Enabling Adaptation of Pervasive Flows: Built-in Contextual Adaptation*

Annapaola Marconi[1], Marco Pistore[1], Adina Sirbu[1], Hanna Eberle[2], Frank Leymann[2], and Tobias Unger[2]

[1] Fondazione Bruno Kessler - Irst, via Sommarive 18, 38050, Trento, Italy
{marconi,pistore,sirbu}@fbk.eu
[2] Institute of Architecture of Application Systems,
Universitatsstrasse 38, 70569 Stuttgart, Germany
{eberle,leymann,unger}@iaas.uni-stuttgart.de

Abstract. Adaptable pervasive flows are dynamic workflows situated in the real world that modify their execution in order to adapt to changes in the execution environment. This requires on the one hand that a flow must be context-aware and on the other hand that it must be flexible enough to allow an easy and continuous adaptation. In this paper we propose a set of constructs and principles for embedding the adaptation logic within the specification of a flow. Moreover, we show how a standard language for web process modeling (BPEL) can be extended to support the proposed built-in adaptation constructs.

1 Introduction

In recent years, domains involving highly dynamic environments, such as pervasive computing and ambient intelligence, have turned their attention towards service oriented architectures (SOA). Indeed, even if SOA was initially designed for business contexts, its concept of building applications by exploiting and combining existing services matches very well the high variability, heterogeneity and dinamicity of these domains; this opens the possibility of re-using in these domains principles, methodologies and tools designed in the SOA framework. Conversely, for SOA, the dynamicity of these fields represents an important challenge that will contribute to speed up research on adaptability of service-based applications.

An example of this trend is the European project ALLOW [1]. The project exploits the well-known "workflow" concept, which has proven successful in the SOA field for modeling service-based applications, and uses it as the core of a new programming paradigm for human-oriented pervasive applications. More precisely, ALLOW's *Adpaptable Pervasive Flows* are workflows situated in the real world, i.e., they are logically or physically attached to entities like artifacts and people, move with them through different contexts. While being carried along, they model the behavior intended for their entity and the conditions on the execution context that guarantee a correct behavior. ALLOW's flows are hence capable to check deviations on the behavior of the entity they are attached to, as well as problems in the execution context, and to trigger adaptation.

There already exist pervasive computing infrastructures that use adaptation mechanisms (e.g., [6], [12]) . However, these mechanisms are mostly short-term, reactive

* This work is partially funded by the FP7 EU FET project Allow IST-324449.

L. Baresi, C.-H. Chi, and J. Suzuki (Eds.): ICSOC-ServiceWave 2009, LNCS 5900, pp. 445–454, 2009.

re-composition of services, or dynamic re-binding of components. The vision behind adaptable pervasive flows is to exploit the advantages of workflows to achieve kinds of adaptation beyond those already mentioned. *Short-term adaptation* will allow reacting to changes in the context by re-planning the structure of the running flow; it will be able to react not only to a change in the context, but also to detect that, given the current execution status, a constraint will be violated before a conflict actually occurs (proactive). Moreover, by analyzing information relative to past executions and adaptations of the flows, it will be possible to devise forms of *long-term adaptation*: the modifications on the flow produce produce a new generation of the flow model on which all future running flow will be instantiated.

A key enabling factor for all the aforementioned automated adaptation mechanisms is a convenient way of embedding the adaptation logic within the specification of a flow. The aim of this work is to present a set of modeling constructs and of tools that support the encoding of context-aware run-time flow adaptation. In particular, we propose a set of *built-in adaptation* modeling constructs that can be useful to add dynamicity and flexibility to flow models. For each built-in adaptation construct we provide a BPMN-like graphical representation and define a BPEL extension, with a clear syntax and operational semantics, that can be used to specify and execute Adaptable Pervasive Flows.

