

See discussions, stats, and author profiles for this publication at: <http://www.researchgate.net/publication/37686527>

Enabling Adaptation of Pervasive Flows: Built-in Contextual Adaptation

ARTICLE · NOVEMBER 2009

DOI: 10.1007/978-3-642-10383-4_33 · Source: OAI

CITATIONS

28

READS

23

6 AUTHORS, INCLUDING:



[Annapaola Marconi](#)

Fondazione Bruno Kessler

44 PUBLICATIONS **533** CITATIONS

[SEE PROFILE](#)



[Adina Sirbu](#)

Fondazione Bruno Kessler

8 PUBLICATIONS **53** CITATIONS

[SEE PROFILE](#)



[Frank Leymann](#)

Universität Stuttgart

494 PUBLICATIONS **11,245** CITATIONS

[SEE PROFILE](#)

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

Annapaola Marconi, Marco Pistore, Adina Sirbu

Fondazione Bruno Kessler - Irst, via Sommarive 18, 38050, Trento, Italy
[marconi,pistore,sirbu]@fbk.eu

Hanna Eberle, Frank Leymann, Tobias Unger

Institute of Architecture of Application Systems, Universitätsstrasse 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 dynamism 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 dynamism of these fields represents an important challenge that will contribute to speed up research on adaptability of service-based applications. Indeed, while the necessity to adapt to changing contexts and to evolving requirements is already present in business-oriented domains, it becomes critical in ubiquitous settings, where all the assumptions made at design time can turn out to be wrong at execution time.

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 *Adaptable 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 re-composition of services, or dynamic re-binding of components. Other forms of adaptation are usually not considered or must be performed manually. 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,

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

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 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. They include conditional branches within flows with context conditions as guard conditions; context handlers that allow to automatically react to context conditions violation during the execution of the flow; constructs that allow to specify a set of alternative paths, each handling a specific execution context; construct that allow reacting to changes in the execution context by "jumping" at run-time from one path to another. 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. Finally, we show how the proposed constructs can be exploited to model the adaptation logic within the Warehouse Management Case Study of the Allow project.

The paper is structured as follows. In Section 2 we present the adaptable pervasive flow paradigm proposed within ALLOW and in Section 3 we show them at work within a scenario from the Warehouse Case Study of the Allow project [2]. In Section 4 we describe the main concepts concerning context-aware flow adaptation that drive the work described in this paper. Section 5 describes the built-in adaptation constructs that we propose for the encoding of context-aware adaptation within flow models and defines their encoding as BPEL extensions. Finally, Section 6 presents some related works and Section 7 draws conclusions and outlines current 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. Due to this reason, another peculiar characteristic of APFs is that they are context-aware: during their execution they need a way to obtain information on their execution environment (e.g. on other flows, on the real world environment).

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 today's workflow standards the ALLOW project has chosen BPEL [11] as a nucleus for the Adaptable Pervasive Flow Language (APFL). BPEL provides an operational description of the stateful behavior of Web processes on top of the service interfaces defined in their WSDL specifications. A BPEL description identifies the partners of a process, its internal variables, and the operations that are triggered upon the

invocation of the process by some of the partners. Operations include assigning variables, invoking other processes and receiving responses, forking parallel threads of execution, and nondeterministically picking one amongst different courses of actions. Standard imperative constructs such as if-then-else, case choices, and loops, are also supported.


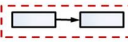

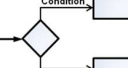

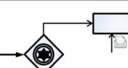

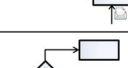
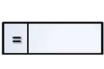

Basic activities		Structured activities	
	abstract activity		scope
	message sending		exclusive decision (if)
	message receiving		pick
	human interaction		parallel fork/join
	data manipulation		
	context event		

Fig. 1. APFL Basic and Structured Activities

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. More precisely, during this activity the flow execution waits until the event is received. 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. Then the actual purpose of the constraint is given by its type. For example, an adaptation constraint will define the activities that can be done when adapting the flow (e.g. we can specify that a flow cannot be adapted without human control). In its basic form, a constraint is a condition on the execution of the flow. The most relevant kind of constraint for the problem addressed in this paper is the contextual one, since it allows to specify conditions on the flow execution environment.

