NOW: A Workflow Language for Orchestration in Nomadic Networks

Eline Philips*, Ragnhild Van Der Straeten, and Viviane Jonckers

Software Languages Lab, Vrije Universiteit Brussel, Belgium
{ephilips,rvdstrae}@vub.ac.be, vejoncke@soft.vub.ac.be

Abstract. Existing workflow languages for nomadic or mobile ad hoc networks do not offer adequate support for dealing with the volatile connections inherent to these environments. Services residing on mobile devices are exposed to (temporary) network failures, which should be considered the rule rather than the exception. This paper proposes a nomadic workflow language built on top of an ambient-oriented programming language which supports dynamic service discovery and communication primitives resilient to network failures. Our proposed language provides high level workflow abstractions for control flow and supports rich network and service failure detection and handling through compensating actions. Moreover, we introduce a powerful variable binding mechanism which enables dynamic data flow between services in a nomadic environment. By adding this extra layer of abstraction on top of an ambient-oriented programming language, 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 which 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 as these nomadic environments consist of both a group of mobile devices and a fixed infrastructure [2]. As these kind 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 by using technologies such as service-oriented computing. The composition of services can be achieved by 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

* Funded by a doctoral scholarship of the "Institute for the Promotion of Innovation through Science and Technology in Flanders" (IWT Vlaanderen).
[1] Not to be confused with the term *nomadic computing*.

languages such as (WS)BPEL[9] are suited to orchestrate (web)services, they are not suited for nomadic networks where services are not necessarily known a priori. Furthermore, 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 [5][8] and nomadic networks [3] are also 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 which happen frequently.

In this paper, we introduce a nomadic workflow language NOW which features the basic control flow patterns [4] sufficient for most workflows and dynamic service discovery. These patterns and services can be robustly deployed in a nomadic network and support complex (network) failure handling through compensating actions. Moreover, the language enables passing data between services using a dynamic variable binding mechanism.

This paper is organized as follows: we first introduce a motivating example which specifies the requirements a nomadic workflow language must fulfill. Afterwards, we present AmbientTalk, an ambient-oriented programming language which is developed at the Software Languages Lab and upon which we build our language. In section 4 we present the concepts of control flow, dynamic data passing mechanism and compensating actions before describing the implementation of a concrete workflow pattern. Finally we discuss related work and conclude with our contributions and future work.

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 delineating the workflow description of this application and afterwards highlight the requirements our workflow language must fulfill.

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 makes a transit at the airport of Frankfurt. Ten minutes before boarding he is still stuck in a traffic jam. At the airport, Peter is announced as a missing passenger and a flight assistant is informed to start looking for him. Peter also receives a reminder on his smart phone. After 10 minutes, the boarding responsible closes the gates and informs Avia Partners (the company that takes care of the luggage) to remove Peter's suitcase from the plane. He also ensures that the airport of Frankfurt is notified of the free seat so a last minute offer from Frankfurt to Newark becomes available. Peter gets notified he can return home and catch another flight later.

This example clearly introduces some of the concepts of a nomadic system. First of all, we can identify different kinds of participants in this scenario: mobile

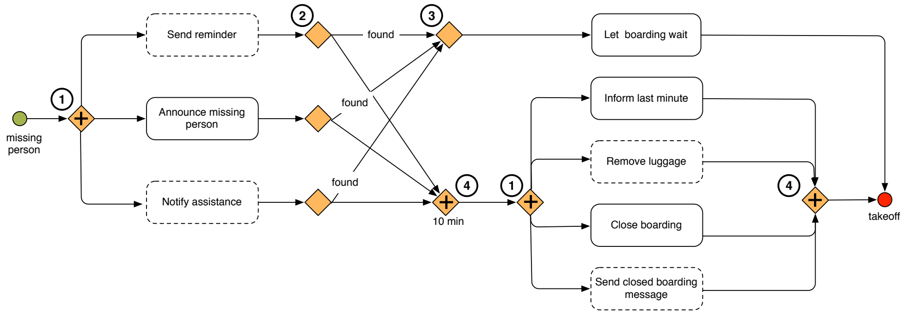

Fig. 1. Workflow description of the motivating example

devices, mobile services and stationary services. Mobile devices represent the 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 (like a departure screen or check-in desk) are also part of the infrastructure, but use a more reliable connection.