The paper is structured as follows. In Section 2 we present the adaptable pervasive flow paradigm proposed within ALLOW and we describe the main concepts concerning context-aware flow adaptation that drive the work described in this paper. Section 3 describes the built-in adaptation constructs that we propose for the encoding of context-aware adaptation within flow models. Finally, Section 4 presents some related works and draws conclusions, as well as on-going and future work.

2 Adaptable Pervasive Flows

Similar to the well-known workflows, adaptable pervasive flows (APF) consist of a set of activities and a corresponding execution order, which is specified using control elements such as sequence, choice, parallel operators.

A particular feature of APFs is that they are situated in the real world. This realizes the *pervasiveness* of the flows and is achieved in two ways. First, the flows are logically attached to physical entities (which can be either objects or humans) and move with them through different contexts. Secondly, they run on physical devices (e.g., PDAs, desktops). For instance, we can have a flow that models the shipment of a box and that is thus logically attached to that box; each fragment of a box flow is then potentially executed on different devices (e.g. the delivery part of the box flow is executed on the flow engine installed on the truck, while the storage part is executed on the PDA of the worker that in charge of storing the box).

Another important aspect of APFs is their *adaptiveness*. A flow is a dynamic entity that modifies its execution in order to adapt to changes in the execution environment. We consider different forms of flow adaptation. *Vertical* adaptation refines the flow or re-maps services to the flow without affecting the flow structure, while *horizontal* adaptation modifies the flow structure by adding, changing, or removing fragments of the flow. Moreover we distinguish between *instance-based* adaptation, were only the flow instance that triggers the adaptation need is modified, from *evolutionary* adaptation that,

on the basis of previous flow executions and adaptations, proactively modifies the flow model on which all future flow instances will be based.

In the following we briefly introduce the most important concepts related to APFs. For a detailed description of adaptable pervasive flows we refer the reader to [9].

After an analysis and comparison [3] of todays workflow standards the ALLOW project has chosen BPEL [11] as a nucleus for the Adaptable Pervasive Flow Language (APFL).

An important characteristic introduced by APFL is the distinction between abstract and concrete activities. An *abstract activity* is a non-executable activity that allows to partially specify the flow model at design-time. It expresses properties which will be used at run-time to properly associate a concrete flow (a flow where all the activities are concrete). A *concrete activity* is an executable flow activity. Concrete activities include all standard BPEL basic and structured activities (e.g. sending/receiving of a message, data manipulation, control constructs, parallel forks) and a set of APF-specific activities that have been defined as BPEL extensions. *Human interaction* activities are activities that require an interaction with a human, e.g. displaying or getting information through a device. *Context events* are a special type of activities for receiving events broadcasted by a particular entity called Context Manager. We call *flow scope* a connected set of flow activities with unique entry and exit points. Moreover, we distinguish between a flow and a flow instance. A *flow instance* is a particular execution of a flow. To better underline the difference, we sometimes refer to flows as *flow models*.

Another basic element of the flow is the *constraint*, which can be used to annotate a flow, a flow scope or an activity. There are multiple types of constraints: security, contextual, adaptation, distribution etc. In its basic form, a constraint is a condition on the execution of the flow. The most relevant kind of constraint for the problem adressed in this paper is the contextual one, since it allows to specify conditions on the flow execution environment. A first extension that has been defined, on which we base the work in this paper, aims at providing a modeling approach to annotate BPEL processes with contextual constraints and an execution model to monitor those constraints during flow execution (see [8] for details).

To better understand the concepts described above, we will consider adaptation examples on a concrete flow model: the flow logically attached to a box, which describes how the box should be handled when reaching a warehouse (see Figure 1). The flow refers to the Warehouse Management Case Study of the ALLOW Project described in [2]. The drawing of flows is based on the graphical representation of APFL basic and structured activities defined in [7].

The basic flow in Figure 1 consists of two abstract activities `Unload Me` and `Store Me`. During the execution of the flow, we assume that the box is not damaged: we model this as a context constraint `b.damaged == false`.

