split, and an exclusive choice). We also wrap this workflow with a multiple instances pattern and repeat the experiment with increasing the number of instances. The results of this experiment are shown in Fig. 10(b).

For both experiments, we can conclude that for the implementations in NOW and AmbientTalk there exists a linear correlation with the following coefficients:



(a) Execution time of airport scenario.

(b) Execution time of scenario at IKEA.

Fig. 10. Measuring the addition of workflow patterns compared to plain AmbientTalk. Results are shown for a multiple instances pattern wrapping the airport example in figure (a) and for the example at IKEA in figure (b).

Coefficients for scenario at airport

	NOW	AmbientTalk
R^2	0.9971	0.9716
a	407.86	192.12
b	278.01	333.19

Coefficients for scenario at IKEA

	NOW	AmbientTalk
R^2	0.9924	0.9876
a	401.92	165.913
b	336.32	131.863

We can derive that for the airport application the implementation in NOW is 2.12 times slower than the one in AmbientTalk. For the application at IKEA this factor is 2.42. The fact that the second application is slower than the first one can be explained as the application at IKEA uses failure detection. In Section 7.3.2 we already mentioned this difference when comparing the results of the two basic control flow patterns with and without failure detection. We refer to the Section 7.3.4 for a more elaborate discussion on the difference in performance between an implementation in NOW and its corresponding one in AmbientTalk.

In this experiment we show that for more complex applications, using several control flow patterns (even a more complex multiple instances pattern), the implementation of NOW is scalable. We can also conclude that our workflow patterns introduce overhead, but still lead to scalable applications.

7.3.4. Discussion

In this section we first discuss the difference in execution time between an application in our nomadic workflow language and its corresponding implementation in AmbientTalk. Thereafter, we describe the threats to validity of our scalability results.

The three experiments we presented each show that NOW's implementation is slower than the one in plain AmbientTalk. First of all, we want to stress that NOW is a proof-of-concept implementation that is implemented as a library for AmbientTalk. The goal of this proof-of-concept implementation is to show that languages can benefit from the extra abstraction layer of workflow patterns, such that the code becomes more readable, reusable and maintainable. When the language has reached full maturity, it could be integrated in the AmbientTalk interpreter itself where more optimizations can be made. The goal of the conducted experiments is to show that the library is scalable.

The difference in execution time between the implementations in NOW and AmbientTalk can be explained, as our workflow language introduces more futures and objects. For instance, instead of just assigning local variables (done in the AmbientTalk implementation of the experiments), the implementation in NOW uses an environment object which requires extra provisions in order to allow the data to flow through the entire workflow. Moreover, the workflow language introduces more objects (for each activity, pattern, …), which each create a new future when started. Hence, more futures must be managed by NOW. This extra complexity causes the extra overhead we showed in our experiments.

The benchmarks shown measuring the execution time of two basic control flow patterns without failure detection (in Section 7.3.1) show that the implementation in NOW is slower than their corresponding implementation in AmbientTalk.

To determine whether our results are generalizable to other workflow patterns, more experiments are needed. We need to extend our experiments to other kinds of applications as well.

As a third experiment we presented the results of two more complex applications that consist of several workflow patterns. From these experiments we could deduce that the implementation of our workflow language is scalable for these applications that use more complex control flow patterns. However, more stress tests on complex applications are needed to make our conclusions generalizable.

8. Related work

Coordination languages

Workflow languages targeting web services, such as Bite [18] and Mobile Business Processes [19], can only operate on services that are known beforehand by means of a fixed URL. CiAN [3] is a workflow language for mobile ad hoc networks. But, since it has a choreographed architecture, the responsibilities are divided a priori by an allocation algorithm. This is not suitable in a dynamically changing environment, where services can join and disjoin at any moment in time. Another workflow system, Workpad [5], is designed with nomadic networks in mind, but also assumes that devices are connected upon startup. Open Workflow [4] is a system that targets workflow construction, allocation, and execution for mobile devices in mobile ad hoc networks. This system focuses on the dynamic construction of workflows based upon contextual information, something we do not target.

There also exist orchestration languages, such as Orc [20,21], which uses a process calculus to express the coordination of different processes. Chromatic Orc [22] extends this calculus with exception handling by introducing throw and try catch expressions that can both run in parallel and hence do not cause termination. However, this language assumes a stable network interconnecting the services. Another orchestration language is Reo [23], 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.

Failure detection and handling

There exists related work concerning the handling of both expected and unexpected exceptions in workflow systems. The exception patterns proposed by Russell [16] are already available in workflow systems and business process modelling languages. The exception patterns Russell proposes (work item failure, work item deadline, external trigger and constraining violation) are all supported by, for instance, BPMN. BPEL [2] offers the concept of a fault handler for handling exception. A fault handler can be attached to a process, a scope or as a shorthand notation on an invoke-activity and it is installed when its associated scope is started. YAWL [24] has support for the proposed exception patterns and for dynamic workflows as they need to be able to change when handling unexpected exceptions. The failures we support are specific to (temporary) network failures inherent to a mobile network (mobile ad hoc or nomadic). More basic exception handling can be achieved in NOW by specifying compensating actions for the exception failure.

