A Formalisation of Adaptable Pervasive Flows[⋆]

Antonio Bucchiarone[1], Alberto Lluch Lafuente[2],
Annapaola Marconi[1], and Marco Pistore[1]

[1] FBK-IRST, via Sommarive 18, 38050, Trento, Italy
{bucchiarone,marconi,pistore}@fbk.eu
[2] Department of Computer Science, University of Pisa
lafuente@di.unipi.it

Abstract. *Adaptable Pervasive Flows* is a novel workflow-based paradigm for the design and execution of pervasive applications, where dynamic workflows situated in the real world are able to modify their execution in order to adapt to changes in their environment. In this paper, we study a formalisation of such flows by means of a formal flow language. More precisely, we define APFoL (Adaptable Pervasive Flow Language) and formalise its textual notation by encoding it in Blite, a formalisation of WS-BPEL. The encoding in Blite equips the language with a formal semantics and enables the use of automated verification techniques. We illustrate the approach with an example of a Warehouse Case Study.

1 Introduction

Flows are models defining a set of activities to be done, and their relations with each other. Flows are deeply seated in many fields, including business processes and service oriented computing. The flow modeling paradigm is often used either implicitly or explicitly in many real life situations. In this paper, we concentrate on a novel usage of flows, which is being investigated by the ALLOW project [1]: the usage of flows as a new programming paradigm for human-oriented pervasive applications. More precisely, *Adaptable Pervasive Flows* (APFs) [10] are proposed as an extension of traditional workflow concepts [19] in order to make them more flexible with respect to their pervasive execution environment. APFs are dynamic workflows situated in the real world that modify their execution in order to adapt to changes in their environment. This requires on the one hand that a flow must be context-aware: during execution it must be possible to obtain information on the underlying environment (e.g. relevant information on world entities, status of other flows, human activities). On the other hand flow models must be flexible enough to allow an easy and continuous adaptation. APFs are based on WS-BPEL [5], a well-known language for specifying flows in a Web Service setting, and extend it in order to implement all the aspects related to pervasive applications.

In [16] the authors define one of these extensions to WS-BPEL. More precisely, they define a set of constructs that allow a convenient way of embedding the adaptation logic within the specification of an APF and show how WS-BPEL can be extended

[⋆] Research supported by the EU, STREP project Allow IST-324449 and Sensoria, IST-2005-016004.

to support the proposed constructs. These constructs allow for capturing interesting cases of adaptation in pervasive applications that are difficult to address with classical workflows and with the standard WS-BPEL language. In this paper, we extend the work in [16], by providing a formal model for APFs and for the extensions of WS-BPEL related to the adaptation logics. This is a significant extension of the previous work since, due to the high dynamicity of pervasive applications, formal methods become crucial to drive design disciplines, equip existing languages with well-defined semantics and increase the reliability by means of automated verification.

Web services are a good example where many efforts are being invested on the development and application of formal methods. For instance, there have been various proposals to define formal semantics for WS-BPEL, typically by means of process calculi (e.g. [12], [15]), Petri nets (e.g. [14], [18]), or graphs (e.g. [7]).

In this paper we propose APFoL, a formal language for adaptable pervasive flows. Amongst the various formal approaches to flow languages we have chosen Blite [13] as a starting point. Blite is a process calculus that captures a significant part of the WS-BPEL language. We build our language as an almost straightforward extension of Blite. More precisely, we equip the language with abbreviations to deal with adaptation mechanisms, flow constructs and activity types typical of adaptable pervasive flows. The formal language proposed permit us to formally specify the built-in adaptation constructs informally proposed in [16].

This document is structured as follows. Section 2 introduces a scenario from the Warehouse Case Study of the Allow project [1] while Section 3 presents the background that is used in section 4 where our flow language is introduced. Section 5 illustrates our language with an example drawn from the Warehouse Case Study. Finally, Section 6 draws conclusions and outlines current and future work.

2 The Running Example: Warehouse Management

We present a scenario from the Warehouse Case Study of the Allow project [1], namely the management of a warehouse that we will use as a reference for all the examples within this document. The main aim of warehouse management is to organize and control the transport 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. The objective of the warehouse management system 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 AIDC (Automatic Identification and Data) technology, such as bar-code scanners, mobile computers, wireless LANs and potentially RFID (Radio Frequency Identification) to efficiently monitor the flow of products. Current systems require that, once data has been collected, synchronization with a centralized system is performed. 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.

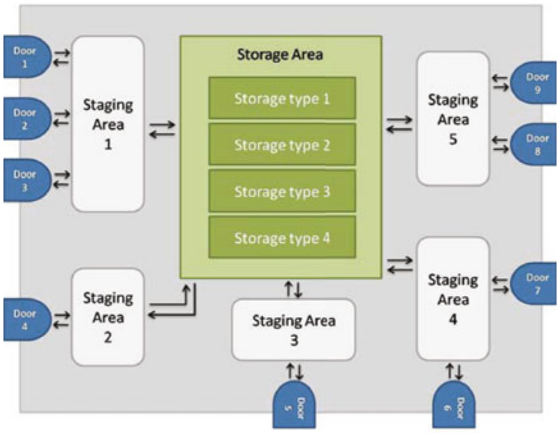

Fig. 1. Warehouse structure

The (logical and physical) structure of a warehouse is described by Figure 1. 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, which describes the three steps (see Figure 2) that have to be performed when the goods arrive at the warehouse: *delivery* (a truck has reached the warehouse and is docked at a door); *unload* (goods are unloaded and temporally stored in the staging area); and *put-away* (goods are moved to the storage area).

