IMPROVING FLEXIBILITY OF WORKFLOW MANAGEMENT SYSTEMS VIA A POLICY-ENHANCED COLLABORATIVE FRAMEWORK

Alfredo Cuzzocrea[1] and Marco Fisichella[2]
[1]*ICAR-CNR and University of Calabria, Cosenza, Italy*
[2]*L3S Research Center, Hannover, Germany*

Keywords: Workflow Management Systems, Process Intelligence, Collaborative Processes.

Abstract: Workflow Management Systems available to date present limitations concerning the flexibility of the resulting processes, which enlarge the gap between business processes models, humans and their activities. In real-world scenarios, especially when collaboration among human actors takes place, it is often the case that the tasks a process consists of appear in many subtle variants according to the actor who has to carry them out. In this paper we focus on a system which integrates Web platform for collaborative processes, COOPER, and a policy manager based on Semantic Web technologies, PROTUNE, can enhance process flexibility and variability to accommodate the needs of different process actors.

1 INTRODUCTION

Workflow Management Systems (WfMSs) support the definition and execution of business processes by combining atomic activities, each one characterized by its own set of input and output resources and parameters, as well as of actors who are enrolled to carry them out. The output of an activity is (part of) the input of the activity (or activities) immediately following it. The possibility of specifying which actor(s) have to carry out an activity enforces coordination among them.

WfMSs available to date present limitations concerning their flexibility, which enlarge the gap between business processes models, humans and their activities (Bardram et al., 2006). In real-world scenarios it is often the case that the tasks a process consists of appear in many subtle variants according to the actor who has to carry them out. This is especially true in all those organizational contexts where collaboration within teams requires people to coordinate by means of “light-weight” processes that allow them to reach a given goal (Bardram et al., 2006; Nodenot et al., 2004), e.g., the release of some artifacts. One can think for example to the development of team-based projects within a company, but also to the so-called ”learning-by-doing”, an increasingly diffused form of learning through which individuals (e.g., students in university, employees in organizations in different domains, company partners and customers, etc.) learn by working on a project and sharing activities with others peers. In all the previous cases, processes exhibit an explosive number of alternatives that depend on the specific needs of the involved actors. Such alternatives may refer to the coordination flow of the different tasks, as well as to the execution of single tasks. Flexibility might be required for example to enlarge or restrict the user access to some services or contents, based on the users’ rights on the company information asset, their maturity level in performing some activities, the evolution of background knowledge and competencies, and so on. Such processes therefore escape the ability of being fully modeled at design time, as it happens for traditional business processes (Bardram et al., 2006), and should be flexible enough to be adapted to the needs of the individual actors. At a larger vision, flexibility may also be intended as the capability of a system to automatically discovery (and integrate) new services/processes via intelligent methodologies (e.g., (Cuzzocrea et al., 2011; Cuzzocrea and Fisichella, 2011)).

In this paper we focus on a system based on policies which can be used to overcome these limitations. According to (Sloman, 1994)’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

Cuzzocrea A. and Fisichella M..
IMPROVING FLEXIBILITY OF WORKFLOW MANAGEMENT SYSTEMS VIA A POLICY-ENHANCED COLLABORATIVE FRAMEWORK.
DOI: 10.5220/0003937507010708
In *Proceedings of the 8th International Conference on Web Information Systems and Technologies* (WEBIST-2012), pages 701-708
ISBN: 978-989-8565-08-2
Copyright © 2012 SCITEPRESS (Science and Technology Publications, Lda.)

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.

Process engines and policy engines can be considered to be complementary tools to enhance process flexibility: process engines allow one to define and execute processes, but they fall short whenever process modifications have to occur during the execution of the process (Fisichella et al., 2011). On the other hand, being based on decision making, it appears natural combining policy and process engines in order to overcome the limitations mentioned above. Policies could be for example exploited within a process task in order to adapt it to the end-users' specific needs and access rights (*intra-activity policies*). On the other hand, policies could be exploited at the level of the overall process in order to determine the activity/ies to be next performed based on the outcome of the preceding activity/ies (*inter-activity policies*).

This paper focus on a system leveraging on both approaches applied to COOPER (Ceri et al., 2009), a Web platform for the definition and execution of collaborative processes, characterized by a reference model that ensures process flexibility, effectively supporting the dynamic, user-based management of collaboration processes. We in particular investigate the exploitation of PROTUNE (Coi and Olmedilla, 2008) for the specification and management of policies. PROTUNE is a policy framework based on Semantic Web technologies, that supports the creation and enforcement of advanced policies, covering not only traditional access control but also trust negotiation (to automate for example privacy-aware information release), and second generation explanation facilities (to improve user awareness about – and control on – policies).