Today's workflow languages (e.g. BPEL) provide no possibility to explicitly model the context model and constraints on the workflow environment. The pervasiveness of a BPEL workflow can be specified only through ad-hoc interactions with external context-aware services. Clearly, this solution affects the readability and transparency of the pervasive aspect within the specified workflow. 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). The main concepts underneath the context model in [8] are that (i) the world model consists of a hierarchy of entities, each having a set of context properties, (ii) a flow/scope can be attached to world entities through context frames, and (iii) the flow/scope can specify context constraints and conditions on its internal variables and on the context properties of the entities within its context frame. From an architectural point of view, the idea is to have a module, the Context Manager, which is responsible of monitoring contextual constraints and can be queried to evaluate contextual conditions. However, the built-in constructs defined in this paper allow to specify contextual constraints in any other existing constraint language.

3 The Warehouse Management Scenario

In this section we present a scenario from the Warehouse Case Study of the Allow project [1] that we will use as a reference for all the examples within this paper.

The main aim of a warehouse management system (WMS) is to organize and control the movement and storage of goods within a warehouse. This is achieved through the definition and processing of complex transactions, including shipping, receiving, put-away, picking and issuing of goods. Its attributions stretch beyond the physical boundaries of the warehouse, involving for example the issuing of goods to other storing locations or final customers. The objective of the WMS is to provide a set of computerised procedures supporting all the aforementioned activities: from the handling of goods reception, storage and shipping, to the management of all the physical storage facilities. Warehouse management often utilizes Auto ID Data Capture technology, such as bar-code scanners, mobile computers, wireless LANs and potentially RFID to efficiently monitor the flow of products. The centralized system is in charge of controlling all aspects of warehouse management. Pervasive flows offer the possibility to distribute the control logics and hence to improve flexibility and context awareness of the executed processes.

The (logical and physical) structure of a warehouse is described by Figure 2. Doors are the locations where the goods arrive at or leave the warehouse. Trucks drive up to the doors of a warehouse in order to unload or load goods there. Staging areas are used for interim storage of goods in the warehouse. These are located in close proximity to the doors associated to them. The storage area is organized in several zones corresponding to different storage types. Storage types are physical or logical subdivisions of a warehouse complex, characterized by its warehouse technique, the space used, its organizational form, or its function.

Warehouse management requires the execution of different procedures which refer to the different objects and human actors, including good receipt, issuing and transfer. Here we focus on the first procedure that, from a high-level perspective, consists of three steps (see Figure 3): (i) delivery: a truck has reached the warehouse and is docking at the door; (ii) unload: goods are moved from the truck to a temporary staging area; (iii) put-away: goods are moved to the appropriate storage area. For the complete description of the goods receipt procedure, as well as complete modeling using adaptable pervasive flows, we refer the interested reader to [2].

We associate adaptable pervasive flows to each entity involved in the procedure. In particular, for the good receipt procedure, we associate APFs to trucks, boxes, workers, just to cite a few entity examples.

