Managing Adaptivity in Web Collaborative Processes Using Policies and User Profiles

Juri Luca De Coi[1], Marco Fisichella[1], and Maristella Matera[2]

[1] Forschungszentrum L3S, Hannover 30167, Germany
{decoi, fisichella}@L3S.de
[2] Politecnico di Milano, Milano 20133, Italy
matera@elet.polimi.it

Abstract. *Adaptive Web collaborative processes* are flexible processes executed on the Web, that can configure themselves according to information not available at definition time.In this paper we show how such processes can be supported by integrating an engine for flexible process definition and execution, COOPER, and a policy engine, PROTUNE. The resulting framework is *open* in its nature, since it can integrate any (external) source of data. In particular, we show how our framework can exploit user profiles available on the Web to support process adaptivity.

1 Introduction

In several contexts, the success and popularity of workflow applications is limited by the intrinsic complexity of Workflow Management Systems (WfMSs), and especially by the low flexibility of workflow enactment, which leaves no freedom to process actors to tailor processes and process activities to their specific needs [15]. This is especially true whenever different actors have to coordinate toward a common goal (e.g., the release of some artifact), since *collaborative processes* are difficult to predict completely in advance and, unlike traditional business processes, they escape the ability of being fully modeled [7]. For this reason, process engines are needed which are flexible enough to be adapted to the preferences of the individual actors and to the evolution of background knowledge and competences.

Flexibility can refer both to the whole process (i.e., to the flow of the different activities) as well as to single process activities, which actors might want to configure and personalize themselves. Flexibility can be achieved in several ways: in this paper we suggest to foster it by adapting processes according to information which is not available at definition time or can vary over time, such as data stored in a user profile. We propose a solution based on the integration of a Web platform for flexible process definition and execution, COOPER [6], and a policy engine, PROTUNE [3,4], which enables adaptivity rules to be enacted during process execution based on context information. The resulting framework is *open* in its nature, since it can integrate any (external) source of data. In particular, we show how our framework can exploit user profiles available on the Web (e.g., the ones provided by social applications) to support process adaptivity.

1.1 Running Scenario

To highlight the requirements that motivated our work and to exemplify the concepts introduced in this paper, we consider a scenario involving a process commonly enforced by employees of the fictitious Computer & Communication Research Center (C3R in the following). C3R employees are assigned to projects which are managed by project leaders. Each C3R employee is allowed to attend up to a given number of conferences per year. This number can be exceeded upon project leader's approval. Finally, for each conference the registration fee and the travelling expenses cannot exceed a given amount altogether.

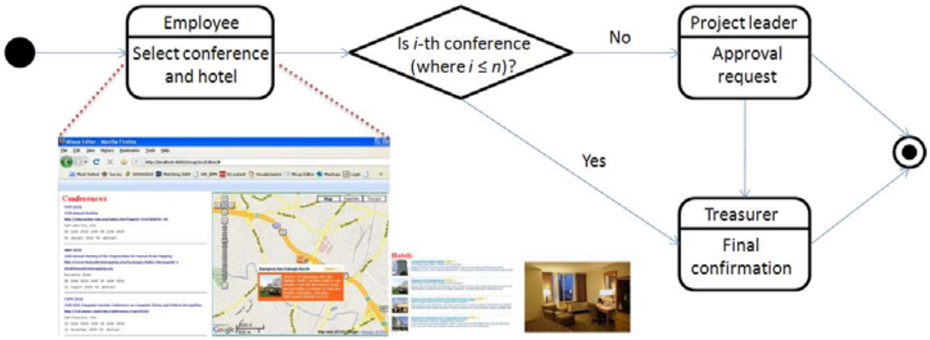

Fig. 1. Workflow of our running scenario

Fig. 1 shows a workflow corresponding to our scenario: whenever an employee wants to attend some conference, she starts the workflow and enters the first activity. She is thus presented with a subset of the conferences indexed by DB-World[1], which are relevant to the project she works in and her research preferences. As soon as the employee selects a conference, a Google map[2] shows its location together with hotels close to it. Only hotels are displayed such that the price of the overnight stay throughout the conference together with the conference registration fee does not exceed the allowed amount. Finally, as soon as a hotel is selected, its images available on Flickr[3] are automatically displayed.