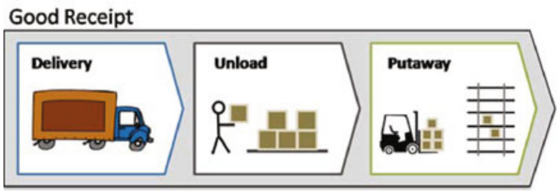

Fig. 2. Good receipt procedure

3 Background

Our work is strongly based on a proposal for the extension of WS-BPEL to deal with adaptation [16] and the process calculus Blite [13].

3.1 WS-BPEL and APFs

Similar to the well-known workflows, APFs consist of a set of activities and a corresponding execution order, which is specified using control elements such as sequence, choice or parallel operators. After a deep analysis and comparison of todays workflow standards using criteria such as industry impact, robustness aspect and extensibility [2], the ALLOW project has chosen WS-BPEL as a nucleus for an APF language.

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). Todays workflow languages (e.g. WS-BPEL) provide no possibility to explicitly model the context model and constraints on the workflow environment. The pervasiveness of a WS-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 [9] that has been done aims at providing a modeling approach to annotate WS-BPEL processes with contextual constraints and an execution model to monitor those constraints during process execution.

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. A key enabling factor for automated adaptation mechanisms is a convenient way of embedding the adaptation logic within the specification of a flow. Adaptation mechanisms cannot be limited to standard recovery constructs (e.g. fault/event/compensation handlers in WS-BPEL), but should also support the specification of flexible context-aware reactions to adaptation needs that can be used to handle run-time flow deviations without requiring a flow recovery/failure. The aim of the work in [16] is to present a set of primitives and principles that can support the encoding of context-aware run-time deviations and changes within a flow model in a secure (from an execution perspective) and convenient (from a modelling perspective) way. The authors propose a set of *built-in adaptation* modeling constructs that can be useful to add dynamicity and flexibility to flow models and for each construct they define the corresponding WS-BPEL extension. In particular, the proposed constructs are (i) conditional branches within flows with context conditions as guard conditions, (ii) context handlers that allow to automatically react to context conditions violation during the execution of the flow, and (iii) constructs that allow to specify a set of alternative scopes, each handling a specific execution context, and that allow to jump at run-time from one scope to another, whenever the context changes or the assumptions on the context turn out to be wrong.

3.2 WS-BPEL and Blite

Blite [13] is a formal language for describing web service orchestrations. It has been designed as a formal model to capture the essentials of WS-BPEL, a de-facto standard for describing web services. Blite is a process calculus and as such it has a well-defined notion of syntax and operational semantics. The language includes features such as

Table 1. Syntax of Blite

Basic activities	$\mathsf{b} ::=$	$\mathsf{inv}\,\ell^i\,\mathsf{o}\,\bar{\mathsf{x}} \mid \mathsf{rcv}\,\ell^r\,\mathsf{o}\,\bar{\mathsf{x}} \mid \mathsf{x} := \mathsf{e}$	invoke, receive, assign
	$\mid$	$\mathsf{empty} \mid \mathsf{throw} \mid \mathsf{exit}$	empty, throw, exit
Structured activities	$\mathsf{a} ::=$	$\mathsf{b} \mid \mathsf{if(e)}\{\mathsf{a}_1\}\{\mathsf{a}_2\} \mid \mathsf{while(e)}\,\{\mathsf{a}\}$	basic, conditional, iteration
	$\mid$	$\mathsf{a}_1\,;\mathsf{a}_2 \mid \sum_{j\in J}\mathsf{rcv}\,\ell^r_j\,\mathsf{o}_j\,\bar{\mathsf{x}}_j\,;\mathsf{a}_j$	sequence, choice (with $\lvert J \rvert > 1$)
	$\mid$	$\mathsf{a}_1 \mid \mathsf{a}_2 \mid [\mathsf{a} \bullet \mathsf{a}_f \star \mathsf{a}_c]$	parallel, scope
Start activities	$\mathsf{r} ::=$	$\mathsf{rcv}\,\ell^r\,\mathsf{o}\,\bar{\mathsf{x}} \mid \sum_{j\in J}\mathsf{rcv}\,\ell^r_j\,\mathsf{o}_j\,\bar{\mathsf{x}}_j\,;\mathsf{a}_j$	receive, choice
	$\mid$	$\mathsf{r}\,;\mathsf{a} \mid \mathsf{r}_1 \mid \mathsf{r}_2 \mid [\mathsf{r} \bullet \mathsf{a}_f \star \mathsf{a}_c]$	sequence, parallel, scope
Services	$\mathsf{s} ::=$	$[\mathsf{r} \bullet \mathsf{a}_f] \mid \mu \vdash \mathsf{a} \mid \mu \vdash \mathsf{a}\,,\mathsf{s}$	definition, instance, multiset
Deployments	$\mathsf{d} ::=$	$\{\mathsf{s}\}_\mathsf{c} \mid \mathsf{d}_1 \,\Vert\, \mathsf{d}_2$	deployment, composition