This has several advantages. First, the control logic can be distributed. For example, the information regarding the location where a particular box has to be staged or stored can be

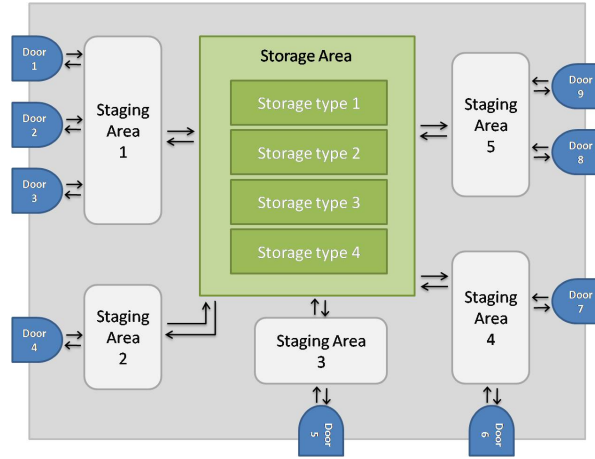


Fig. 2. Warehouse (Physical and Logical) Structure

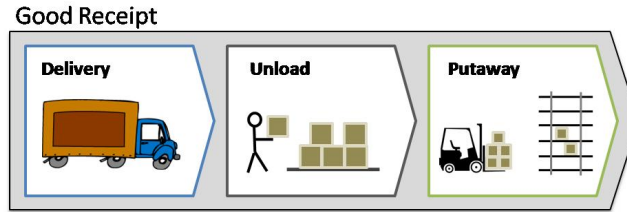


Fig. 3. Good Receipt Procedure

encoded in the unloading APF associated to the box. This allows the workers to obtain information directly from the box, without interacting with the warehouse. Since flows are context-aware, a second advantage is that decisions can be taken at run-time, based on information from the execution environment. Consider for example the case of a box containing perishable items. The flow associated to this box will know the amount of time the box can spend out of a refrigerator, and will be able to trigger an alarm as soon as a critical moment is reached. The third and most important advantage is flexibility. This is due to the fact that flows can modify their execution in order to adapt to changes in the underlying environment. Consider again the example of the perishable box. When the critical moment is reached, a possible solution to the problem is to temporarily move the box into a fridge case. The adaptation strategy will affect the structure of the flow, by adding activities that handle the new moving operation.

3.1 APF at Work

In this Section we present some flows defined for the good receipt procedure within the Warehouse Management scenario. The drawing of flows is based on the graphical representation of APFL basic and structured activities sketched in Figure 1.

Figure 4 shows the flow modeling the unloading of a truck. When a truck enters the warehouse it is sensed (`Truck sensed` context event) and the warehouse system is aware that a truck is approaching and has to be routed to a door in order to unload. The procedure for reaching a door may vary according to the specific warehouse. For this reason, this activity is modeled using the abstract activity `Reach the door`, that can be properly refined with the set of warehouse-specific activities to be performed (e.g. authentication, getting driving directions). In Figure 4 we present a possible refinement where the truck flow exchanges a

Truck Unloading(t, wa, cred)

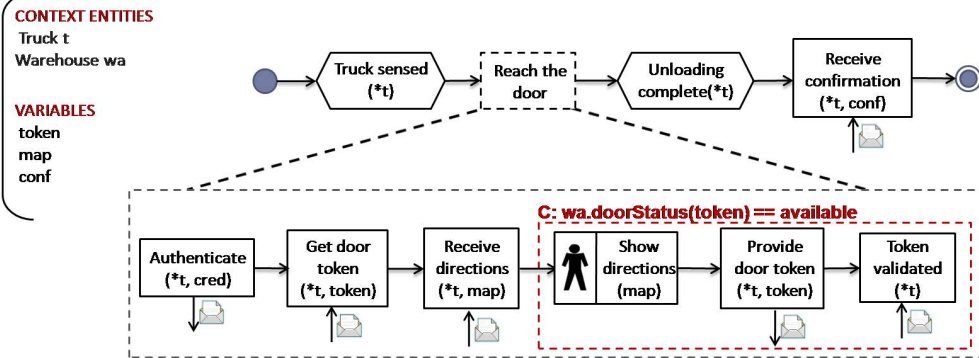


Fig. 4. Truck Unloading Flow

set of messages with the warehouse to authenticate and to get the access token and directions for the assigned door. The driving directions are shown to the driver (Show directions human interaction). Once the truck has reached the door, it must prove it is allowed to unload at that door by providing the obtained token. While the truck is approaching the assigned door, we monitor the fact that the door is available through the context constraint `wa.doorStatus(token) == available`. When the unload completes, the truck expects to receive a notification (Unloading complete context event), and then waits for the warehouse receipt.