Research to handling of expected failures has led to the development of transactional workflows [25–27]. The ConTracts [28] project proposes a coordinated transaction model that allows both forward, backward and partial recovery when a failure event is signalled. OPERA [29] incorporates language primitives for exception handling and their strategies into a workflow system. OPERA introduces the notion of spheres that are used to categorize operations as units with transactional properties. Another way to specify exception handling strategies is by using event-condition-action (ECA) rules, as seen in, for example, the exception language Chimera-Exc [30]. The types of events that are defined by Chimera-Exc are data manipulation events, temporal events, external events, and workflow events. The actions supported by this exception language are either data modification primitives (for instance, to modify the value of an object's attribute) or workflow management primitives (which can, for example, start a new task). Since maintaining transactions in a mobile network where participants can disconnect indefinitely, rolling back transactions, etc. can be problematic and we believe that this approach is not ideally suited for the environment we are targeting.

Adaptive workflow research and workflow evolution research focus on the detection and handling of unexpected failures introduced due to the dynamic changing of the workflow. ADOME-WFMS [31] is an example of a project that provides both support for expected and unexpected failures where exception handlers are specified using ECA rules. The compensations they support (like skipping, or repeating a task) are also provided by NOW.

Failure handling or recovery concepts are proposed by Eder [32]. That paper categorizes two recovery techniques within workflow execution, namely automatic and manual tasks. Automatic tasks are further divided into restarting the same task, starting an alternative task and manual intervention. Recovery for a manual task must be performed by the workflow participant responsible for that particular task. The several compensations supported by NOW cover the automatic tasks proposed by Eder. Recovery via manual tasks can be achieved in NOW by specifying a component (subworkflow or activity)

as the compensating action of a failure, where that component implements the human intervention that is needed, for instance, by implementing a service that asks for user input.

The coordination language Reo [23] is already applied in areas like business process modelling [33] and web service composition [34]. Reo supports patterns to compose sub-processes and exception handling is possible by using so called routers (which can interrupt a process) to propagate cancel messages. Each sub-processes can be interrupted by such a cancel message or an internal exception. Moreover, there exist tools that can translate BPMN and UML into Reo [33].

9. Conclusion and future work

In this paper, we have presented the design and implementation of a nomadic workflow language on top of a runtime system that allows the orchestration of distributed services in a nomadic network, thanks to a dynamic peer-to-peer service discovery mechanism and communication primitives resilient to the volatile connections inherent to such networks. NOW provides high level patterns for control flow and mechanisms for detecting and handling both service and network failures that are inherent to a nomadic network. The language also supports a dynamic mechanism for variable bindings such that data can be passed between the services of the nomadic workflow.

We identified a number of key items as future work: First of all, we would like to support *group interaction*. Group interaction is useful for a dynamically changing environment where the number of communication partners (services or entities) is not known beforehand and can vary over time. Another useful extension would be supporting *intensional descriptions* of communication partners, as the changing network topology complicates extensional reasoning over these partners. The addition of these intentional descriptions also gives rise to the need for compensating actions when constraints are violated or group compositions are altered. We want to enable writing constraints for groups of services (and entities) by writing logical rules in Crime [35], a logic-based coordination language we have developed. Furthermore, support for some more advanced synchronization patterns is necessary because complete synchronization will not always be possible in environments where communication partners in a group can go out of range at any point in time. We want to design synchronization patterns for group interaction that will only succeed when certain conditions are met, for instance, when a certain percentage of communication partners has replied. Finally, a graphical input language, used in this paper, is being developed along with the textual programming language. Tool support for this graphical language is envisioned in the future. When the language NOW has reached full maturity, the next step can be taken to integrate the language into the AmbientTalk core instead of providing a library.

Acknowledgements

We would like to acknowledge Jorge Vallejos for his invaluable comments and suggestions. Eline Philips is funded by a doctoral scholarship of the "Institute for the Promotion of Innovation through Science and Technology in Flanders" (IWT Vlaanderen).