An example of vertical adaptation is the refinement of the abstract activity `Unload Me` to obtain a concrete flow that can be executed. This refinement is done at run-time and can be achieved through different techniques, e.g. binding the abstract activity to a concrete activity (e.g. web service, human task...), or, as in this example, substituting the abstract activity with a concrete flow that can either be pre-defined or computed by composing other concrete activities/flows. The flow waits to receive a context event that it has been `Picked Up` by a worker, and then sends to the worker, through a human interaction activity, the information on the location where the box should be brought to. It

then waits for a context event that it has been dropped. A characteristic of vertical adaptation is that, although a new flow is introduced, the structure of the original abstract flow remains unchanged.

On the contrary, horizontal adaptation affects the structure of the flow model. Consider for instance the situation where the context constraint `not(b.damaged)` is violated right after the unloading of a box. The adaptation mechanism tries to handle this assumption violation by modifying the flow instance structure. In particular, the damage extent is evaluated and, if the box can be repaired, the damage is fixed and the box can proceed with normal storage, otherwise the procedure for handling damaged items is started.

The adaptation cases presented so far are both examples of instance-based adaptation: a running flow instance is modified, refined or recomposed respectively, to react to an adaptation need. Now, suppose that analysing all past executions of `Box Unloading` flows, we find out that 10% of executions required to handle damaged boxes, and that in 90% of the cases, the horizontal adaptation variant devised in Figure 1 allowed to properly handle the violation of the flow constraint. We can decide to proactively embed this adaptation variant within the box flow model in such a way that all future executions will be able to cope directly with this adaptation need (evolutionary adaptation). Specifying such a flow requires having modelling tools that allow on the one hand to specify flexible, context and adaptation-oriented flows and on the other hand allow to keep trace of adaptation variants within the flow model.

The aim of *Built-in adaptation* is to tackle this problem, providing a set of constructs for embedding the adaptation logic within the specification of a flow. Although this is just a first (design-time, manual) form of adaptation, we believe it is not a trivial problem. Moreover, solving this problem will provide the modelling language that can be used when tackling automated adaptation problems.

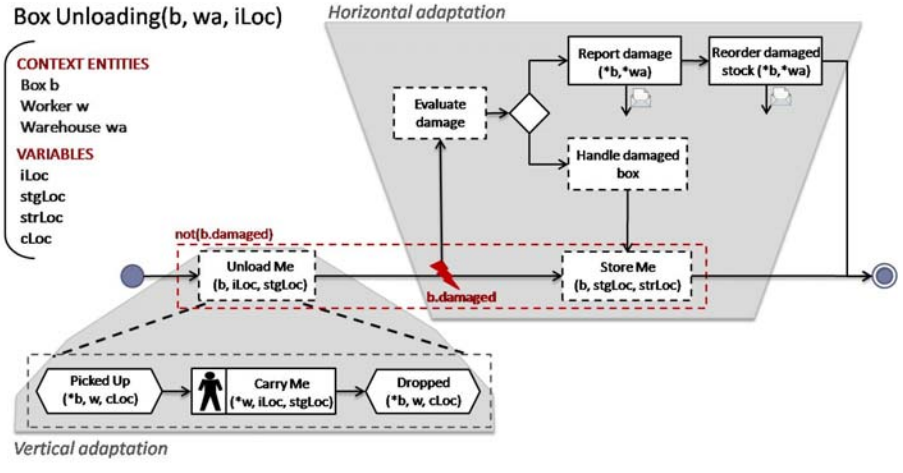


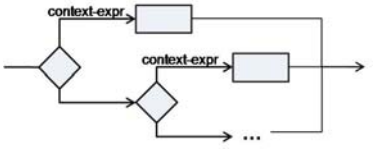
Fig. 1. Different Forms of Flow Adaptation

3 Built-in Adaptation Constructs

In the rest of this Section we present a set of built-in adaptation constructs that can support the encoding of context-aware adaptation within a flow model and for each construct we define the corresponding BPEL extension.