2 THE PROTUNE POLICY FRAMEWORK

Policies are encountered in many situations of our daily life. Especially with the advent of the digital era, the specification of policies has emerged in many web-related contexts and software systems. E-mail client filters are a typical example of policies. Some of the main application areas where policies have been lately used are security and privacy as well as specific business domains, where they take on the name of "business rules" (but (Li et al., 2006), (Li and Wang, 2006) and (Stoller et al., 2007) for other policies' application areas). In general, policies are a well-known approach to protecting security and privacy of users in the context of the Semantic Web: policies specify who is allowed to perform which action on which object depending on properties of the requester and of the object as well as parameters of the action and environmental factors (e.g., time). The application of suitable policies for protecting services and sensitive data may determine success or failure of a new service.

The use of formal policies yields many advantages compared to conventional approaches: formal policies are usually dynamic, declarative, have a well-defined semantics and allow to be reasoned over (Lloyd, 1987) and (Sloman, 1994). 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 (Coi and Olmedilla, 2008). A PROTUNE program is basically a set of normal logic program *rules* (Lloyd, 1987) $A \leftarrow L_1, \ldots, L_n$ 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: (i) *policy language* – PROTUNE is a policy language and not a language for data retrieval; (ii) *actions* – the evaluation of a literal might require to perform actions; (iii) *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: whilst a Datalog query is meant to retrieve (explicit or implicit) information from a knowledge base, the typical PROTUNE query is a request by a human or software agent to access some resource or service. Moreover, 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 (Lloyd, 1987): the evaluation of a negative literal (i.e., of a literal of the form *not A*, where *A* is an atom) is (un)successful if it is (not) the case that the evaluation of the body of all rules whose head matches *A* is unsuccessful. The evaluation of a positive literal (i.e., of a literal of the form *A*, where *A* is an atom) is (un)successful if it is (not) the case that the evaluation of the body of some rule whose head matches *A* is successful. 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 PROTUNE engine univocally identifies the action to be performed based on the predicate exploited in the literal. The name and arity of the predicate are interpreted as an URL pointing to a software component (a `jar` file). By need such component is automatically resolved, deployed and finally exploited to actually perform the action.

Beside minor things (PROTUNE constants can be booleans and numbers as well) 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 $id.attr$, according to the well-known Java-like dot notation. Finally, a value *val* can be defined for the attribute $attr$ of an object id by asserting the fact $id.attr = val$.

We conclude by briefly mentioning further features of PROTUNE, namely the distributed query evaluation process and the explanation facility. In Datalog the evaluation of a query and, more generally, of a literal is a local process: only the local knowledge base (i.e., the Datalog program) is inspected in order to retrieve the requested information. On the contrary, the author of a PROTUNE policy can state that some literal has to be evaluated by the requester: if during the evaluation of a query Q_1 issued by a requester R_1 it happens that such literal has to be evaluated, a counter-query Q_2 is issued to R_1 with the request to evaluate it. Counter-queries can nest to an arbitrary depth, so that the evaluation of a PROTUNE query can evolve to a *negotiation* between the actors involved. In order to make explicit that a literal has to be evaluated by the requester, the policy author must tag that literal with an annotation which in PROTUNE's jargon is called *metarule*.

Metarules are also used to support PROTUNE's explanation facility: explanation metarules define verbalization patterns which are instantiated by the explanation facility to produce a human-readable description of the policy. Such description is then exploited to answer a predefined set of user queries. So far four kinds of queries are supported: *How-to* queries describe which activities the requester has to carry out in order to get access to a given resource or service. *What-if* queries are meant to help requesters to foresee the evolution of a hypothetical situation. Finally, *Why* and *Why-not* queries merge the description of the policy with transaction-dependent information in order to explain the outcome of a transaction which already occurred.

3 THE COOPER PLATFORM

This Section introduces COOPER, the engine we chose to use for process definition and execution. The choice of COOPER is due to the flexibility that it is able to support at process level, which is possible thanks to the easy integration of ”externally” defined process tasks. The most salient feature of COOPER is that it allows cooperating team members (i.e., the platform end-users) to dynamically define collaborative processes, and easily modify planned processes, to cope with the evolution of individuals as well as of the whole team. Giving the end-users the possibility to define and modify their processes requires the system to offer easy-to-use definition interfaces, able to guide people in the composition of processes without requiring any specific knowledge and expertise on process design. Guiding inexperienced users requires that the system be “aware” of the semantics of the domain where processes must be executed. Such awareness can be achieved through libraries of pre-defined *activity types*, able to reflect and support the possible tasks that users might need to coordinate and execute in a given context.