Box Unloading(b, wa, iLoc)

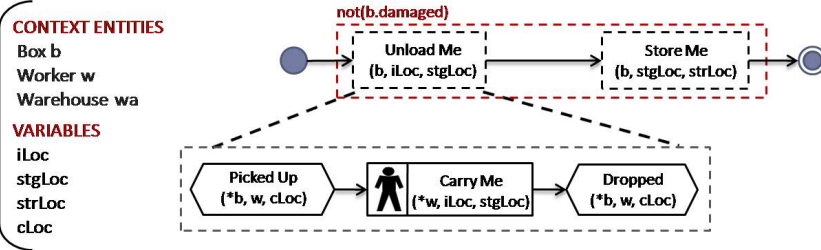


Fig. 5. Box Unloading Flow

Figure 5 presents the flow logically attached to a box, which describes how the box should be handled when reaching the warehouse. As already mentioned, the main idea here is that flows attached to boxes “know” their state (damaged/undamaged, hazardous, perishable, etc.), they know where they should be placed in the staging and storage area, and so on. The boxes interact with the equipment and personnel of the warehouse in order to transfer instructions, as well as to acquire information on the evolution of the flow status.

The flow consists of two abstract activities `Unload Me` and `Store Me`. Here, we also present the refinement for the `Unload Me`. 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. During the execution of the flow, we assume that the box is not damaged: we model this as a context constraint `b.damaged == false`.

4 Adaptation: Needs, Strategies and Techniques

The main purpose of adaptation is to detect when design-time assumptions no longer hold, and to react appropriately. This means bringing the flow into a stable situation that corresponds to the actual state of the world. At the same time, simply recovering from the erroneous state is not enough: the flow execution should be changed such that the flow can achieve its goals.

In the following, we present the methodology for handling adaptation proposed for adaptable pervasive flows. The first step is to identify the *adaptation need*. This is a complete description of the problem requiring adaptation and involves answering to several questions:

- What needs to be adapted? This means identifying the specific flow instance and flow activity or scope triggering the adaptation, as well as all the relevant context information.
- What is the cause? There are different events that can trigger the need for adaptation: violation of a constraint, an abstract activity that needs to be refined, impossibility to meet the flow goals etc.
- What should be the outcome? Here we must take into account the goal of the entity triggering adaptation, as well the adaptation constraints.

The next step is to find the *adaptation strategy* that best fits the adaptation need. There exist several adaptation strategies, e.g., flow binding, flow selection, flow composition etc. The appropriate adaptation strategy is chosen based on several criteria.

- When should the adaptation take place: design-time vs. run-time. With *design-time* adaptation, we encode the adaptation points and flow variants within the flow model. This requires extensive knowledge of the problems that might arise and their corresponding solutions. When we do not have this knowledge, or it is incomplete, *run-time* adaptation becomes appropriate: the adaptation variants will be computed during the execution of the flow instance.
- What should be adapted: instance-based vs. evolutionary. With *instance-based* adaptation, we modify only the flow instance that triggers the adaptation need, without affecting other already existing or future flow instances. In contrast, *evolutionary* adaptation modifies the flow model, and therefore all future flow instances. Evolution should typically be based on the experience from previous flow executions and previous adaptations.
- How should the adaptation affect the flow: vertical vs. horizontal. *Vertical* adaptation refines the flow or re-maps services to the flow, without affecting the flow structure. On the contrary, *horizontal* adaptation modifies the flow structure by adding, changing, or removing fragments of the flow.
- Who should perform the adaptation: the adaptation may be fully automatic, or may require user intervention.