3.1 Basic Constructs: Context Conditions in Standard Control Constructs

The first and most simple kind of built-in adaptation constructs exploits the possibility to specify contextual conditions and applies it to standard BPEL control constructs (e.g. if, while).




```
<ext:contextualIf standard-attributes>
  standard-elements
  <ext:contextCondition expressionLanguage="anyURI"?>
    context-expr
  </ext:contextCondition>
  activity
  <ext:elseif>*
    <ext:contextCondition expressionLanguage="anyURI"?>
      context-expr
    </ext:contextCondition>
    activity
  </ext:elseif>
  <ext:else>?
    activity
  </ext:else>
</ext:contextualIf>
```

Fig. 2. `<ContextualIf>` Activity

For instance, the *Contextual If* construct, presented in Figure 2, allows to define several flow fragments as possible branches in the execution of the flow. Each flow fragment has an associated context condition. We can define also one flow fragment without a context condition, which will encode the default behavior. The syntax of the Contextual If is defined in Figure 2[1], where `context-expr` is a contextual condition and `activity` is any APFL simple or structured activity. The operational semantics of the construct is similar to a traditional if: the first fragment for which the context condition holds will be selected and executed.

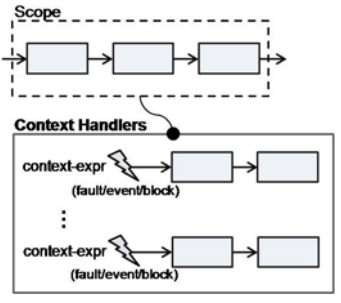
Similarly, we can extend other BPEL traditional control constructs.

3.2 Context Handlers

Testing a context condition at a certain moment in time is not always sufficient. Rather, we might need to monitor the condition during the execution of several activities, and to react to changes of this condition. If we consider the example of Figure 1, it can be the case that while the box is unloaded/stored the staging/storage location is not free anymore (e.g. because some other worker dropped a box there), or the box gets damaged. It would be useful to have the possibility to specify that a certain context condition must be monitored during the execution of a flow/scope and, if it is violated, execute a set of predefined activities.

This possibility is offered by the *Context Handler* construct (see Figure 3).

[1] The syntax of all BPEL extensions is defined using W3C XML Schema language and, when not explicitly specified, refers to standard BPEL constructs.




```
<ext:contextHandlers>?
  <ext:onContextEvent type="fault|event-blocking|event">+
    <ext:contextCondition expressionLanguage="anyURI"?>
      context-expr
    </ext:contextCondition>
    activity
  </ext:onContextEvent>
</ext:contextHandlers>
```

Fig. 3. `<contextHandler>` Activity

A context handler is associated to a scope, including the flow scope, and it defines a set of `contextEvents`, each specifying a context condition (`context-expr`) and a flow fragment (any APFL `activity`) that models the activities to be performed in case the corresponding context condition is violated. During the execution of the main flow, the context conditions are monitored and, as soon as one of them is violated, its corresponding flow fragment is executed.

We defined different kinds of context events within a context handler, namely `fault`, `event-blocking`, and `event`, which differ basically in the way their violation influence the execution of the scope to which the handler is attached.

When a fault-triggering condition is violated, the handling of the fault begins by stopping all active activities within the scope. Then, the flow fragment specified for that condition within the context handler is executed. The scope is considered to have not completed normally and as such is not eligible for compensation for that execution. Then normal process execution can resume from the point of the scope on. If this happens at the process level, then the process completes normally but would not be eligible for process instance compensation.