Activity types represent the definition of process tasks, that are regularly performed by users to collaborate, and that can be used within collaborative processes. Their definition implies associating the underlying task with a hypertext front-end, which is used as user interface for the execution of the activity. Some atomic activities have a general nature (e.g., those related to the management of documents), and can therefore be adopted in several domains where collaborative processes are required. Some other activities may however be particular for specific contexts and their identification requires an investigation of the addressed domain. Starting from a library of activity types, the system thus guides the composition of sound, “well-structured” processes (Kiepuszewski et al., 2000). In (Ceri et al., 2009) we show how the offered mechanisms for process definition guarantee the semantically correct execution and termination of process instances, and the possibility to easily (flexibly) modify processes even during runtime. Providing guarantees on the process semantics aims at assisting the continuous re-definition or evolution of running processes by users that in most cases are inexperienced.

Figure 1 shows the architecture of the COOPER platform, where different modules interoperate and make use of data and metadata. Process definition is performed via a process editor, a Web front-end that makes use of a predefined activity type library and, possibly, of existing process/template models. Process execution is performed via the COOPER's collaboration environment, which leverages on the hypertext front-ends of the predefined activity types to allow users to produce and consume process data in form of resources stored in the resource repository. Process advancement is then governed by the stored process definitions, which are interpreted during process execution by a dedicated process engine that contains the necessary application logic to maintain the running processes' metadata and, hence, to drive the activity flow in the collaboration environment.

The COOPER approach overlaps the ideas of process model and process instance: each process definition has only one executing instance or, the other way around, each process execution has its own process definition. A process definition bears all metadata (process structure, role assignments, resource assignments) and runtime data (state of the process, states of activities, execution time stamps) necessary for the correct execution. This allow us to support process modification operations.

Figure 1 also highlights the competences of the individual actors in the COOPER platform. The *activity designer* identifies and designs the activity types that are available in the platform. The *process designer* then instantiate the activities types and defines process templates. The process designer can also be a team leader who wants to organize the wok of his/her team. The *users* then perform the actual work. Users can also play the role of process designers. They are indeed enabled to define new processes by extending templates, or by composing new models from scratch. They are in any case allowed to modify template-based process definitions during runtime, after the process has been launched, with the only limit of not violating the template constraints that the process may hold by definition.

Based on this architecture, COOPER can be extended by *delegating* external modules (from now on *handlers*) to process given activities. Each handler is responsible for activities of a given *type*. Policies with the same type have the same input/output parameters. As soon as COOPER starts processing a new activity, it checks its type and, if some handler is available for activities of that type, it hands the activity over to such handler. Input parameters and context information (like the current user) are provided by COOPER to the handler and return parameters are provided back from the handler to COOPER. To some extent, control on the workflow itself can be delegated to handlers, since they can provide COOPER with the activities to be started after they return.

Such extension point (Birsan, 2005) of COOPER will be exploited for the approaches presented in Section 5 and Section 6.

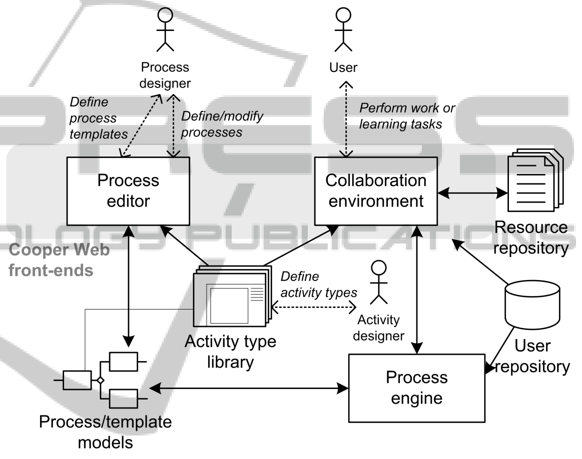


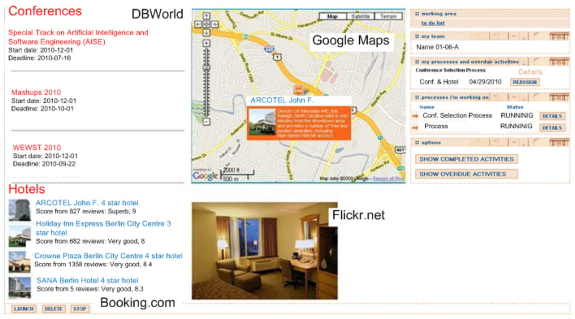
Figure 1: The architecture of the COOPER platform (Ceri et al., 2009).