After the employee is done with her activity, the control flow comes either to the treasurer or to the project leader. The first case happens if and only if in the current year the employee did not attend the maximum allowed number of conferences n yet . In the second case the project leader must be asked, who can in turn either deny the request (thereby terminating the workflow) or accept it, in which case the workflow falls back to asking the treasurer for the final confirmation. Upon conclusion, an approval or denial message is sent to the employee.

[1] http://www.cs.wisc.edu/dbworld/
[2] http://maps.google.com/
[3] http://www.flickr.com/

The scenario we just described shows the advantages adaptive processes could bring to daily life. On the one hand, *intra-activity personalization* would allow to configure single process activities according to the user needs and preferences (e.g., by showing a C3R employee only relevant conferences and only hotels satisfying her budget constraints). On the other hand, *inter-activity personalization* would allow to configure the process flow according to the user needs (e.g., by – not – asking the project leader to approve a request according to employee-related information). User-related information can be derived from user's profiles, which can be both those managed locally at the user's company (e.g., to store the number of attended conference and the available budget), or those available at external community-based Web applications (e.g., the user's Linkedin profile storing her skills and research topics).

This paper is organized as follows: Sections 2 and 3 introduce the technologies and tools our solution makes use of, namely, the Cooper process engine and the Protune policy engine respectively. Section 4 describes our reference architecture. We report about related work in Section 5 and conclude in Section 6.

2 The COOPER Process Engine

Cooper is the tool we chose for the definition and execution of processes on the Web. The choice of Cooper is mainly due to its extension mechanisms, which ease the integration of external resources. The most remarkable feature of Cooper is indeed the possibility to integrate libraries of predefined *activity types*, i.e., definitions of activities that are regularly performed by users collaborating in specific domains. Cooper can be easily extended by delegating external modules (from now on *handlers*) to process activities of a given *type*. Since Cooper is a Web platform supporting the execution of processes on the Web, the definition of a new activity type implies associating an activity with a Web front-end, playing the role of the user interface during the execution of the activity, and with a handler in charge of managing the execution of the activity. At run-time, as soon as Cooper starts processing an activity, it checks its type and, if some handler is available for activities of that type, it hands the activity over to that handler. Input parameters and environmental information (like the current user and run-time data) are provided by Cooper to the handler and return parameters are provided back from the handler to Cooper. To some extent, control on the activity flow can be also delegated to handlers, since they can provide Cooper with the activities to be started after they return.

Libraries of activity types are one of the mechanisms used by Cooper to compose sound, "well-structured" processes [11]. In [6] we showed how Cooper's mechanisms for process definition guarantee the semantically correct execution and termination of process instances. This result is especially important since Cooper allows end-users to act as process designers: in particular, they can define new processes by extending process templates or by composing new models from scratch. End-users can also modify template-based process definitions at run-time, as long as the template constraints the process might hold are not violated. Such a flexibility is achieved because each process definition bears all metadata (process structure, role and resource assignments) and run-time data (state of the process and of the activities, execution timestamps) needed for the correct execution. On the other hand, the automatic enforcement of constraints helps (often unexperienced) users during the re-definition and evolution of running processes.

More recently, COOPER has also been extended to enable the definition of *mashup*-like activity types which reuse external services [9]. The mashup paradigm is offered both to process designers who need to introduce new activity types, and to end users who want to customize their activities at run-time by integrating external services which help them to accomplish their tasks. The integration of external services has been made possible by the addition of an activity type (*container activity*), whose handler is in charge of composing and coordinating the execution of the services. This feature further increases the extensibility of COOPER and greatly facilitated its integration with the policy engine and context data.