The last step is choosing the *adaptation techniques* to implement the strategy. Adaptation techniques are actual algorithms and tools, e.g., algorithms for automatic composition of flows, data/protocol mediation techniques for run-time binding of flows.

To better understand the concepts described above, we will consider adaptation examples on a concrete flow model: the `Box Unloading` flow presented in Section 3.1.

An example of vertical adaptation is the refinement of the abstract activity `Unload Me` to obtain a concrete flow that can be executed (see in Figure 6). 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

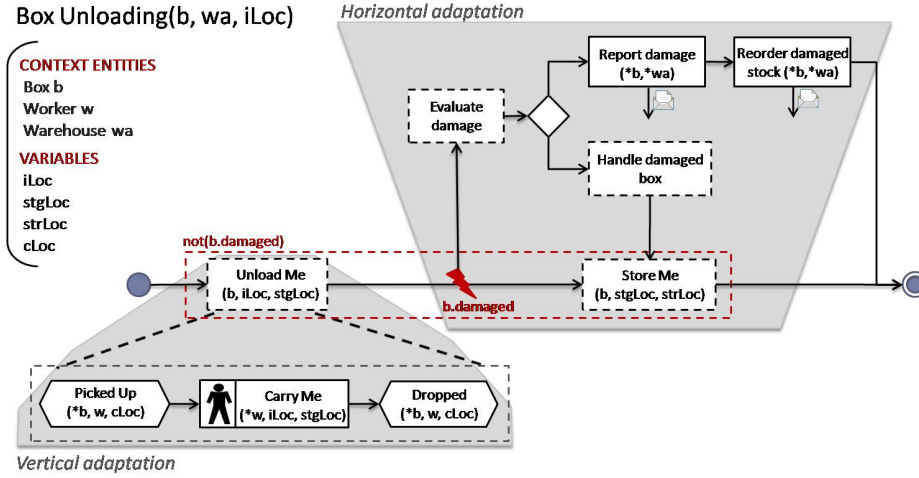


Fig. 6. Different Forms of Flow Adaptation

other concrete activities/flows. 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 $\text{not}(b.\text{damaged})$ 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 6 allowed to avoid 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.

5 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.

5.1 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).

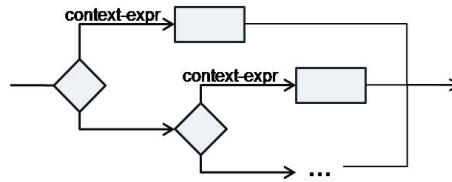


Fig. 7. <ContextualIf> Activity

For instance, the *Contextual If* construct, presented in Figure 7 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 the following ¹:

```
<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>
```

where *context-expr* is a contextual condition and *activity* is any BPEL 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. Clearly, such an extension requires the flow engine to query at run time the Context Manager in order to evaluate contextual conditions.

Similarly, we can extend other BPEL traditional control constructs.

5.2 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 5, 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 monitor these events and to model how to react to their occurrence. That is, it would be useful when modelling a flow 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 8).

A context handler can be associated to a scope, including the flow scope. The syntax of the *contextHandlers* construct is the following:

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

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

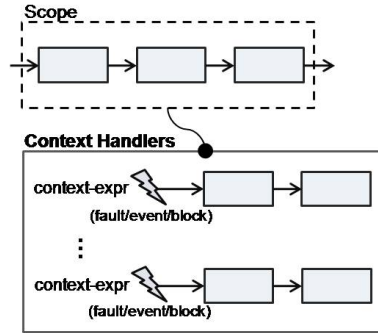


Fig. 8. <contextHandler> Activity

```

</ext:contextCondition>
  activity
</ext:onContextEvent>
</ext:contextHandlers>

```

A contextHandlers specifies 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.

The condition violation might affect the execution of the scope that severely, that no successful completion is possible any more. If the violation of the context condition expresses a less severe breach, the scope might proceed with its execution but additional activities must be performed to handle the condition violation. Due to this, 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.