Figure 1 gives a representation of this example where BPMN notation is used for the patterns involved. The circles in the picture represent *events*: in the beginning of the workflow there's a general event and at the end there's a termination event. Note that the numbered circles are not part of the syntax, they are used later on for pointing to specific elements of the description. Rounded rectangles represent *activities* which are a generic type of work that can be performed by either mobile (dashed line) or stationary services (solid line). The figure also contains several diamonds which depict *gateways* that can merge or split the control flow. There are two different kind of gateways used in this workflow description: the diamond with a + sign represents a logic *and*, whereas the empty one symbolizes a *xor*.

The different control flow patterns [4] which are needed to describe the scenario as a workflow are (labeled in figure 1 with their corresponding numbers):

1. *parallel split*: execution of two or more branches in parallel [4].
2. *exclusive choice*: divergence of a branch into two or more branches, such that the thread of control is passed to one subsequent branch. This branch is chosen based on a selection mechanism [4].
3. *structured discriminator*: convergence of two or more incoming branches into one subsequent branch, such that the thread of control is passed to the subsequent branch the first time an incoming branch is enabled [4].
4. *synchronize*: convergence of two or more branches into one subsequent branch, such that the thread of control is passed to the subsequent branch when all incoming branches have been enabled [4].

2.2 Requirements

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

1. support two kind of services: stationary and mobile.
2. automatic handling of failures in communication with mobile services (for example transparently rediscover a service of the same type).
3. support basic control flow patterns.
4. detection of failures (such as disconnection and timeout) and ability to specify compensating actions for these detected failures.
5. support data flow such that information can be passed between services.

It is important to note that the description and execution of the workflow resides on the backbone of the nomadic system, whereas the different services are either located on fixed devices or on mobile devices that move around through the environment. By situating the workflow description on the fixed infrastructure, we ensure that the workflow itself cannot disconnect and become unavailable during its execution 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 with a mobile ad hoc network.

3 AmbientTalk

In this section, we briefly explain the programming language AmbientTalk [6][7], which we use to build our nomadic workflow language on top of. The *ambient-oriented* programming paradigm [1] is specifically aimed at such applications. We highlight only the important features which we will need for the rest of this paper.

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

– In a mobile network (mobile ad hoc or nomadic network), 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 an mobile network (mobile ad hoc or nomadic network), 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 and resent later, when the object reconnects.

3.1 Distributed Programming in AmbientTalk

Listing 1 shows how to create an AmbientTalk object with a method `missingPerson`. This object is exported with the *type tag* `Announcement Service`, which means that from now on this object can be discovered using this tag.

Listing 1. Implementation of an announcement service in AmbientTalk.

```
deftype AnnouncementService;

def service := object: {
    def missingPerson(person, time) {
        def found := false;
        // Announce missing person
        // If person turns up, change value of "found" variable
        found; // Return the value of found }};
export: service as: AnnouncementService;
```

AmbientTalk uses a classic event-handling style by relying on blocks of code that are triggered by event handlers. Event handlers are (by convention) registered by a call to a function that starts with `when:`. So, when you want to discover an annoucement service and call its method `missingPerson`, you should write the following piece of AmbientTalk code:

```
// When a service classified as AnnouncementService is discovered,
// this object is accessible via "service"
when: AnnouncementService discovered: { |service|
    // Send asynchronous message "missingPerson" to the discovered object
    when: service<-missingPerson("Peter", 10) becomes: { |reply|
        // When a reply is received and it equals true, the person is found
        // and boarding should wait. Otherwise, boarding can be closed
        // and the luggage can be removed }};
```

The syntax `obj<-msg()` denotes an *asynchronous message sent* which 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.