3 The Protune Policy Engine

According to [16]'s well-known definition, policies are "rules governing the choices in the behavior of a system", i.e., statements which describe which decision the system must take or which actions it must perform according to specific circumstances. *Policy languages* are special-purpose programming languages which allow to specify policies, whereas *policy engines* are software components able to enforce policies expressed in some policy language.

PROTUNE is a framework for specifying and cooperatively enforcing security and privacy policies on the Semantic Web. Protune is based on Datalog and, as such, it is an LP-based policy language (cf. [8]). A PROTUNE program is basically a set of normal logic program *rules* [12] $A \leftarrow L_1, \ldots, L_n$ where $n \geq 0$, A is an *atom* (called the *head* of the rule) and $L_1, \ldots, L_n$ (the *body* of the rule) are *literals*, i.e., $\forall i : 0 \leq i \leq n$ L_i equals either A_i or $\sim A_i$ for some atom A_i. Rules whose body is empty are called *facts.*

Given an atom $p(t_1, \ldots, t_a)$ where $a \geq 0$, p and a are called the *name* and the *arity* respectively of the *predicate* exploited in the atom, whereas $t_1, \ldots, t_a$ are *terms*, i.e., either constants or variables.

With respect to Datalog, PROTUNE presents the following main differences.

Policy language. PROTUNE is a policy language and not a language for data retrieval
Actions. The evaluation of a literal might require to perform actions
Objects. PROTUNE supports objects, i.e., sets of (*attribute*, *value*) pairs linked to an identifier

Being PROTUNE a policy language, its application scenarios are essentially different than the ones of Datalog, which is a language for data retrieval: whoever issues a Datalog query is automatically allowed to retrieve the requested information, whereas in general not everyone issuing a PROTUNE query is allowed to access the requested resource or service. For this reason, whilst the process of evaluating a Datalog query is single-step, the process of evaluating a PROTUNE query involves up to two steps: checking whether the query can be evaluated and, if this is the case, actually evaluating it.

The evaluation of a Datalog (and, more generally, of a Prolog) literal is based on SLDNF-Resolution [12]. Only PROTUNE *logical* literals are evaluated in such a way, whereas the evaluation of PROTUNE *provisional* literals requires to perform an action: the evaluation of a positive (resp. negative) provisional literal is successful if the execution of such action is (resp. is not) successful.

The biggest difference between Datalog and PROTUNE with respect to the built-in data types is that PROTUNE supports *objects*, i.e., sets of (*attribute*, *value*) pairs linked to an identifier. Attributes and values can be objects in turn and attributes can be multi-valued. Objects are referred to by their identifiers, whereas a generic value of the attribute *attr* of an object *id* is denoted by *id.attr*.Finally, a value *val* can be defined for the attribute *attr* of an object *id* by asserting the fact $id.attr = val$.

4 Integrated Architecture for Adaptive Processes

This section describes our reference architecture and in particular illustrates how it enables access to context data (Section 4.1), inter-activity adaptivity (Section 4.2) and intra-activity adaptivity (Section 4.3).

4.1 User Profiles

In order to reduce dependencies among components, we designed an architecture as loosely-coupled as possible. As a consequence, COOPER does not communicate with user profiles directly but only through the PROTUNE engine, which is responsible for each adaptivity issue. Notice that our aim is not to manage the construction of such profiles or provide mechanisms for their integration[4]. Rather, we provide a way to access, according to specific codified rules, attributes that are distributed across different profiles.

Fig. 2 is meant to show the interplay between the PROTUNE engine and the user profiles in our running scenario. It shows the user profiles of two social networking sites as well as the C3R one. The last one is supposed to contain information like the role of C3R employees, the projects they are assigned to and the number of conferences they already attended during the current year, as well as the maximum value traveling expenses can amount to for an employee, and the maximum number of conferences an employee can attend in one year.

Beside the policy engine and the user profiles, Fig. 2 also shows *wrappers*, one for each user profile, and the so called *Meta-wrapper*: wrappers provide clients

[4] User profile integration is still an open issue; the few proposed approaches (see for example [2]) are not widely accepted yet.