5.3 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 5.3, consists of a set of alternative flow fragments, each of them associated to a contextual condition *context-expr*, modeling 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.

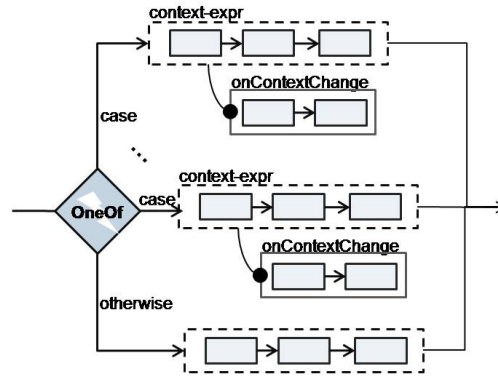


Fig. 9. <contextualOneOf> Activity

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 default 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.

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 5.3, 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

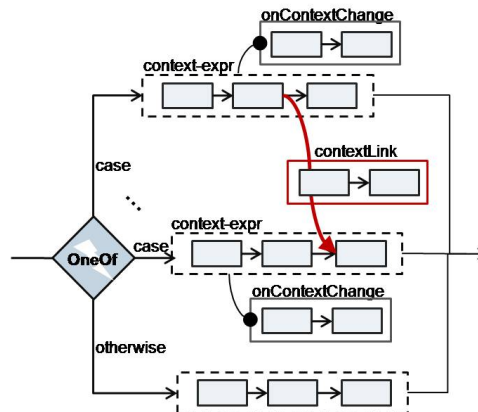


Fig. 10. Cross-context links

2. otherwise the condition violation is handled as described for the standard Contextual one-of.

In the following we present the BPEL syntax for the `contextualOneOf` with `contextLinks`.

```
<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:otherwise>?
    oneOf-activity
  </ext:otherwise>
</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>
```

As can be noticed from the syntax definition, it is possible to specify for each `case` local fault and compensation handlers. Semantically, the specification of local fault handlers and/or a local compensation handler is equivalent to the presence of an implicit `scope` activity immediately enclosing the `case` providing these handlers.

5.4 Built-in Constructs at Work: the Box Flow example

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 being unloaded to the staging area, or moved to the storage area.

Box Unloading(b, wa, iLoc)