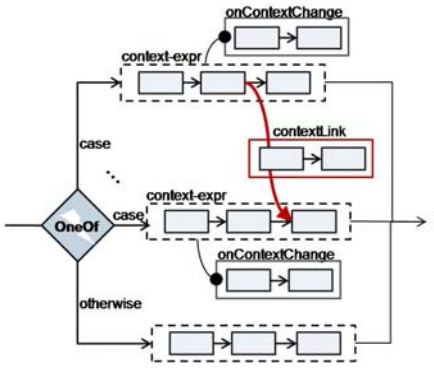
For what concerns event-triggering conditions, we propose two alternative kinds of context events: blocking and non-blocking. In the former case, whenever the condition is violated the execution of the scope is stopped, then the flow fragment specified within the context handler is executed and finally the scope execution is resumed. Whereas in the latter case the execution of the scope proceeds normally and the flow fragment specified within the context handler is executed concurrently.

Figure 5 presents an example of an event-blocking `contextHandler` used to check the availability of the assigned staging location during the unloading of a box.

3.3 Contextual One-of and Cross-Context Links

The aim of the *Contextual One-of* is to allow the design-time specification of a set of alternative flow fragments, each handling the execution of the flow within a specific context, and to allow at run-time to jump from one flow fragment to another, whenever the context changes or the assumptions on the context turn out to be wrong.

The Contextual One-of, as shown in Figure 4, consists of a set of alternative flow fragments, each of them associated to a contextual condition `context-expr`, modeling




```
<ext:contextualOneOf standard-attributes>
  standard-elements
  <ext:contextLinks>?
    <ext:contextLink name= NCName >+
      activity
    </ext:contextLink>
  </ext:contextLinks>
  <ext:case standard-attributes>+
    <ext:contextCondition expressionLanguage="anyURI"?>
      context-expr
    </ext:contextCondition>
    oneOf-activity
    <ext:onContextChange>
      oneOf-activity
    </ext:onContextChange>
    <faultHandlers>...</faultHandlers>
    <compensationHandlers>..</compensationHandlers>
  </ext:case>
  <ext:otherwhise>?
    oneOf-activity
  </ext:otherwhise>
</ext:contextualOneOf>
```

A `oneOf-activity` is any APFL activity where `standards-elements` are enriched with

```
<targetsCL>?
<targetCL contextLink="NCName" />+
</targetsCL>
<sourcesCL>?
<sourceCL contextLink="NCName"/>+
</sourcesCL>
```

Fig. 4. `<contextualOneOf>` Activity

the context assumption for that fragment, and a roll-back flow, `onContextChange`, that can be executed to undo the partial and unsuccessful work of the fragment.

At run-time, the first flow fragment for which the property holds is chosen and executed. During the fragment execution, its context condition is monitored and, as soon as it is violated, the following actions are performed:

1. *stop execution*: all running activities within the fragment are stopped;
2. *undo partial work*: the roll-back flow associated to the current fragment is executed;
3. *context jump*: the first fragment for which the associated context condition holds is executed and its context condition is monitored.

Roll-back flows, like any other flow, can throw faults/exceptions (e.g. to handle the fact that the work done within the fragment cannot be undone), and in this case the flow is terminated following normal flow fault handling. If this is not the case, and the roll-back flow completes successfully, the main flow is considered successfully running, no matter how many times contextual one-of fragments are rolled back and re-executed.

It is possible (not mandatory) to specify a default flow fragment for which no context condition is specified. If this is the case, the defualt fragment is executed only if no other context condition holds and, during its execution, no context condition is monitored (that is, unless faults occur, it will complete its execution and the contextual one of will complete successfully). During the execution of a contextual one-of, if all the context conditions are evaluated to false and no default fragment is specified, a fault is thrown and the flow terminates abnormally.