service definition and instantiation, typical flow constructs, communication primitives and failure handling and compensation mechanisms. We offer here a brief, intuitive overview of Blite and refer to [13] for a detailed presentation.

The syntax of the Blite language is summarised in Table 1 (borrowed from [13]). *Basic activities* include variable assignments, flow success related operations (throw and exit) and communication primitives to send (inv) or receive (rcv) values from partner links. *Structured activities* organise basic activities in flows by using typical flow constructs such as branches, sequences, loops, fork&join and (input-prefixed) choices[1]. In addition, scopes can be defined with appropriate failure and compensation activities. *Start activities* are structured activities starting with a choice of receive operations. The reason for this is that service definitions are inactive until they receive a request. Services already instanced, instead are represented by their memory μ (an assignment of values to variables) and their flow (a structured activity). Deployments (i.e. the system) are sets of correlated services.

4 A Formal Language for Adaptable Pervasive Flows

This section presents our formalisation of adaptable pervasive flows in terms of a flow language that we call APFoL.

We present here our language for adaptable pervasive flows, describing each ingredient and giving its encoding in Blite. It is worth mentioning that the encoding automatically equips our language with a formal semantics.

We start offering an informal presentation of our visual notation, which we plan to formalise in the future, possibly by means of a graphical encoding of our language. Here we just present the informal visual notation as Figures 3, 4 and 5, that act as an illustration of the textual notation that we shall describe in detail.

We now present the textual notation of APFoL (summarised in Table 2), which basically extends the syntax of Blite (c.f. Table 1) with ad-hoc constructs for built-in adaptation mechanisms.

[1] Called *pick* in [13] but *choice* here to avoid confusion with the pick flow construct of APFoL.