As can be seen from the above example, service discovery and replies of remote queries are represented in AmbientTalk as events that trigger the appropriate event handlers. While in this simple example the control flow remains apparent enough to understand, the control flow of large-scale event-driven applications can quickly become puzzling. In the following sections we discuss how to add a 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.

4 Nomadic Workflow Patterns

This section describes the control flow patterns in our workflow language NOW, the support for data flow and compensating actions.[2] Afterwards, we present the concrete implementation of a pattern, *synchronize*, and show in detail the concepts we introduced.

4.1 Control Flow Patterns

In this section we describe the implementation of workflow patterns of van der Aalst [4] on top of AmbientTalk (requirement 3 in section 2.2). The current

[2] The implementation is available at `http://code.google.com/p/ambienttalk/`

implementation consists of 13 control flow patterns: sequence, parallel split, synchronize, exclusive choice, simple merge, multi choice, structured synchronizing merge, multi merge, structured discriminator, structured partial join, multiple instances without synchronization, static partial join for multiple instances and implicit termination. We first show the syntax of these patterns and afterwards describe how these patterns can be composed.

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 the control flow patterns.

```
<components>          := <component> | <component> “,” <components>
<component>           := <activity> | <pattern>
<ATblock>             := “{  |”  <parameter list>  “|”  <body> “}”
<condition action>    :=  “[”  <ATblock> “,”  <component> “]”
<condition actions>   := <condition action> | <condition action> “,”  <condition actions>
<syncpattern>         := <synchronize> | <structured synchronizing merge> |
                         <simple merge> | <multi merge> | <structured partial join>
<pattern>             := <syncpattern> | <sequence> | <parallel split> | <multi choice> |
                         <exclusive choice>  |  <structured discriminator> |  <connection> |
                         <multiple instances without synchronization> |
                         <static  partial  join  for  multiple  instances>
<sequence>            := “Sequence(”  <components> “)”
<parallel  split >    := “ ParallelSplit (”  <components> “)”
<synchronize>         := “Synchronize(”  <component> “)”
<exclusive choice>    := “ExclusiveChoice(”  <condition actions>  “)”
<simple merge>        := “SimpleMerge(” component “)”
<multi choice>        := “MultiChoice(”  <condition actions>  “)”
<multi merge>         := “MultiMerge(” <component> “)”
<connection>          := “Connection(”  <syncpattern> “)”
<structured discriminator>            := “StructuredDiscriminator(”  <component> “)”
<structured partial  join>            := “StructuredPartialJoin (”  <component> “)”
<structured synchronizing merge>      := “StructuredSynchronizingMerge(” <component> “)”
<multiple instances without synchronization>    := “MIWithoutSync(” <component> “)”
<static  partial  join  for  multiple  instances>    := “StPartJoinMI(” <component> “)”
```

Patterns are implemented as AmbientTalk objects which always implement the `init` method (constructor) and `start` method which starts the execution of the workflow.

Composition of Control Flow Patterns. As we can derive from the above syntax, each pattern consists of several components which can be either activities or patterns themselves. Executing an activity results in the invocation of a service which is implemented as a distributed object in AmbientTalk. Recapitulate from section 3.1, this invocation is executed by sending an asynchronous message to the distributed object. The result of this invocation is a future, and the installed event handlers are triggered when this future is resolved. In order to compose patterns in a flexible way, patterns must be oblivious to the difference between their components, which can either be activities or patterns themselves. Hence, the execution of a pattern must also return a future so the necessary event handlers can be installed to wait for its termination. Consider two sequence patterns in the code snippet below. The execution of the first sequence, `seq1.start()`, returns a future which resolves when `seq1` is fully executed. The second line in the code shows a simple example of a composition, namely a sequence `seq1` which is part of a bigger sequence `seq2`.

```
def seq1 := Sequence( serviceB.b(), serviceC.c() );
def seq2 := Sequence( serviceA.a(), seq1, serviceD.d() );
```

However, composition of patterns becomes more complex when patterns like synchronization are involved. In listing 2 we see that some patterns like synchronization or simple merge are considered `syncpatterns`. This distinction is made because patterns with multiple incoming branches need to be composed using `connections`.