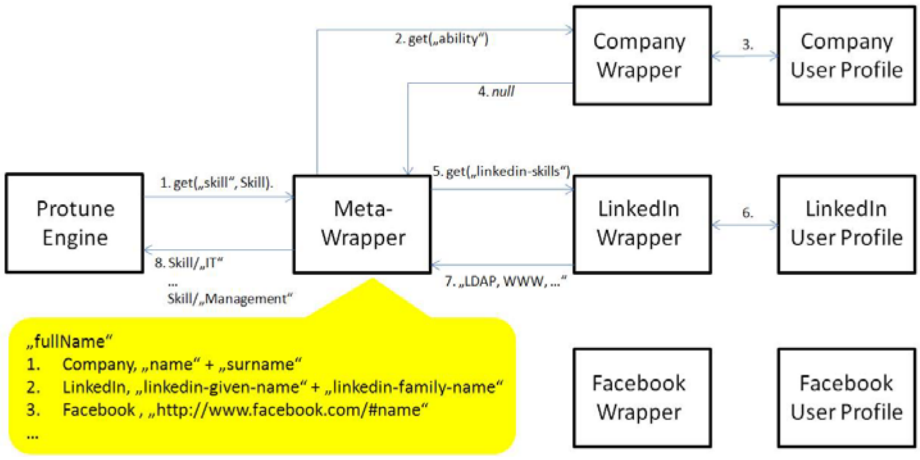

Fig. 2. Interplay between the PROTUNE engine and the user profiles

with an interface `get` which takes as input the (user profile-dependent) name of an attribute to be retrieved in a profile and returns the set of values of that attribute or `null` in case the value is undefined.

The *Meta-wrapper* is the key component responsible for performing the user-profile alignment which is needed in order to merge information coming from different sources. An example of user-profile alignment is provided by the cartoon in Fig. 2: whenever an incoming request asks for the value of the attribute (whose user profile-independent name is) `fullName`, Meta-wrapper first forwards the request to the C3R wrapper. If no result is returned (i.e., if the returned value is `null`), the LinkedIn wrapper is queried. If no result is returned, the Facebook wrapper is queried. The value returned by the Meta-wrapper is the concatenation of the (values of the) C3R user-profile attributes `name` and `surname` (assuming that they are available), otherwise it is the concatenation of the LinkedIn user-profile attributes `linkedin-given-name` and `linkedin-family-name` (assuming that they are available), otherwise it is the Facebook user-profile attributes `http://www.facebook.com/#name`.

As this example shows, the mapping implemented by the Meta-wrapper requires to specify for each user profile-independent attribute: (i) the ordering according to which the available user profiles have to be queried; (ii) the set of user profile-dependent attributes involved in the computation of the (value of the) user profile-independent one; (iii) how the user profile-dependent attributes have to be combined in order to compute the user profile-independent one.

Fig. 2 shows the steps involved in answering a given query issued by the PROTUNE engine to the Meta-wrapper. The interface provided by the latter is the PROTUNE-like predicate `get/2` whose first argument is the name of a user profile-independent attribute and whose second argument is (one of) its value(s). The query `get("skill", Skill).` asks for value(s) of the user profile-independent attribute `skill` (step 1). According to its built-in mapping, the Meta-wrapper is aware of the fact that such user profile-independent attribute maps to: (i) the C3R attribute `ability`—each skill is described in a different `ability` attribute; and (ii) the LinkedIn attribute `linkedin-skills`—all skills are collected into a single comma-separated `linkedin-skills` attribute. Moreover, according to the built-in mapping, the C3R user profile must be queried before the LinkedIn one. For this reason the query `get("ability")` is issued to the C3R wrapper (step 2) and the C3R user profile is accessed (step 3). It happens that no skill information is available in it (step 4), therefore the query `get("linkedin-skills")` is issued to the LinkedIn wrapper (step 5) and the LinkedIn user profile is accessed (step 6). Since skill information is available in it, there is no need to query the Facebook wrapper. The Meta-wrapper works out the results (step 7) and eventually ends up by identifying a set of skills (among which `IT` and `Management`) which are returned one by one to the Protune engine (step 8).