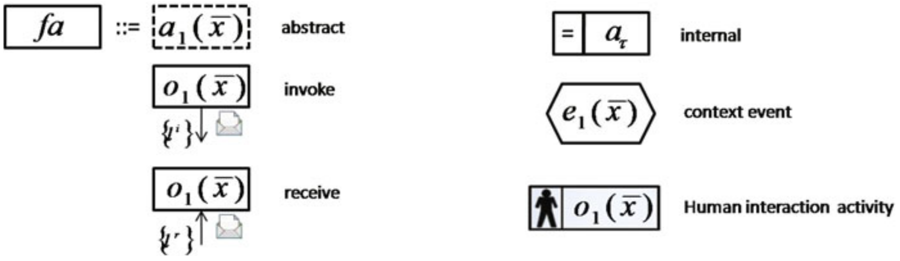

Fig. 3. Visual syntax for Flow Activities of Table 2

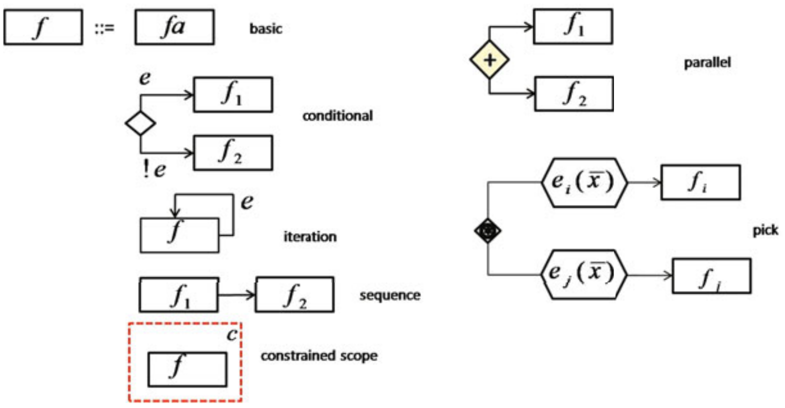

Fig. 4. Visual syntax for Flow Instances (Part-I) of Table 2

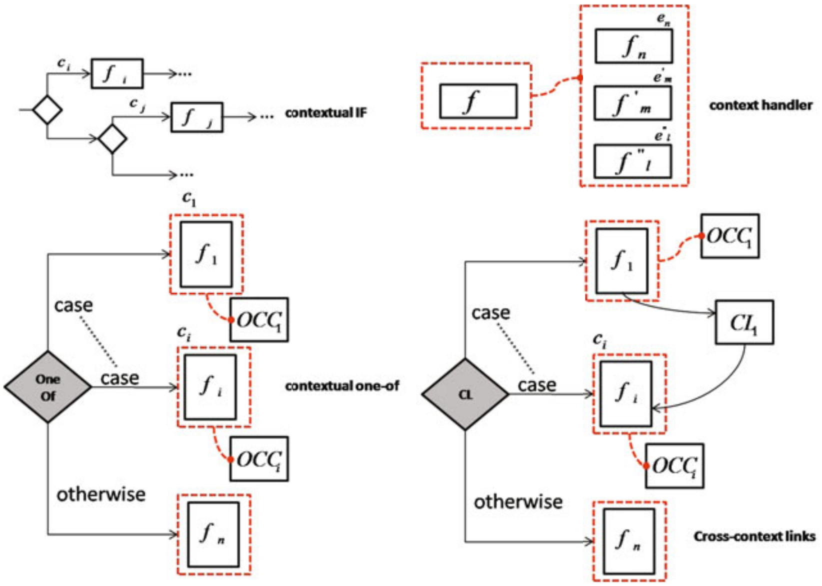

Fig. 5. Visual syntax for Flow Instances (Part-II) of Table 2

The main differences with respect to Blite's syntax regard basic and structured activities. We call them called *basic and structured flows* in APFoL to avoid confusion and stick to the APF slang. They include some new constructs to model the relevant primitives of adaptable pervasive flows. Services and deployments remain identical but, again, we call them differently (*flows and flow* systems) to avoid confusion. Finally, we avoid presenting the productions for *start flows* for simplicity: they are a straightforward adaptation of those for *start activities* in the same way as flow instances are an adaptation of activity instances.

First, APFoL includes the same control flow constructs of Blite. In addition, even if not part of the primitive syntax summarised in Table 1, APFoL includes typical control flow constructs such as different forms of branching (e.g. switch) and looping (e.g. loop-exit) which are straightforwardly encoded in Blite.

An *abstract activity* $A(\overline{x})$ represents either a partial design-time specification of a flow model or an abbreviation of a complex activity. Abstract activities are modelled just as function symbols A of type fa (flow activity) or fr (start activity), thus $A(\overline{x})$: fa∪fr in Table 2. For each such symbol we assume a definition to exist (for abbreviations) or to be given at run-time (for partial designs). Refining an abstract activity then means

Table 2. Syntax of APFoL

Flow activities	fa ::=	sinv l o x̄ $\mid$ srcv l o x̄ $\mid$ a_τ	invoke, receive, internal
	$\mid$	$\langle e\overline{x}\rangle$ $\mid$ $A(\overline{x})$	context event, abstract activity
Flows instances	f ::=	fa $\mid$ if(e){f$_1$}{f$_2$} $\mid$ while(e) {f}	basic, conditional, iteration
	$\mid$	f$_1$; f$_2$ $\mid$ f$_1$ $\mid$ f$_2$	sequence, parallel
	$\mid$	$[\![f]\!]_a^c$ $\mid$ $\mathsf{pick}_{i\in J}(e_i \rightarrow f_i)$	constrained scope, pick
	$\mid$	$\mathsf{cIF}_{i\in J}(c_i \rightarrow f_i)$	contextual IF
	$\mid$	$[\![f]\!]\lfloor \overset{e_1}{f_1}\rfloor\lfloor \overset{e_n}{f_n}\rfloor\lfloor \overset{e'_1}{f'_1}\rfloor\lfloor \overset{e'_m}{f'_m}\rfloor\lfloor \overset{e''_1}{f''_1}\rfloor\lfloor \overset{e''_l}{f''_l}\rfloor$	context handler
	$\mid$	one − of$[\![f_1]\!]_{r_1}^{c_1} \ldots [\![f_n]\!]_{r_n}^{c_n}$	contextual one-of
Flows	ff ::=	[fr • f] $\mid$ $\mu \vdash$ f $\mid$ $\mu \vdash$ f , ff	definition, instance, multiset
Flow system	fs ::=	{ff}$_c$ $\mid$ fs$_1$ ‖ fs$_2$	deployment, composition

Table 3. Blite encoding of the main ingredients of APFoL

Sending activity	$\mathsf{sinv}\ q\ o\ \bar{x} \overset{\mathbf{def}}{=} \mathsf{inv}\ \langle \mathsf{id}, q\rangle\ o\ \bar{x}\ ;\ \mathsf{rcv}\ \langle \mathsf{id}\rangle\ o\ ack$
Receiving activity	$\mathsf{srcv}\ q\ o\ \bar{x} \overset{\mathbf{def}}{=} \mathsf{rcv}\ \langle \mathsf{id}, q\rangle\ o\ \bar{x}\ ;\ \mathsf{inv}\ \langle q\rangle\ o\ ack$
Context event	$\langle e\ \bar{x}\rangle \overset{\mathbf{def}}{=} \mathsf{rcv}\ ContextManager\ get_e\ \bar{x}$
Internal event	$\langle e\rangle \overset{\mathbf{def}}{=} while(e)\{empty\}$
Constrained scope	$[\![f]\!]_a^c \overset{\mathbf{def}}{=} \langle c\rangle\ [\{f\}^{\neg c} \mid (\langle \neg c\rangle;\ \mathsf{throw})\ \bullet\ a * \mathsf{empty}]$
Pick	$\mathsf{pick}_{i\in J}(e_i\bar{x}_i \rightarrow a_i) \overset{\mathbf{def}}{=} \sum_{i\in J}(\langle e_i\bar{x}_i\rangle\ ;\ a_i)$
Contextual IF	$\mathsf{cIF}_{i\in J}(c_i \rightarrow f_i) \overset{\mathbf{def}}{=} \mathsf{switch}_{i\in J}\mathsf{c_i} \rightarrow [\![f_i]\!]_{\mathsf{throw}}^{c_i}$