Consider such an example in figure 2 where first a parallel split (marked with circle 1) diverges the control flow into three outgoing branches, and afterwards a synchronization (marked with circle 2) converges these branches.

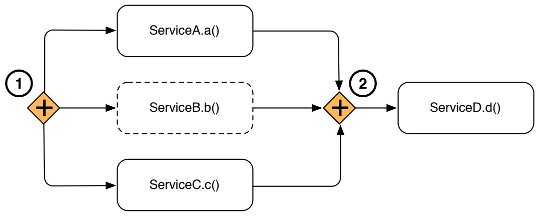

Fig. 2. Control flow patterns: parallel split followed by synchronize

This example can be written in NOW as shown in listing 3. The first line in the code defines a synchronization pattern which must invoke a method `d()` on `ServiceD` after all its incoming branches are enabled.

Listing 3. Parallel split follwed by synchronize.

```
def sync := Synchronize( serviceD.d() );
def parSplit := ParallelSplit( [ serviceA.a(), sync ],
                               [ serviceB.b(), sync ],
                               [ serviceC.c(), sync ] );
```

By introducing a `Connection`, a link is made between the outgoing branches of the parallel split and the incoming ones of the synchronization. This connection pattern informs the synchronize `sync` of how many incoming branches it has. The parallel split which is defined on the second line is initialized with three tables (one for each branch), containing the components of that branch. Recall from figure 1 that it is not required that each outgoing branch of a parallel split is connected to the same synchronization pattern (thus the need to introduce connections). The first branch of the parallel split in the example (on line 2 in the code snippet) is converted into `Sequence.new( serviceA.a(), Connection.new(sync) )`.

4.2 Data Flow in Nomadic Networks

In this section we describe a data flow mechanism which enables us to pass information between (possibly non-consecutive) activities (requirement 5 in section 2.2). We present the mechanism used for parameter bindings when invoking a service (execute an activity). The formal syntax for an activity is given below:

Listing 4. Abstract grammar of activities.

```
<activity>   := <service> "." <method> "(" <arguments> ")" |
                <service> "." <method> "(" <arguments> ")@output(" <parameters> ")"
<parameters> := <parameter> | <parameter> "," <parameters> | <void>
<arguments>  := <expression> | <expression> "," <arguments> | <void>
<parameter>  := <symbol>
```

When an activity is executed, its formal parameters are bound to their values which are looked up in the *environment* before the method is invoked at the service. The environment is a simple dictionary associating variables with values. The invoked method at the service can return a number of result values which are bound to the correct variables using the `@output` syntax, as shown in the example below. Returning a wrong number of values or parameters which are not found in the current environment results in an error.

```
TemperatureService.getTemp(location)@output(currentTemp)
```

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. See figure 3 for an example of how the environment gets updated in each step of a sequence pattern. It is important to note, that each workflow instance has its own environment satisfying the *case data* pattern of Russell [18] where data is specific to a certain process instance.

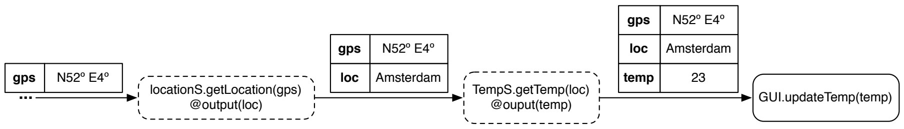

Fig. 3. Passing Environments in a Sequence

By introducing this dynamic data flow by means of passing an environment, we satisfy the key requirements of a data flow mechanism in a workflow model. As Sadiq et al stated [17], a data flow model must have the ability to:

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

The first requirements 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 name 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 also satisfied. The last requirement is also fulfilled because of the same reason.

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 figure 4. If a part of a workflow can be reached by more than one single path, it is possible that the environments are completely different.