4.2 Inter-activity Adaptivity

As we mentioned in Section 1, in order to enforce inter-activity personalization, context data, and in particular user profiles, have to be taken into account by WfMSs whenever they are about to identify the next activity to be performed. To favor separation of concerns, it is advisable to uncouple user-profile management and control-flow management so that, whilst being the process engine still responsible for advancing the workflow to the next activity to be performed, the task of identifying the upcoming activity itself is delegated to an external component. This section describes an application of this approach to the Cooper.

As we mentioned in Section 2, Cooper's activities are typed and, as soon as COOPER identifies the type of the next activity to be performed, it delegates the handling of the activity to the component responsible for that activity type . New activity types can be defined and corresponding handlers can be registered at COOPER's, so that they will be invoked whenever needed.

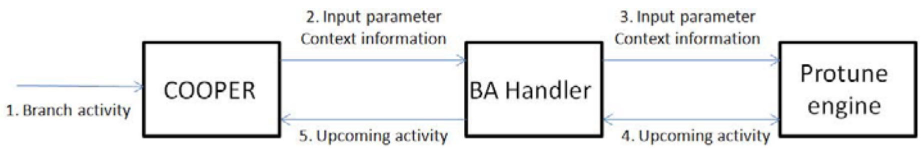

Fig. 3. Extension of COOPER by means of intra-activity policies

We exploited the Cooper's extension mechanism to achieve inter-activity adaptivity. We defined a new handler for branch activities[5] (namely, *branch-activity handler* or *BA-handler*). Fig. 3 shows how a generic branch activity is processed by COOPER extended with our BA-handler. As soon as a branch activity comes in (step 1), it is redirected to the BA-handler, together with

[5] Branching activities in Cooper are *system activities*, which can be used to specify the structure of the process, rather than the single work items. They distinguish from *user activities*, which are instead instantiations of activity types.

input parameters and context information (step 2). The BA-handler acts as a mediator and forwards such information to the PROTUNE engine (step 3). The PROTUNE engine evaluates its inter-activity policy in order to answer the BA-handler's question about the upcoming activity (step 4), which is eventually forwarded back to COOPER (step 5). Policy evaluation might require to use the information forwarded by the BA-handler as well as the one contained in the available user profiles, which are accessed as described in Section 4.

A generic inter-activity policy is a set of rules like the following.

```
allow(getNextActivity(Activity)) ←
    ...
```

Against query `getNextActivity(Activity)` issued by the BA-handler, the PROTUNE engine instantiates the variable `Activity` with the identifier of the activity to be performed next. The dots (...) represent conditions which have to be fulfilled in order for a given instantiation to take place and which might involve information forwarded by the BA-handler as well as user-profile information and, more generally, can be as expressive as described in Section 3.

For instance, the inter-activity policy mentioned in Section 1.1 might look like the following.

```
(1) allow(getNextActivity("a163")) ←
(2)       get("attendedConferences", AttendedConferences),
(3)       get("maximumNumber", MaximumNumber),
(4)       AttendedConferences ≤ MaximumNumber.

(5) allow(getNextActivity("a41")) ←
(6)       ∼ allow(getNextActivity("a163")).
```

The rule at lines 1-4 states that the *Final confirmation* activity (whose identifier is supposed to be `a163`) will be performed next if the number of conferences the current employee already attended in the current year does not exceed the maximum allowed number (line 4). Both numbers are obtained by querying the Meta-wrapper according to the interface described in Section 4 (lines 2-3) and providing as input parameters the names of the metadata introduced in Section 4.1.

The rule at lines 5-6 states that the *Approval request* activity (whose identifier is supposed to be `a41`) will be performed next whenever the evaluation of the preceding rule is not successful, i.e., whenever the PROTUNE engine realizes that the *Final confirmation* activity should not be performed next. This is the case not only if the current employee already attended the maximum allowed number of conferences in the current year, but also if for some reason either of metadata `attendedConferences` and `maximumNumber` cannot be retrieved (i.e., if the evaluation of either of the literals at lines 2-3 is unsuccessful).