For exemplification, consider again the box flow presented in Section 2. A first problem that can occur here is that the box can be damaged. The damage may have occurred either before, during transportation, but it may also occur at any point while the box is

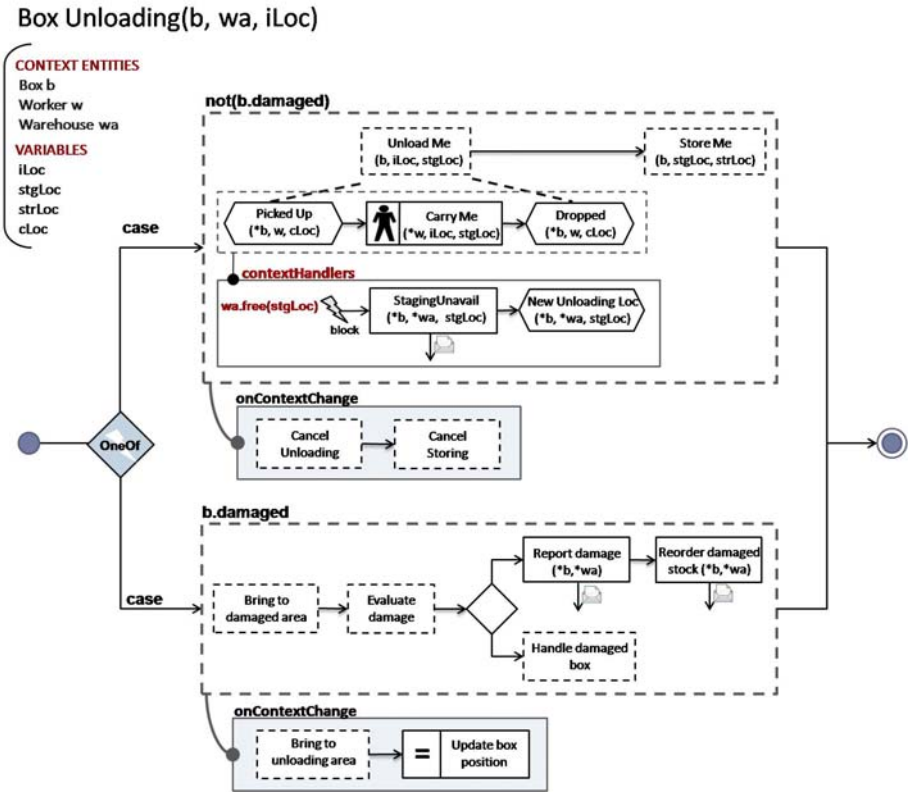


Fig. 5. Built-in Constructs at work: the Box Flow Example

being unloaded to the staging area, or moved to the storage area. In Figure 5 we use the Contextual OneOf construct to model the handling of damaged boxes.

When using the Contextual OneOf, it may be the case that, when jumping from one execution context to another, we do not want to undo the work done or the complete flow rollback is not possible. The *Cross-context link* (CL) is designed especially for this case. As can be seen from Figure 4, CLs connect two activities of different scopes within a Contextual OneOf. CLs allow adapting to a context change by jumping from a certain execution state of the current activity (`source` activity) to an execution activity (`target` activity) of another fragment suitable for the actual context. After the jump the flow instance must be in a consistent state. Therefore, a CL has an associate flow specifying the activities that are needed to prepare the flow to the jump.

At run time, if the contextual condition associated to the running scope turns out to be false, two possibilities are considered:

1. if there exists some context link leaving the active activity for which the context conditions holds
   (a) the roll-back flow associated to the cross-context link is executed

   (b) the monitoring for the new context condition is activated
   (c) the flow execution is re-started from the target activity of the cross-context link
2. otherwise the condition violation is handled as described for the standard Contextual one-of.

4 Conclusions and Future Works

We have provided an overview of the adaptable pervasive flows, a paradigm introduced in the ALLOW European project. We have presented the main concepts, as well as the associated adaptation methodology. After detailing the main adaptation strategies, we have focused on built-in adaptation, which is a design-time, evolutionary, horizontal, and fully manual strategy. For this particular strategy, we have presented several constructs which allow to encode the adaptation logic in the flow language and extended BPEL with suitable notations for the built-in adaptation constructs. These built-in adaptation constructs play an important role: they will serve as a basis for automated adaptation solutions.