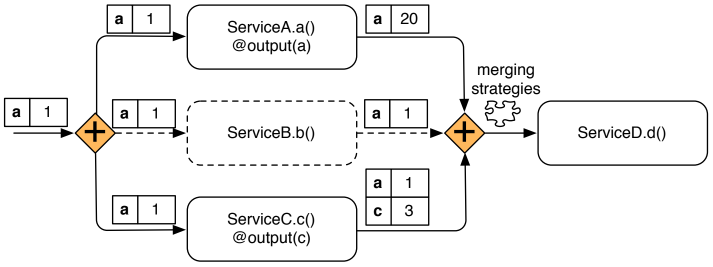

Fig. 4. Environments: Merging Strategies for Synchronization Patterns

As is shown in the example, when a `parallel split` is encountered, the environment is conceptually duplicated for each 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 identified several possible merging strategies which might be useful in certain cases:

- Prioritize one of the incoming branches when resolving conflicts.
- Pick the environment of one incoming branch and ignore the others.
- Conflicts are merged into a table containing the different values (with or without duplicates)
- Remember the “scope” from before merging and restore it (not always possible)
- Employ a user provided function to resolve conflicts.

The dynamic approach we propose, provides a much richer and 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. Our approach combines several advantages of the data patterns that are proposed by Russell et al [18].

4.3 Compensating Actions

In a dynamically changing environment, the challenge is to make the high 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 4 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. Examples of events we capture in a failure pattern are disconnections, reconnection, 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 5. Abstract grammar of the Failure pattern and the possible compensations.

```
<component>          := <activity> | <pattern>
<failure pattern>    := “ Failure (” <component> “,” <compensations> “)”
<compensations>      := <failure event> | <failure event> “,” <compensations>
<failure event>      := <disconnection> | <timeout> | <exception> | <service not found>
<timeout>            := “Timeout(” <duration> “,” <compensation> “)”
<disconnection>      := “Disconnect(” <compensation> “)”
<exception>          := “Exception(” <symbol> “,” <compensation> “)”
<service not found>  := “ServiceNotFound(” <compensation> “)”
<compensation>       := <retry> | <rediscover> | <skip> | <restart> | <wait> | <component>
<retry>              := “Retry(” <times> “,” <compensation> “)”
<rediscover>         := “Rediscover(” <times> “,” <compensation> “)”
<skip>               := “Skip()”
<restart>            := “Restart(” <times> “)”
<wait>               := “Wait(” <time> “,” <compensation> “)”
```

Listing 5 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 amount of times each compensating action is tried. When a compensating action has reached its maximum attempts, another (more drastic) one can be provided. We provide four kind of failures (disconnection, timeout, service not found and exception from a service) for which six different forms of compensating actions (retry, rediscover, skip, restart, wait, or execute a new subworkflow) are possible. 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. Another possible compensating action can be the execution of a subworkflow (activity or a pattern).

Russell [11] already classified workflow exception patterns that are used by workflow systems. The exceptions he discusses are for instance *constraint violation*, *deadline expiracy* 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.

Consider the example of updating a user interface with the current temperature at a user's location. As is depicted in figure 5, the location and weather service or both mobile. The default compensating actions for mobile services are to retry sending the message on a timeout, and to rediscover on a disconnection. By using the failure pattern (drawn as a dashed rectangle wrapping part of a workflow), we can specify other compensating actions and override this default behavior, as is shown for the weather service in figure 5.