4.3 Intra-activity Adaptivity

For each user who has to carry out the activity, intra-activity adaptivity consists of configuring an activity according to the user profile on the basis of some specified policies. As mentioned in section 2, an activity type can also be instantiated through a mashup. If this is the case, the adaptivity policies could

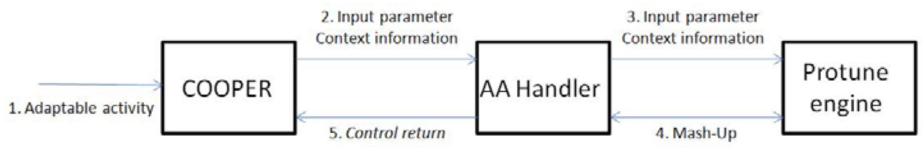

Fig. 4. Extension of COOPER by means of intra-activity policies

also be applied at a finer-grained level, addressing atomic services the mashup is composed of, to describe, for example, which services the mashup should consist of and how they should be configured.

As Fig. 4 shows, for intra-activity adaptivity we adopted a solution that is much similar to the approach we exploited for inter-activity adaptivity depicted in Fig. 3. We defined a new activity type (namely, *adaptable activity*) and a handler for it (namely, *adaptable-activity handler* or *AA-handler*). As soon as an adaptable activity comes in (step 1), it is redirected to the AA-handler, together with input parameters and context information (step 2). The AA-handler forwards such information to the PROTUNE engine (step 3), which uses (among else) such information in order to answer the AA-handler's question about how the activity should be configured (step 4). Differently than the BA-handler described in Section 4.2, the AA-handler does not limit itself to forwarding such information back to COOPER, but is responsible for: (i) configuring the activity based on the information provided by the PROTUNE engine; (ii) coordinating the interaction with the user; and in particular (iii) identifying when the user completed the activity. At this point the control is returned to COOPER (step 5).

A generic intra-activity policy is a set of rules like the following.

(1) **allow(**getContent(Content)**)** $\leftarrow$
(2) $\quad\ldots$

(3) content.$name$:$value$ $\leftarrow$
(4) $\quad\ldots$

Against query `getContent(Content)` issued by the AA-handler, the PROTUNE engine instantiates the variable `Content` with the information needed to configure the activity. Whilst in Section 4.2 the information to be returned (namely, the identifier of the activity to be performed next) could be modeled as an atomic entity (specifically, a PROTUNE string), the configuration of an adaptable activity can be modeled in the most natural way as a structured object (specifically, a PROTUNE object – cf. Section 3). However, as described in Section 3, before returning a PROTUNE object, the PROTUNE engine collects all $(name, value)$ pairs of its, and the process is iterated in case either of $name$ and $value$ is an object in turn. For this reason, before returning the PROTUNE object identified by the label `content`, the PROTUNE engine collects all the pairs it consists of, i.e., all pairs $(name, value)$ such that the PROTUNE atom `content.`$name$`:`$value$ holds. Notice that this requires to evaluate conditions in line 4, which do not conceptually differ from the ones in line 2 or the ones described in Section 4.2.

For instance, (a fragment of) the intra-activity policy mentioned in Section 1.1 might look like the following.

```
(1)    allow(getContent(mashUp)).

(2)    mashUp.“application”:dbWorld ←
(3)        currentActivity(“a532”).
(4)    mashUp.“application”:googleMaps ←
(5)        currentActivity(“a532”).
(6)    mashUp.“application”:flickr ←
(7)        currentActivity(“a532”).

(8)    dbWorld.“conference”:Conference ←
(9)        get(“skill”, Skill),
(10)       getConferenceByTopic(Skill, Conference).
(11)   dbWorld.“conference”:Conference ←
(12)       ∼ get(“skill”, Skill),
(13)       getConference(Conference).
...
```

This policy makes use of the following provisional predicates (cf. Section 3).