replacing the left-hand side of a definition by its right-hand side. Clearly, this is not a real extension of the language and is standard machinery of all algebraic specifications.

Communication activities allow for sending (resp. receiving) a message to (resp. from) another flow. Invoke and receive activities are synchronous. Thus we encode them as suggested in [13] by the authors of Blite, namely by a pair of receive and invoke actions. In the definition, id stands for the flow instance identity. A sending activity is thus encoded as the invocation of operation o at partner l, where the identity of the flow is passed to receive the response. Similarly, the receive activity expects to receive an invocation of operation o at himself (id) together with the invoker's identity q which is used to send the response.

Data manipulation activities are internal activities that change the value of local variables and do not interact with their environment. Data manipulation activities are modelled as structured activities whose component basic activities a_τ are all assignments. Note that the grammar for a_τ can be given but, for the sake of a clear presentation, prefer to avoid adding an ad-hoc syntactical category and its (rather redundant) productions.

Human interaction activities are activities that require an interaction with a human, e.g. displaying or getting information through a device. Human interaction activities are modelled as communication operations, since devices are represented by flows.

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 model context events as particular receive operations. More precisely, we shall model context managers as services that broadcast their events e via replicated sending operations named get_e. Thus, the reception of the event is modelled as a reception operation with the context manager as partner, operation get_e and the corresponding tuple of values.

We shall also use a sort of *internal events*, denoted by $\langle e \rangle$ whose meaning is to wait until the expression (i.e. a condition or trigger) e is true.

A *constrained scope* $[\![f]\!]_a^c$ is a flow f enclosed into a scope with unique entry and exit points, a constraint c and adaptation a (triggered if the constraint is not valid). They are represented in our syntax by terms of the form $[\![f]\!]_a^c$, where f is the normal flow, c is the constraint and a is the adaptation to be performed in case the condition fails inside the scope. This is modelled in Blite by exploiting the failure mechanism. First, we wait until the condition is true. Then we open a Blite scope where we put a condition observer flow in parallel with the normal flow conditioned to $\neg c$[2]. Conditioning is necessary to avoid the normal flow to progess in case the context condition violated. Note that this semantics does not really interrupt the flow. The exception is raised and the failure code performs the adaptation activity, only when the observer is executed.

A *pick* is a branching point in the process where the alternatives are based on events, rather than the evaluation of expressions. More precisely, it is the receipt of a message or of a context event that determines which of the paths will be taken. Event-based decision is modelled by a non deterministic choice of activities preceded by the corresponding triggering event.

[2] This is done by inserting a condition (in form of an *internal event*) between each activity and can be easily defined in a recursive manner.

$$\llbracket main \rrbracket \lfloor \overset{fault_1}{ff_1} \rfloor \ldots \lfloor \overset{fault_n}{ff_n} \rfloor \lfloor \overset{event_1}{ef_1} \rfloor \ldots \lfloor \overset{event_m}{ef_m} \rfloor \lfloor \overset{block_1}{bf_1} \rfloor \ldots \lfloor \overset{block_l}{bf_l} \rfloor \overset{\mathbf{def}}{=}$$

```
    new $done$;
    $done := false$;
    $\llbracket \{main\}^{suspend}$
        $| fault_1 \rightarrow throw$
        $| \ldots$
        $| fault_n \rightarrow throw$
        $| event_1 \rightarrow ef_1$
        $| \ldots$
        $| event_m \rightarrow ef_m$
        $| block_1 \rightarrow bf_1; done := true$
        $| \ldots$
        $| block_l \rightarrow bf_l; done := true$
    $\rrbracket^{fault_1 \vee \ldots \vee fault_1 n}_{switch\{fault_1 \rightarrow ff1 \ldots fault_n \rightarrow ff_n\}}$
```

$$suspend \overset{\mathbf{def}}{=} (\neg done \wedge block_1) \vee \ldots \vee (\neg done \wedge block_l)$$

Fig. 6. Context handler in Blite

A *contextual IF* 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 behaviour. The operational semantics of this construct is similar to a traditional if: the first fragment for which the *context condition* holds will be selected and executed.

A *context handler* is a particular flow associated to a *main* flow or flow scope. It specifies an alternative flow (in the form of a contextual IF) to be applied if a corresponding scope condition is violated. There are various forms of conditions within the context handler flow.