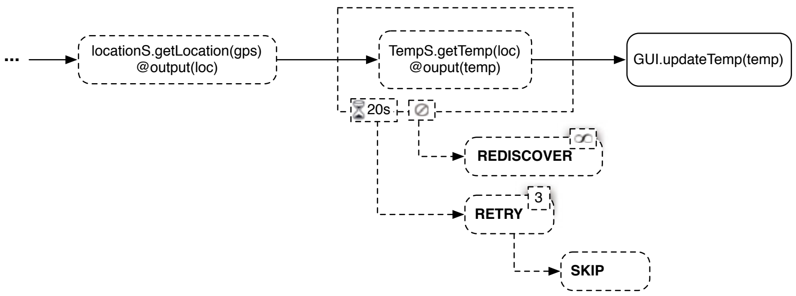

Fig. 5. Compensating actions specified for different kind of failures

When the weather service activity has a timeout (after 20 seconds no reply is returned) we try to resend the message three times. If this is still unsucessfull we move on to the next compensating action which just skips this activity (so no temperature gets displayed on the screen). In case of a disconnection however, a service of the same type must be (re)discovered. The implementation of this workflow can be seen in the following code snippet. Note that since there are no arguments specified, the compensating action of a disconnect event (`Rediscover` on line 4) is tried indefinitely.

```
def seq := Sequence( locationS.getLocation(‘gps)@output(‘loc),
                     Failure( TempS.getTemp(‘loc),
                              [ Timeout(20, Retry(3, Skip())),
                                Disconnect(Rediscover()) ]),
                     GUI.updateTemp(‘loc, ‘temp) );
```

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 seconds and try to rediscover” and smaller parts of this workflow can be wrapped with more specific failure patterns, which possibly overrides (shadows) the behavior imposed by the first failure pattern.

As the first requirement in section 2.2 stated, a workflow language targetting nomadic networks must support both stationary and mobile services. Mobile services are wrapped with a `Failure` pattern such that these services are rediscovered in case of a disconnection. Transparently compensating possible disconnections of mobile services is hence also satisfied (requirement 2).

4.4 Implementation of a Concrete Pattern

As van der Aalst defines, synchronization is a pattern that converges two or more incoming branches into a subsequent one when all incoming branches have been enabled [4]. A synchronize can occur after a pattern, for instance a parallel split, that has split the control flow into several outgoing branches.

Just like every control flow pattern we have implemented, a synchronize is an AmbientTalk object which has an `init` and `start` method. A synchronization is instantiated with a component (`cmp`)that must be executed after each branch is enabled.

Besides a start and init method (which are common by all patterns), a synchronize has a specific method `addSync` and data member `syncTable`. The `addSync` method is called by the connection pattern to notify the synchronization that it must wait for one more branch to be enabled. As a synchronization has to wait for each branch to be enabled, it has to wait for each branch to

```
def Synchronize := object: {
    def started := false;             // check if pattern is already started or not
    def result, resolver;             // [result, resolver] is a future
    def nextComponent;                // the component to be executed after synchronization
    def newEnv := Environment.new();  // the environment that is passed to nextComponent
    def syncTable := [];              // stores [future, env] tables for each incoming branch

    def init(cmp) {
        nextComponent := cmp;
        [result, resolver] := makeFuture(); };

    def addSync(future) {
        syncTable := syncTable + [ [future, []] ]; };

    def start(env) {
        // only start the component once
        if: !started then: {
            started := true;
            execute(); };
        result; };

    def execute() {
      def envs := [];
      syncTable.each: { |branch| envs := envs + branch[2]; };
      if: (envs.length() == syncTable.length()) then: {
          // all incoming branches have been enabled (futures are resolved)
          // merge the environments using the predefined strategy
          if: (is: nextComponent taggedAs: Activity) then: {
                // find the variable bindings in newEnv
                // invoke service (nextComponent.service) with these variable bindings
                // and possibly bind the output variables
                // and afterwards resolve the future with its reply
                 resolver.resolve(reply); // explicitly resolve the future
            } else: { // Start the component and resolve future with its reply
                 when: nextComponent.start(newEnv) becomes: { |reply|
                      resolver.resolve(reply); // explicitly resolve the future
                 }}}
        } else: { // not every future (of incoming branch) is resolved yet }};
} taggedAs: [SyncPattern];
```

resolve its *future* (explained in section 3.1). So, whenever `addSync` is called, the future of that branch is added to a `syncTable` such that when the synchronization pattern is started, it can iterate over this table and wait for each future to be resolved. Besides saving the futures, the `syncTable` also stores the environment with which the future is resolved.