currentActivity/1. Part of the interface to the context information available to the AA-handler: it checks the identifier of the current activity (lines 3, 5, 7)

get/2. The interface to the Meta-wrapper described in Section 4 (lines 9, 12)

getConferenceByTopic/2. Part of the interface to the DBWorld RSS feeds: it retrieves the conferences related to a given topic (line 10)

getConference/1. Part of the interface to the DBWorld APIs: it retrieves the conferences listed on DBWorld (line 13)

The rules at lines 2-7 state that, if the current activity is *Select conference and hotel* (whose identifier is supposed to be `a532`), it will show a mashup consisting of the following applications: DBWorld (represented by the PROTUNE object `dbWorld` – line 2), Google Maps (represented by `googleMaps` – line 4) and Flickr (represented by `flickr` – line 6). The rules at lines 8-13 then configure the DBWorld application by listing the conferences it should show (represented by the values of the `conference` attribute of `dbWorld` – lines 8, 11). In particular, the rule at lines 8-10 states that, if the skills of the current employee are available (line 9), only conferences relevant to them should be shown (line 10). On the other hand, the rule at lines 11-13 states that, if the skills are not available (line 12), all conferences should be shown (line 13).

Notice that the adaptation of component services (e.g., DBWorld) within a mashup-based instantiated activity is possible thanks to the availability of *mashup descriptors*, specifying both the characteristics of single components (e.g., the way they can be accessed), both the way they are integrated and executed in the mashup. For more details on such descriptors and the mashup execution logic they are able to support the reader is referred to [17].

5 Related Work

In order to support changing and non-repetitive processes, which are typical in collaboration scenarios, the main efforts have been done in the field of communication support systems or *Groupware* (see [13] for a survey). These applications support the execution of individual tasks, especially based on asynchronous activities (e.g., sending emails), but offer very limited support to structuring tasks into process flows sustaining collaboration. WfMSs can in principle support the design of collaborative processes. However, they are too rigid to support the variable nature of such processes [7].

Positioned between the two extremes are *evolving workflows*. The authors of [15] identify *adaptability* as a dimension characterizing workflow evolution, and define it as the ability of a process to react to exceptional situations. In literature, this dimension is mainly addressed by mechanisms for exception handling that extend workflow definition languages with new and more complex control constructs [1]. The main disadvantage of such approaches is that the new languages are proprietary solutions lacking portability and whose execution engines are complex to implement. A general drawback of this paradigm is also the complexity of foreseeing at design time the whole set of exceptions that could occur at execution time [10]. Since the adaptation logic is built in the process design, dealing with non planned exceptions is not trivial: the process specification must be changed, and this in turn compromises the compliance of the already active workflow instances, enacted according to the original model, with the new process specification [5,14].

The approach proposed in this paper tries to overcome the drawbacks mentioned above by targeting *adaptivity*, i.e., the capability of a process to react at run-time in response to context parameters that cannot be adequately fixed in the workflow definition, such as properties of (evolving) user profiles. To the best of our knowledge, this issue has been scarcely investigated in literature, being process adaptivity often confused with process flexibility and adaptability [10]. The solution described in this paper allows us to introduce adaptivity features without affecting the process models. Moreover, context data and adaptivity policies can be managed separately from the process deployment, thereby easing the evolution of adaptivity requirements as well as of the process itself.

6 Conclusions and Further Work

This paper has presented an integrated framework where flexibility of Web collaborative processes is enhanced through run-time adaptivity with respect to context data. Collaborative processes are highly characterized by the need of adapting the process and the execution of single tasks to the variability of the needs, preferences and background of the involved actors. This need is even more accentuated if we consider that several sources of context data, and in particular user profiles, are available on the Web and can be exploited as a solid basis to provide adaptivity. We have therefore experimented the integration of the Cooper process engine, initially conceived to support flexible and dynamically defined collaborative processes on the Web, with an external policy manager, Protune, in charge of accessing heterogenous sources of context data and executing adaptivity policies. Although we adopted such two specific platforms, the integration approach described in this paper is general in its nature, and can be easily replicated in the context of different methodologies for flexible processes and policy management. The integration logics, indeed, is mainly based on the adoption of mediators, the so-called *handlers*, in charge of managing the interaction between the process engine and the policy engine, as well as the access to heterogeneous sources of context data.