- *Fault-triggering* suspend all active tasks in the main flow and execute the corresponding error-handler. If the main flow is a flow scope, the fault is propagated to the enclosing scope.
- *Event-triggering* conditions can be *non-blocking* (execution of the main flow proceeds normally and the corresponding flow is executed concurrently) or *blocking* (execution of the main flow suspends, then the corresponding flow is executed and finally the main flow is resumed).

The encoding of context handlers in Blite is defined in Figure 6. The construct consists of a main flow *main* and three sets of observers: fault-handlers, non-blocking event handlers, and blocking event handlers, whose triggers are respectively denoted by *fault*, *event* and *block*, while the corresponding flows are respectively denoted by *ff*, *ef* and *bf*. The idea is as follows: the main flow (conditioned to *suspend*) is put in parallel with various observers one for each fault and event trigger. When a fault condition triggers, an error is raised and handled by the error handler which selects a fault and fires the corresponding flow. Non-blocking events trigger the corresponding flow. Blocking events perform similarly, but note that the *suspend* condition depends on the blocking event

```
one − of$[\![f_1]\!]_{r_1}^{c_1} \ldots [\![f_n]\!]_{r_n}^{c_n} \overset{\text{def}}{=}$
    new $success$;
    loop
        switch
            $c_1 \rightarrow [\![f_1; success := true]\!]_{r_1}^{c_1}$
            ...
            $c_n \rightarrow [\![f_n; success := true]\!]_{r_n}^{c_n}$
        if($success$) exit;
```

Fig. 7. Contextual One-of in Blite

conditions: if one of them is true and the corresponding flow has not been performed, the main flow remains suspended until *done* becomes true[3].

A *contextual one-of* consists of a set of alternative flow fragments, each of them associated to a contextual condition modeling the contextual assumption for that fragment, and a rollback flow that can be executed to undo the partial and unsuccessful work of the fragment. At run-time, the first flow fragment for which the contextual condition 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 holds is executed and its context condition is monitored.

Roll-back flows can throw fault/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.

The encoding in Blite is rather easy, a loop is used to guarantee that performing a rollback returns to the selection of one of the choices. The only way to exit a loop is to successfully finish one of main flows.

When using the *contextual one-of*, 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. CLs connect two activities of different scopes within a *contextual one-of*. 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 associated flow needed to prepare the flow to the jump. At runtime, if the contextual condition associated to the running scope turns out to be false, two possibilities are considered:

[3] Note that in order to guarantee a unique *done* variable we declare it as *new* at the beginning of the encoding. This is not a feature of Blite but can be added straightforwardly by considering the local store μ as a stack of assignment sets instead of a plain set.

```
CL$[\![f_1]\!]_{r_1}^{c_1} \dots [\![f_n]\!]_{r_n}^{c_n} \overset{\text{def}}{=}$
    new $success$; new $next\_flow$; $next\_flow := any$;
    loop
        switch
            $c_1 \wedge proceed(1) \rightarrow [\![f_1; success := true]\!]_{r_1}^{|c_1|}$
            ...
            $c_n \wedge proceed(n) \rightarrow [\![f_n; success := true]\!]_{r_n}^{|c_n|}$
        if($success$) exit;
```

Fig. 8. Contextual One-of with CLs in Blite

1. if there exists some context link leaving the active activity for which the context condition 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 CL
2. otherwise the condition violation is handled as described for the standard *contextual one-of*.

The encoding in Blite is similar to the encoding of ordinary contextual one-of. The first difference is that the guard of each flow f_i is enriched with $proceed(i)$ which is an abbreviation for $next\_flow = i \vee next\_flow = any$. This serves to control which flow should be executed next. The second difference is that each compensation c must take the ad-hoc roll-back flows for cross-context jumps into consideration. With $|c|$ we denote the introduction of a choice that decides whether to apply the ordinary compensation c or the roll-back flow associated to the jump to the next flow.

5 The Box Unloading Example

In this section we present a complete example that summarizes most elements introduced before. For exemplification, we consider the box flow. 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.