A synchronize pattern must be started with one parameter, `env`, which is an environment that is passed among activities (as is explained in section 4.2). When the pattern is not yet started (line 23), the `execute` method is called which results in looking up the environments of the resolved futures in the `syncTable`. The if-test on line 17 is necessary, since the *start* method is called by each incoming branch. When the environments of all incoming branches have been stored in the table, they are merged using a specified strategy. Thereafter, the component after synchronization is executed. In case of an activity, the formal parameters of the method are looked up in the environment, and the service is invoked. After its invocation, the output variables are bound to the output values of the service invocation. At last, the future of the synchronize pattern is resolved with the reply of this component. Recall that this is needed to allow composition of patterns (as is explained in section 4.1). When the component is a workflow pattern, the pattern is started (with the merged environment) and afterwards the future of the synchronize is resolved with the received result (lines 42–45).

5 Related Work

Workflow languages targetting web services, like Bite [12] and Mobile Business Processes [16], can only operate on services that are known beforehand by means of a fixed URL. There also exist orchestration languages, like Orc [14] [15] which uses a process calculus to express the coordination of different processes. Chromatic Orc [19] extends this calculus with exception handling by introducing throw and try catch expressions that can both run in parallel and hence do not cause terminations. However, this language assumes s stable network interconnecting the services. Another orchestration language is Reo [13], a glue language that allows orchestration of different heterogeneous, distributed and concurrent software components, however one has to manually manage the disconnection of a component. Hence, these orchestration languages do not (completely) cope with the transient disconnections that are inherent to dynamically changing environments, like a nomadic network. CiAN [5] 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 approach does not hold in a dynamically changing environment, where services can join and disjoin at any moment in time. Another workflow system, Workpad [3], is designed with nomadic networks in mind, but also assumes that devices are connected upon startup. Open Workflows [8] is a system which targets workflow construction, allocation, and execution for mobile devices in mobile ad hoc networks. This system focusses on the dynamic construction of workflows based upon contextual information, something we do not target.

6 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 does allow the orchestration of distributed services in a nomadic network, thanks to both a peer-to-peer and dynamic service discovery mechanism and communication primitives resilient to the volatile connections inherent to such networks. Now provides high level patterns for control flow and mechanisms for detecting and handling both service and network failures which 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 communication* and the notion of an *entity* such as a passenger in the airport who does not execute a certain task like a service. Group communication is opportune for a dynamically changing environment where the number of communication partners (services or entities) can not be known beforehand and can vary over time. It would also be appropriate to enable *intentional descriptions* of communication partners as the changing network topology complicates extensional reasoning over these partners. The idea is to extend our language with Crime [10], a logic coordination language we have developed, and enable writing constraints for groups of services (and entities). The addition of these intentional descriptions also gives rise to the need for compensating actions when constraints are violated. Furthermore, support for some more advanced synchronization patterns is opportune because complete synchronization will not always be possible in environments where communication partners can go out of range at any point in time. We want to come up with synchronization patterns for group communication which can succeed for instance when only a certain percentage of communication partners has replied.

Acknowledgements. The authors would like to thank Stijn Verhaegen, Andoni Lombide Carreton, Niels Joncheere and the anonymous reviewers for their comments on earlier drafts of this paper. They have signicantly improved the readability and quality of this paper.

References

1. Dedecker, J., Van Cutsem, T., Mostinckx, S., D'Hondt, T., De Meuter, W.: Ambient-Oriented Programming. In: OOPSLA 2005, p. 3. ACM Press, New York (2005)
2. Mascolo, C., Capra, L., Emmerich, W.: Mobile computing middleware. In: Gregori, E., Anastasi, G., Basagni, S. (eds.) NETWORKING 2002. LNCS, vol. 2497, pp. 20–58. Springer, Heidelberg (2002)
3. Mecella, M., Angelaccio, M., Krek, A., Catarci, T., Buttarazzi, B., Dustdar, S.: Workpad: an adaptive peer-to-peer software infrastructure for supporting collaborative work of human operators in emergency/disaster scenarios. In: CTS 2006 (2006)
4. Russell, N., ter Hofstede, A.H.M., van der Aalst, W.M.P., Mulyar, N.: Workflow control-flow patterns: A revised view. Technical report, BPMcenter.org (2006)