An initial prototype has allowed us to prove the feasibility of our approach. Our future work is devoted to fully implementing the integration, and to testing its effectiveness and efficiency. This activity will also imply the definition of pre-defined *adaptive activity types* and of related adaptivity policies, as well as of policy-enhanced process templates covering the most frequent adaptivity requirements. We will also investigate the possibility that the users personalize the *Meta-wrapper*, for example giving them the freedom to select their most preferred user profiles and priorities for each profile.

References

1. van der Aalst, W.M.P., ter Hofstede, A.H.M., Kiepuszewski, B., Barros, A.P.: Workflow patterns. Distributed and Parallel Databases 14(1), 5–51 (2003)
2. Abel, F., Heckmann, D., Herder, E., Hidders, J., Houben, G.J., Krause, D., Leonardi, E., van der Slujis, K.: A framework for flexible user profile mashups. In: AP-WEB, pp. 1–10 (2009)
3. Bonatti, P.A., Olmedilla, D.: Driving and monitoring provisional trust negotiation with metapolicies. In: POLICY, pp. 14–23 (2005)
4. Bonatti, P.A., Olmedilla, D., Peer, J.: Advanced policy explanations on the web. In: ECAI, pp. 200–204 (2006)
5. Casati, F., Ceri, S., Pernici, B., Pozzi, G.: Workflow evolution. Data Knowl. Eng. 24(3), 211–238 (1998)
6. Ceri, S., Daniel, F., Matera, M., Raffio, A.: Providing flexible process support to project-centered learning. IEEE Trans. Knowl. Data Eng. 21(6), 894–909 (2009)
7. Charoy, F., Guabtni, A., Faura, M.V.: A dynamic workflow management system for coordination of cooperative activities. In: Eder, J., Dustdar, S. (eds.) BPM Workshops 2006. LNCS, vol. 4103, pp. 205–216. Springer, Heidelberg (2006)
8. De Coi, J.L., Olmedilla, D.: A review of trust management, security and privacy policy languages. In: SECRYPT, pp. 483–490 (2008)
9. Fisichella, M., Matera, M.: Process Flexibility trough Customizable Activity Types: a Mashup-based Approach. Submitted for publication (2010)
10. Kammer, P.J., Bolcer, G.A., Taylor, R.N., Hitomi, A.S., Bergman, M.: Techniques for supporting dynamic and adaptive workflow. Computer Supported Cooperative Work 9(3/4), 269–292 (2000)
11. Kiepuszewski, B., ter Hofstede, A.H.M., Bussler, C.: On structured workflow modelling. In: Wangler, B., Bergman, L.D. (eds.) CAiSE 2000. LNCS, vol. 1789, pp. 431–445. Springer, Heidelberg (2000)
12. Lloyd, J.W.: Foundations of Logic Programming, 2nd edn. Springer, Heidelberg (1987)
13. Rama, J., Bishop, J.: A survey and comparison of CSCW groupware applications. In: SAICSIT 2006, pp. 198–205. Republic of South Africa (2006)

14. Sadiq, S.W.: Handling dynamic schema change in process models. In: Australasian Database Conference, pp. 120–126 (2000)
15. Sadiq, S.W., Orlowska, M.E., Sadiq, W.: Specification and validation of process constraints for flexible workflows. Inf. Syst. 30(5), 349–378 (2005)
16. Sloman, M.: Policy driven management for distributed systems. J. Network Syst. Manage. 2(4) (1994)
17. Yu, J., Benatallah, B., Saint-Paul, R., Casati, F., Daniel, F., Matera, M.: A framework for rapid integration of presentation components. In: Williamson, C.L., Zurko, M.E., Patel-Schneider, P.F., Shenoy, P.J. (eds.) WWW, pp. 923–932. ACM, New York (2007)