References

[1] C. Mascolo, L. Capra, W. Emmerich, Mobile computing middleware, in: E. Gregori, G. Anastasi, S. Basagni (Eds.), Advanced Lectures on Networking, NETWORKING 2002, in: Lecture Notes in Computer Science, vol. 2497, Springer, 2002, pp. 20–58.
[2] D. Jordan, J. Evdemon, et al., Web Services Business Process Execution Language, version 2.0, 2007. http://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.html.
[3] R. Sen, G.-C. Roman, C.D. Gill, Cian: a workflow engine for manets, in: D. Lea, G. Zavattaro (Eds.), Coordination Models and Languages, 10th International Conference, COORDINATION 2008, Oslo, Norway, June 4–6, 2008. Proceedings, in: Lecture Notes in Computer Science, vol. 5052, Springer, 2008, pp. 280–295.
[4] L. Thomas, J. Wilson, G.-C. Roman, C. Gill, Achieving coordination through dynamic construction of open workflows, in: Proceedings of the 10th ACM/IFIP/USENIX International Conference on Middleware, Middleware'09, Springer-Verlag New York, Inc., New York, NY, USA, 2009, 14:1–14:20.
[5] M. Mecella, M. Angelaccio, A. Krek, T. Catarci, B. Buttarazzi, S. Dustdar, Workpad: an adaptive peer-to-peer software infrastructure for supporting collaborative work of human operators in emergency/disaster scenarios, in: Proceedings of the International Symposium on Collaborative Technologies and Systems, IEEE Computer Society, Washington, DC, USA, 2006, pp. 173–180.
[6] E. Philips, R.V.D. Straeten, V. Jonckers, Now: a workflow language for orchestration in nomadic networks, in: D. Clarke, G.A. Agha (Eds.), Coordination Models and Languages, 12th International Conference, COORDINATION 2010, Amsterdam, The Netherlands, June 7–9, 2010. Proceedings, in: Lecture Notes in Computer Science, vol. 6116, Springer, 2010, pp. 31–45.
[7] T.V. Cutsem, S. Mostinckx, E.G. Boix, J. Dedecker, W.D. Meuter, Ambienttalk: object-oriented event-driven programming in mobile ad hoc networks, in: XXVI International Conference of the Chilean Computer Science Society (SCCC 2007), 8–9 November 2007, Iquique, Chile, IEEE Computer Society, 2007, pp. 3–12.
[8] T.V. Cutsem, S. Mostinckx, W.D. Meuter, Linguistic symbiosis between event loop actors and threads, Computer Languages, Systems & Structures 35 (2009) 80–98.
[9] Object Management Group, Business Process Model and Notation, version 2.0, 2011. http://www.omg.org/spec/BPMN/2.0/.
[10] N. Russell, A.H.M. ter Hofstede, W.M.P. van der Aalst, N. Mulyar, Workflow Control-Flow Patterns: A Revised View, BPM Center Report BPM-06-22, BPM Center, 2006. http://www.workflowpatterns.com/documentation/documents/BPM-06-22.pdf.
[11] J. Dedecker, T.V. Cutsem, S. Mostinckx, T. D'Hondt, W.D. Meuter, Ambient-oriented programming, in: R.E. Johnson, R.P. Gabriel (Eds.), Companion to the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2005, October 16–20, 2005, San Diego, CA, USA, ACM, 2005, pp. 31–40.
[12] G. Agha, Actors: A Model of Concurrent Computation in Distributed Systems, MIT Press, Cambridge, MA, USA, 1986.
[13] B. Chin, T.D. Millstein, Responders: language support for interactive applications, in: D. Thomas (Ed.), ECOOP 2006 - Object-Oriented Programming, 20th European Conference, Nantes, France, July 3–7, 2006, Proceedings, in: Lecture Notes in Computer Science, vol. 4067, Springer, 2006, pp. 255–278.