5. Sen, R., Roman, G.C., Gill, C.D.: Cian: A workflow engine for manets. In: Lea, D., Zavattaro, G. (eds.) COORDINATION 2008. LNCS, vol. 5052, pp. 280–295. Springer, Heidelberg (2008)
6. Van Cutsem, T., Mostinckx, S., Gonzalez Boix, E., Dedecker, J., De Meuter, W.: Ambienttalk: object-oriented event-driven programming in mobile ad hoc networks. In: Proceedings of SCCC 2007, pp. 3–12. IEEE Computer Society, Los Alamitos (2007)
7. Van Cutsem, T., Mostinckx, S., De Meuter, W.: Linguistic symbiosis between event loop actors and threads. Computer Languages Systems & Structures 35(1) (2008)
8. Thomas, L., Wilson, J., Roman, G.C., Gill, C.D.: Achieving Coordination through Dynamic Construction of Open Workflows. In: Middleware 2009, pp. 268–287 (2009)
9. Andrews, T., Curbera, F., Dholakia, H., Goland, Y., Klein, J., Leymann, F., Liu, K., Roller, D., Smith, D., Thatte, S., Trickovic, I., Weerawarana, S.: Business Process Execution Language for Web Services (2003)
10. Mostinckx, S., Scholliers, C., Philips, E., Herzeel, C., De Meuter, W.: FactSpaces: Coordination in the Face of Disconnections. In: Murphy, A.L., Vitek, J. (eds.) COORDINATION 2007. LNCS, vol. 4467, pp. 268–285. Springer, Heidelberg (2007)
11. Russell, N., van der Aalst, W., ter Hofstede, A.: Workflow Exception Patterns. In: Dubois, E., Pohl, K. (eds.) CAiSE 2006. LNCS, vol. 4001, pp. 288–302. Springer, Heidelberg (2006)
12. Curbera, F., Duftler, M., Khalaf, R., Lovell, D.: Bite: Workflow composition for the web. In: Krämer, B.J., Lin, K.-J., Narasimhan, P. (eds.) ICSOC 2007. LNCS, vol. 4749, pp. 94–106. Springer, Heidelberg (2007)
13. Arbab, F.: Reo: a channel-based coordination model for component composition. Mathematical Structures in Comp. Sci. 14(3), 329–366 (2004)
14. Kitchin, D., Quark, A., Cook, W., Misra, J.: The orc programming language. In: Lee, D., Lopes, A., Poetzsch-Heffter, A. (eds.) FMOODS 2009. LNCS, vol. 5522, pp. 1–25. Springer, Heidelberg (2009)
15. Cook, W., Patwardhan, S., Misra, J.: Workflow patterns in orc. In: Ciancarini, P., Wiklicky, H. (eds.) COORDINATION 2006. LNCS, vol. 4038, pp. 82–96. Springer, Heidelberg (2006)
16. Pajunen, L., Chande, S.: Developing workflow engine for mobile devices. In: EDOC 2007, p. 279 (2007)
17. Sadiq, S., Orlowska, M., Sadiq, W., Foulger, C.: Data Flow and Validation in Workflow Modelling. In: Proceedings of ADC 2004, pp. 207–214 (2004)
18. Russell, N., ter Hofstede, A.H.M., Edmond, D., van der Aalst, W.M.P.: Workflow Data Patterns: Identification, Representation and Tool Support. In: Delcambre, L.M.L., Kop, C., Mayr, H.C., Mylopoulos, J., Pastor, Ó. (eds.) ER 2005. LNCS, vol. 3716, pp. 353–368. Springer, Heidelberg (2005)
19. Matsuoka, A., Kitchin, D.: A semantics for Exception Handling in Orc (2009), http://orc.csres.utexas.edu/papers/OrcExceptionSemantics.pdf