In Figure 9 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

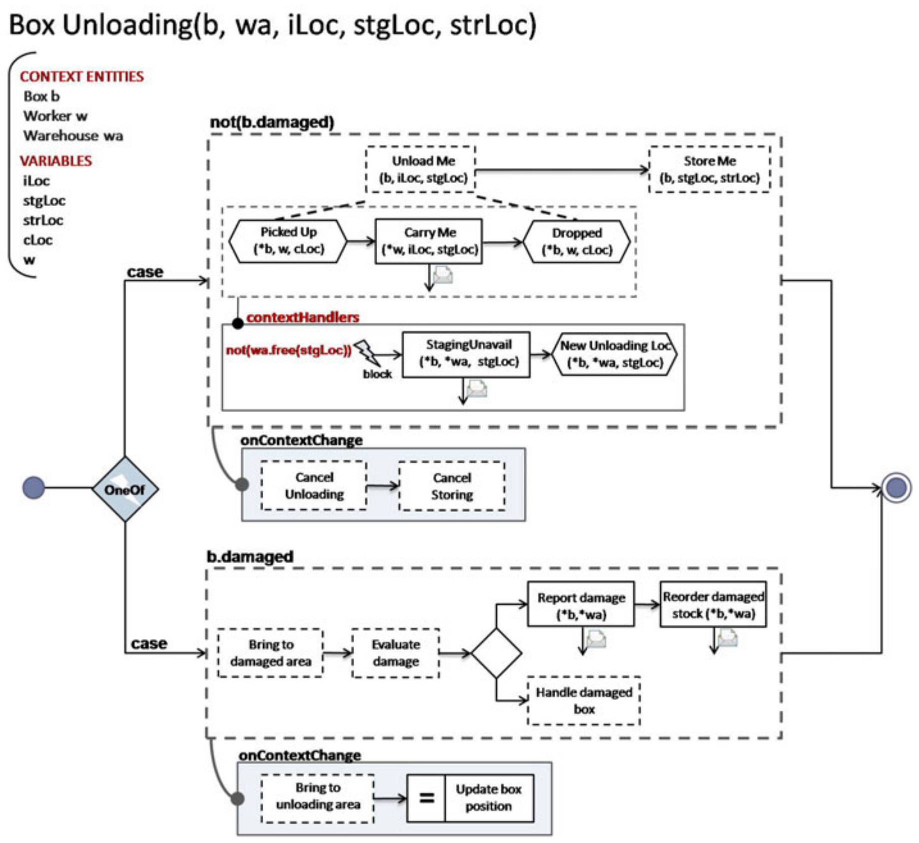

Fig. 9. The Box Unloading example

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 9 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 then the main scope is resumed.

The APFoL code of this example is flow Box Unloading shown in Figure 5, while its WS-BPEL code is listed in [3].

$\text{Box Unloading} \overset{\text{def}}{=} \text{one}-\text{of}[\![notDamaged]\!]^{c_1}_{occ_1}\,[\![Damaged]\!]^{c_2}_{occ_2}$

$\quad c_1 \overset{\text{def}}{=} not(b.damaged)$

$\quad occ_1 \overset{\text{def}}{=} CancelUnloading;$

$\qquad\quad CancelStoring$

$\quad c_2 \overset{\text{def}}{=} b.damaged$

$\quad occ_2 \overset{\text{def}}{=} BringtoUnloadingArea;$

$\qquad\quad UpdateBoxPosition$

$notDamaged \overset{\text{def}}{=} UnloadMe(b, iLoc, stgLoc);$

$\quad StoreMe(b, stgLoc, strLoc)$

$\quad UnloadMe(b, iLoc, stgLoc) \overset{\text{def}}{=} [\![main]\!]\lfloor \overset{block_l}{bf_l} \rfloor$

$\quad StoreMe(b, stgLoc, strLoc) \overset{\text{def}}{=} \ldots$

$\quad main \overset{\text{def}}{=} \langle PickedUp\ (*b, w, cLoc)\rangle;$

$\qquad\quad \text{sinv}\ w\ \text{CarryMe}\ (*\text{w}, \text{iLoc}, \text{stgLoc});$

$\qquad\quad \langle Dropped\ (*b, w, cLoc)\rangle$

$\quad block_1 \overset{\text{def}}{=} not(wa.free(stgLoc))$

$\quad bf_1 \overset{\text{def}}{=} \text{sinv}\ wa\ \text{StagingUnavail}\ (*\text{b}, *\text{wa}, \text{stgLoc})\ ;$

$\qquad\quad \langle NewUnloadingLoc\ (*b, *wa, stgLoc)\rangle$

$Damaged \overset{\text{def}}{=} BringtoDamagedArea;$

$\qquad\quad EvaluateDamage;$

$\qquad\quad \text{if(repairable)}\{\text{f}_1\}\{\text{f}_2\}$

$\quad f_1 \overset{\text{def}}{=} HandleDamagedBox$

$\quad f_2 \overset{\text{def}}{=} \text{sinv}\ wa\ \text{ReportDamage}\ (*\text{b}, *\text{wa});$

$\qquad\quad \text{sinv}\ wa\ \text{ReorderDamagedStock}\ (*\text{b}, *\text{wa})$

Fig. 10. APFoL encoding of flow Box Unloading

6 Conclusion and Future Work

We have described a preliminary version of APFoL, a language for adaptable pervasive flows with formal support. More precisely, we have presented a language whose textual notation is based on Blite [13], a process calculus for WS-BPEL.

The formalisation of the language equips the language with a well-defined (and hence non-ambiguous) semantics. More precisely, the formalisation as a process calculus (Blite) facilitates the use of automated verification techniques. Some support for the analysis and verification of Blite specifications exists (an encoding from Blite into another calculus with tool support [4]), but we are working in a direct implementation of Blite semantics in the rewrite engine Maude [8], in order to exploit its generic built-in capabilites in form of analysis tools such as a model checker and a theorem prover. With suchan implementation at hand, APFoL can be implemented as a derived rewrite theory in Maude.