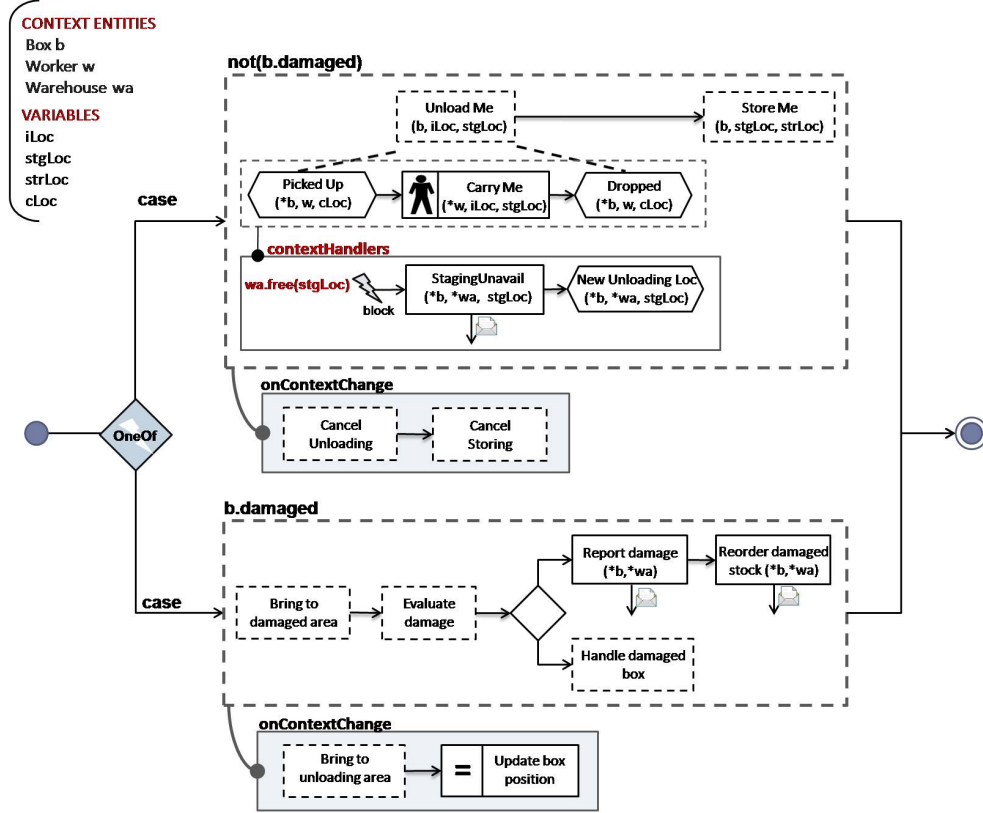


Fig. 11. Built-in Constructs at work: the Box Flow Example

In Figure 11 we use the Contextual OneOf construct to model the handling of damaged boxes. In case the box is not damaged, the first flow scope is chosen and executed. If at any point the box gets damaged, the context condition `not (b.damaged)` is violated and the `onContextChange` flow is executed. That is, pending activities for unloading and/or storing are canceled (e.g. the reserved staging/storage location is made available for other boxes, the request for unloading/storing sent to workers are revoked). The specific activities to be performed clearly depend on the state of execution. Due to this, abstract activities are specified, namely `Cancel Unloading` and `Cancel Storing`, and at run-time they will be refined with context-specific concrete activities. Once the `onContextChange` flow is executed, the control goes back to the `OneOf` and the scope handling damaged boxes is executed. If at some point in the execution of the flow scope for handling damaged boxes the box is repaired, the context condition associated to the scope (`b.damaged`) is violated, the execution stops and the `onContextChange` flow is executed. This way, the box is brought to a waiting area and its position is updated, and then the scope for handling undamaged boxes can start.

Another built-in construct exploited within the example in Figure 11 is the `contextHandler`. In particular, during the execution of the `Unload Me` refinement, it may be the case that the assigned staging location is no more available. If this is the case, the contextual constraint `wa.free(stgLoc)`, monitored during the whole execution of the refinement flow, is violated and the contextHandler is executed. Since the handler is defined as a blocking event, the execution of the main scope is suspended, then the handler flow is executed and once it completes, the main scope is resumed.

6 Related Works

Several adaptation approaches have been proposed to address problems that are closely related to the built-in adaptation tools presented in this paper. In the following we briefly recall them and point out the way in which they relate to our work.

In [13], the authors extend the Business Process Execution Language (BPEL) [11] in order to explicitly model the influence of the execution context on the workflow. The resulting context-aware workflows are modeled using context events, context queries and context decisions. With context events, the workflow waits asynchronously for a special environment state. Therefore, they realize a form of context handling. The context queries are used to filter or select objects based on spatial predicates. Thus, they are a specific form of context frames, as introduced in this paper. Lastly, the context decisions are used to route process flow based on internal/external context data, and therefore realize a form of `Contextual If`.

In [4], the authors start from Activity Theory principles and derive properties for a workflow support system. First, the process model should be regarded as a guide rather than a prescription. Then, a repertoire of actions (worklets) should be made available for each task. Each worklet-enabled task has associated a set of rules, and worklets can be selected at runtime using these rules and the relevant context. As a last property, the repertoire should be extendible at runtime. Therefore, both rules and worklets can be manually added at runtime. Since rules can be seen as context conditions, this approach realizes a declarative form of context handling.