[14] E. Philips, Website now, http://soft.vub.ac.be/~ephilips/NOW, 2011.
[15] S. Sadiq, M. Orlowska, W. Sadiq, C. Foulger, Data flow and validation in workflow modelling, in: Proceedings of the 15th Australasian Database Conference - Volume 27, ADC'04, Australian Computer Society, Inc., Darlinghurst, Australia, Australia, 2004, pp. 207–214.
[16] N. Russell, A.H.M. ter Hofstede, D. Edmond, W.M.P. van der Aalst, Workflow data patterns: identification, representation and tool support, in: L.M.L. Delcambre, C. Kop, H.C. Mayr, J. Mylopoulos, O. Pastor (Eds.), Conceptual Modeling - ER 2005, 24th International Conference on Conceptual Modeling, Klagenfurt, Austria, October 24–28, 2005, Proceedings, in: Lecture Notes in Computer Science, vol. 3716, Springer, 2005, pp. 353–368.
[17] P. Haller, M. Odersky, Event-based programming without inversion of control, in: D.E. Lightfoot, C.A. Szyperski (Eds.), Modular Programming Languages, 7th Joint Modular Languages Conference, JMLC 2006, Oxford, UK, September 13–15, 2006, Proceedings, in: Lecture Notes in Computer Science, vol. 4228, Springer, 2006, pp. 4–22.
[18] F. Curbera, M.J. Duftler, R. Khalaf, D. Lovell, Bite: workflow composition for the web, in: B.J. Krämer, K.-J. Lin, P. Narasimhan (Eds.), Service-Oriented Computing - ICSOC 2007, Fifth International Conference, Vienna, Austria, September 17–20, 2007, Proceedings, in: Lecture Notes in Computer Science, vol. 4749, Springer, 2007, pp. 94–106.
[19] L. Pajunen, S. Chande, Developing workflow engine for mobile devices, in: 11th IEEE International Enterprise Distributed Object Computing Conference, EDOC 2007, 15–19 October 2007, Annapolis, Maryland, USA, IEEE Computer Society, 2007, pp. 279–286.
[20] D. Kitchin, A. Quark, W.R. Cook, J. Misra, The Orc programming language, in: D. Lee, A. Lopes, A. Poetzsch-Heffter (Eds.), Formal Techniques for Distributed Systems, Joint 11th IFIP WG 6.1 International Conference FMOODS 2009 and 29th IFIP WG 6.1 International Conference FORTE 2009, Lisboa, Portugal, June 9–12, 2009. Proceedings, in: Lecture Notes in Computer Science, vol. 5522, Springer, 2009, pp. 1–25.
[21] W.R. Cook, S. Patwardhan, J. Misra, Workflow patterns in Orc, in: P. Ciancarini, H. Wiklicky (Eds.), Coordination Models and Languages, 8th International Conference, COORDINATION 2006, Bologna, Italy, June 14–16, 2006, Proceedings, in: Lecture Notes in Computer Science, vol. 4038, Springer, 2006, pp. 82–96.
[22] A. Matsuoka, D. Kitchin, A semantics for exception handling in Orc, 2009.
[23] F. Arbab, Reo: a channel-based coordination model for component composition, Mathematical Structures in Computer Science 14 (2004) 329–366.
[24] A.H.M.T. Hofstede, Yawl: yet another workflow language, Information Systems 30 (2005) 245–275.
[25] R. Stiphout, T. Meijler, A. Aerts, D. Hammer, R. Comte, TREX: Workflow TRansactions by Means of EXceptions, Technical Report, Eindhoven University of Technology, 1997.
[26] A.P. Sheth, M. Rusinkiewicz, On transactional workflows, IEEE Data Engineering Bulletin 16 (1993) 37–40.
[27] G. Alonso, D. Agrawal, A.E. Abbadi, M. Kamath, R. Günthör, C. Mohan, Advanced transaction models in workflow contexts, in: Proceedings of the Twelfth International Conference on Data Engineering, ICDE'96, IEEE Computer Society, Washington, DC, USA, 1996, pp. 574–581.
[28] A. Reuter, F. Schwenkreis, Contracts — a low-level mechanism for building general-purpose workflow management-systems, IEEE Data Engineering Bulletin 18 (1995) 4–10.
[29] C. Hagen, G. Alonso, Flexible exception handling in the opera process support system, in: Proceedings of the The 18th International Conference on Distributed Computing Systems, ICDCS'98, IEEE Computer Society, Washington, DC, USA, 1998, pp. 526–533.
[30] F. Casati, S. Ceri, S. Paraboschi, G. Pozzi, Specification and implementation of exceptions in workflow management systems, ACM Transactions on Database Systems 24 (1999) 405–451.
[31] D.K.W. Chiu, Q. Li, K. Karlapalem, ADOME-WFMS: Towards Cooperative Handling of Workflow Exceptions, Springer-Verlag New York, Inc., New York, NY, USA, pp. 271–288.
[32] J. Eder, W. Liebhart, Workflow recovery, in: IFCIS Conference on Cooperative Information Systems, Society Press, 1996, pp. 124–134.
[33] F. Arbab, N. Kokash, S. Meng, Towards using reo for compliance-aware business process modeling, in: T. Margaria, B. Steffen (Eds.), Leveraging Applications of Formal Methods, Verification and Validation, Third International Symposium, ISoLA 2008, in: Communications in Computer and Information Science, vol. 17, Springer, 2008, pp. 108–123.
[34] S. Meng, F. Arbab, Web services choreography and orchestration in reo and constraint automata, in: Proceedings of the 2007 ACM Symposium on Applied Computing, SAC'07, ACM, New York, NY, USA, 2007, pp. 346–353.
[35] S. Mostinckx, C. Scholliers, E. Philips, C. Herzeel, W.D. Meuter, Fact spaces: coordination in the face of disconnection, in: A.L. Murphy, J. Vitek (Eds.), Coordination Models and Languages, 9th International Conference, COORDINATION 2007, Paphos, Cyprus, June 6–8, 2007, Proceedings, in: Lecture Notes in Computer Science, vol. 4467, Springer, 2007, pp. 268–285.