Our approach has been illustrated with examples from the Warehouse case study of the Allow project [1].

We plan to develop a graph-based formalisation of our visual notation, possibly basing existing techniques for the graphical encoding of process calculi (e.g. [6]). The main goals of having a formal graph-based representation is formalise the relation betweeen textual and visual notation and to enable the use of graph transformation techniques, and their corresponding tools.

In the future we would like to investigate the connection with apparently similar approaches in the data base community around the notion of *business artifacts* [17,11], which are flows attached to physical objects moving through different contexts.

Acknowledgments

The authors would like to thank Roberto Bruni for sharing with us his knowledge on WS-BPEL formalisations and the authors of Blite (Rosario Pugliese, Francesco Tiezzi and Alessandro Lapadula) for providing us with useful material.

References

1. EU-FET project 213339 ALLOW, `http://www.allow-project.eu/`
2. D3.1 Basic flow-model and language for Adaptable Pervasive Flows. ALLOW Project Deliverable (November 2008)
3. APFoL homepage, `http://www.antoniobucchiarone.it/APFoL.html`
4. Blite: A formal account of WS-BPEL, `http://rap.dsi.unifi.it/blite/`
5. OASIS WSBPEL Tecnical Committee. Web Services Business Process Execution Language, version 2.0 (2007), `http://docs.oasis-open.org/wsbpel/2.0/wsbpel-v2.0`
6. Bruni, R., Gadducci, F., Lluch Lafuente, A.: A graph syntax for processes and services. In: 9th International Workshop on Web Services and Formal Methods, WS-FM 2009 (2009)
7. Bundgaard, M., Glenstrup, A.J., Hildebrandt, T.T., Højsgaard, E., Niss, H.: Formalizing higher-order mobile embedded business processes with binding bigraphs. In: Lea, D., Zavattaro, G. (eds.) COORDINATION 2008. LNCS, vol. 5052, pp. 83–99. Springer, Heidelberg (2008)
8. Clavel, M., Durán, F., Eker, S., Lincoln, P., Martí-Oliet, N., Meseguer, J., Talcott, C.: All About Maude - A High-Performance Logical Framework. How to Specify, Program and Verify Systems in Rewriting Logic. LNCS, vol. 4350. Springer, Heidelberg (2007)
9. Eberle, H., Fll, S., Herrmann, K., Leymann, F., Marconi, A., Unger, T., Wolf, H.: Enforcement from the inside: Improving quality of bussiness in process management. In: IEEE 7th International Conference on Web Services, ICWS 2009 (2009) (to appear)
10. Herrmann, K., Rothermel, K., Kortuem, G., Dulay, N.: Adaptable Pervasive Flows - An Emerging Technology for Pervasive Adaptation. In: Workshop on Pervasive Adaptation (PerAda), September 2008. IEEE Computer Society, Los Alamitos (2008)
11. Hull, R.: Artifact-centric business process models: Brief survey of research results and challenges. In: Meersman, R., Tari, Z. (eds.) OTM 2008, Part II. LNCS, vol. 5332, pp. 1152–1163. Springer, Heidelberg (2008)
12. Laneve, C., Zavattaro, G.: Web-pi at work. In: De Nicola, R., Sangiorgi, D. (eds.) TGC 2005. LNCS, vol. 3705, pp. 182–194. Springer, Heidelberg (2005)
13. Lapadula, A., Pugliese, R., Tiezzi, F.: A formal account of WS-BPEL. In: Lea, D., Zavattaro, G. (eds.) COORDINATION 2008. LNCS, vol. 5052, pp. 199–215. Springer, Heidelberg (2008)

14. Lohmann, N.: A feature-complete Petri net semantics for WS-BPEL 2.0. In: Dumas, M., Heckel, R. (eds.) WS-FM 2007. LNCS, vol. 4937, pp. 77–91. Springer, Heidelberg (2008)
15. Lucchi, R., Mazzara, M.: A pi-calculus based semantics for WS-BPEL. Journal of Logic and Algebraic Programming 70(1), 96–118 (2007)
16. Marconi, A., Pistore, M., Sirbu, A., Eberle, H., Leymann, F.: Enabling adaptation of pervasive flows: Built-in contextual adaptation. In: Baresi, L., Chi, C.-H., Suzuki, J. (eds.) ICSOC 2009. LNCS, vol. 5900, pp. 389–403. Springer, Heidelberg (2009)
17. Nigam, A., Caswell, N.S.: Business artifacts: An approach to operational specification. IBM Systems Journal 42(3), 428–445 (2003)
18. Ouyang, C., Verbeek, E., van der Aalst, W.M.P., Breutel, S., Dumas, M., ter Hofstede, A.H.M.: Formal semantics and analysis of control flow in WS-BPEL. Science of Computer Programming 67(2-3), 162–198 (2007)
19. van der Aalst, W.M.P., van Hee, K.M.: Workflow Management: Models, Methods, and Systems. MIT Press, Cambridge (2002)