Another work realizing a specific form of context handling is [14]. The author extend BPEL to deal with coordination constraints that may impose different reaction of a set of coordinated processes, depending on exogenous events. The presented approach is based on a compilation of the constraints into the original BPEL language, making use of event handlers to support this form of adaptability. The approach is quite at a preliminary stage; while no formal description of coordination constraints is given, examples hint at the fact they directly map to single exogenous events.

[5] proposes a framework (Dynamo) for self-healing BPEL compositions. More precisely, supervision rules are added to the BPEL process, specifying a location where the property should be verified, the monitoring parameters (priority, validity, trusted providers), a monitoring expression specified in WSCoL (Web Service Constraint Language) and a recovery strategy specified in WSReL (Web Service Recovery Language). Since the process is blocked while executing recovery, the approach is a form of context handling with fault-triggering conditions. Then, there can be different recovery strategies for the same violated constraint. Since the selection is done based on context, this can be seen as a form of `Contextual If`.

In [10], the authors use an aspect-oriented programming (AOP) approach to adaptation. Their motivation is twofold: because adaptation logic should be separated from business logic, and because adaptation can be seen as a cross-cutting concern. The paper provides a taxonomy of mismatches between external specification and service implementation. For each mismatch, a template, given in terms of a set of `<pointcut, advice>` pairs, defines the points where adaptation should be applied and the corresponding adaptation logic.

7 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.

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, reactively 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, Sept 2008.
3. D3.1 Basic flow-model and language for Adaptable Pervasive Flows. ALLOW Project Deliverable, Nov. 2008.
4. M. Adams, A. H. M. ter Hofstede, D. Edmond, and W. M. P. van der Aalst. Worklets: A Service-Oriented Implementation of Dynamic Flexibility in Workflows. In *Proc. CoopIS'06, OTM Conferrences*, pages 291–308, 2006.
5. L. Baresi, S. Guinea, and L. Pasquale. Self-healing BPEL processes with Dynamo and the JBoss rule engine. In *Proc. of International workshop on Engineering of software services for pervasive environments (ESSPE07)*, pages 11–20, 2007.
6. C. Becker, M. Handte, and G. Schiele. PCOM - A Component System for Pervasive Computing. In *Proc. of the International Conference on Pervasive Computing and Communications (PERCOM)*, 2004.
7. A. Bucchiarone, A. L. Lafuente, A. Marconi, and M. Pistore. A formalisation of Adaptive Pervasive Flows. Submitted to WFSM'09.
8. H. Eberle, S. Fil, K. Herrmann, F. Leymann, A. Marconi, T. Unger, and H. Wolf. Enforcement from the Inside: Improving Quality of Bussiness in Process Management. Accepted for ICWS'09.
9. K. Herrmann, K. Rothermel, G. Kortuem, and N. Dulay. 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, 2008.
10. W. Kongdenfha, R. Saint-Paul, B. Benatallah, and F. Casati. An Aspect-Oriented Framework for Service Adaptation. In *Proc. ICSOC'06*.
11. OASIS WSBPEL Technical Committee. *Web Services Business Process Execution Language Version 2.0*, 21 2005. Committee Draft, work in progress.
12. M. Roman, C. K. Hess, R. Cerqueira, A. Ranganathan, R. H. Campbell, and K. Nahrstedt. Gaia: A Middleware Infrastructure to Enable Active Spaces. *IEEE Pervasive Computing*, pages 74–83, Oct–Dec 2002.
13. M. Wieland, O. Kopp, D. Nicklas, and F. Leymann. Towards Context-aware Workflows. In *CAiSE'07 Proceedings of the Workshops and Doctoral Consortium*, 2007.
14. Y. Wu and P. Doshi. Making BPEL Flexible: Adapting in the Context of Coordination Constraints Using WS-BPEL. In *WWW 2008*.