Several adaptation approaches [13,4,14,5,10] have been proposed to address problems that are closely related to the built-in adaptation constructs presented in this paper. Most of them support the specification of context constraints within workflows [13,4,14,5] proposing different forms of context handling. In [10], the authors use an aspect-oriented programming (AOP) approach to adaptation.

Ongoing work aims at providing design tools and mechanisms for addressing automated flow adaptation. In particular, we are defining a formal language for APF that will enable the use of automated flow verification techniques [7]. Then, we plan to address run-time adaptation problems by providing a set of mechanism that can be used to compute adaptation variants during the execution of the flow instances. In the long term, we will devise mechanisms for analyzing historical data on flow executions and adaptations and that, on the basis of this analysis, proactively compute flow evolutions.

References

1. EU-FET Project 213339 ALLOW, `http://www.allow-project.eu/`
2. D2.1 Results of scenario analysis. ALLOW Project Deliverable (September 2008)
3. D3.1 Basic flow-model and language for Adaptable Pervasive Flows. ALLOW Project Deliverable (November 2008)
4. Adams, M., ter Hofstede, A.H.M., Edmond, D., van der Aalst, W.M.P.: Worklets: A Service-Oriented Implementation of Dynamic Flexibility in Workflows. In: Meersman, R., Tari, Z. (eds.) OTM 2006. LNCS, vol. 4275, pp. 291–308. Springer, Heidelberg (2006)
5. Baresi, L., Guinea, S., Pasquale, L.: Self-healing BPEL processes with Dynamo and the JBoss rule engine. In: Proc. of International workshop on Engineering of software services for pervasive environments (ESSPE 2007), pp. 11–20 (2007)
6. Becker, C., Handte, M., Schiele, G.: PCOM - A Component System for Pervasive Computing. In: Proc. of the International Conference on Pervasive Computing and Communications, PERCOM (2004)
7. Bucchiarone, A., Lafuente, A.L., Marconi, A., Pistore, M.: A formalisation of Adaptive Pervasive Flows. Submitted to WSFM 2009 (2009)
8. Eberle, H., Föll, S., Herrmann, K., Leymann, F., Marconi, A., Unger, T., Wolf, H.: Enforcement from the Inside: Improving Quality of Bussiness in Process Management. Accepted for ICWS 2009 (2009)

9. Herrmann, K., Rothermel, K., Kortuem, G., Dulay, N.: Adaptable Pervasive Flows – An Emerging Technology for Pervasive Adaptation. In: Proc. of the Self-Adaptive and Self-Organizing Systems Workshops (SASOW 2008). IEEE Computer Society, Los Alamitos (2008)
10. Kongdenfha, W., Saint-Paul, R., Benatallah, B., Casati, F.: An Aspect-Oriented Framework for Service Adaptation. In: Dan, A., Lamersdorf, W. (eds.) ICSOC 2006. LNCS, vol. 4294, pp. 15–26. Springer, Heidelberg (2006)
11. OASIS WSBPEL Technical Committee. Web Services Business Process Execution Language Version 2.0, 21, Committee Draft, work in progress (2005)
12. Roman, M., Hess, C.K., Cerqueira, R., Ranganathan, A., Campbell, R.H., Nahrstedt, K.: Gaia: A Middleware Infrastructure to Enable Active Spaces. IEEE Pervasive Computing, 74–83 (October–December 2002)
13. Wieland, M., Kopp, O., Nicklas, D., Leymann, F.: Towards Context-aware Workflows. In: CAiSE 2007 Proceedings of the Workshops and Doctoral Consortium (2007)
14. Wu, Y., Doshi, P.: Making BPEL Flexible: Adapting in the Context of Coordination Constraints Using WS-BPEL. In: WWW 2008 (2008)