4 APPLICATION SCENARIOS

To better understand our approach, we will consider two simple scenarios that require the collaboration of multiple actors.

4.1 Scenario I

Imagine a team leader wants to suggest to team components some events e.g., conferences to attend. Team components select one conference and search for a hotel in the event location. Later, they ask the organization treasurer for the monetary availability. The process is composed by three activities: (i) an activity is related to the team leader who has to suggest a listing of conferences to attend; (ii) an activity associated to the employee, who must select the conference and book a hotel; and (iii) an activity associated to the treasurer, who has to check the cash and allocate, if available, the amount request. The treasurer will conclude the activity with a confirmation or negation



Special Track on Artificial Intelligence and Software Engineering (AISE)
Start date: 2010-12-01
Deadline: 2010-07-16

Mashups 2010
Start date: 2010-12-01
Deadline: 2010-10-01

WEWST 2010
Start date: 2010-12-01
Deadline: 2010-09-22

ARCOTEL John F. 4 star hotel
Score from 827 reviews: Superb, 9

Holiday Inn Express Berlin City Centre 3 star hotel
Score from 682 reviews: Very good, 8

Crowne Plaza Berlin City Centre 4 star hotel
Score from 1358 reviews: Very good, 8.4

SANA Berlin Hotel 4 star hotel
Score from 5 reviews: Very good, 8.3

Name 01-06-A

Conference Selection Process
Conf. & Hotel 04/29/2010

Name	Status
Conf. Selection Process	RUNNINIG
Process	RUNNINIG

Figure 2: Example of Mashup encapsulated within the process flow according to the definition of an activity supported by four components.

message sent to the employee.

Suppose that the definition of this process requires the inclusion of an activity for the process of booking the hotel and that such activity can be supported by four components as shown in Figure 2. We suppose these four components are already registered in the platform: the service "Find a Conference" supplying the list of the conferences indexed by DBWorld, the service "Hotel Retrieve" supplying a list of hotels for a given place of interest, the Flickr.NET component to display the images for a given hotel, and Google Maps to show the directions to a given address or point of interest. When the team leader selects a conference from the conference listing component, the hotel listing will be updated and will display all the hotels in that area, the image displayer will show an image of the selected hotel while the map will display its location. In other words, *the way* the employees' activity is performed depends on the outcome of the first one (namely, which conference the team leader opted for). In such a scenario, policies can help to adapt the first activity to the end user's need or right, e.g. some team leader could select only conferences which will be organized in places within the same continent or below a kilometric distance threshold from his/her own company location.

4.2 Scenario II

Imagine a team working in a business project that requires the involvement of professionals from different fields. A first version project proposal must be approved by the team leader at certain point of time. After that, there must be: (i) a proper description of the necessary technical work; (ii) a description of the material and people involved; (iii) a description of costs; and (iv) a description of the marketing planning. After these activities, a new activity is assigned to the team leader to review all the documentation to continue the project development.

Suppose that the definition of the activity is a mashup utilizing Google Docs API[1], allowing document editing and versioning control. The same activity can be reused for the four activities previously listed if the necessary actions are the same; however each activity refers to a very specific group of team members. So it is up to the team leader to approve the proposal in the first instance. Afterwards the process will split into four different parallel sub-processes. Continuing in each activity, developers would document the technical issues, technicians and personal would describe the necessary material and people, treasurer would calculate the finances and the marketing members would describe market strategies. Once each team has finished each respective assignment, the workflow joints back together to an approval activity for the team leader. Notice that *whether* the four parallel activities are performed depends on the outcome of the first one (namely, whether the team leader approved the proposal).

5 INTER-ACTIVITY POLICIES

As we argued in Section 4, in many real-world workflows it is often the case that: (i) the way an activity is performed and (ii) whether an activity is at all performed depends on the outcome of the activity/ies previously performed. Furthermore, the possibility to select which next activity will be executed based on some conditions or output from previous activity/ies has been previously solved by other process engine, but in this Section we concentrate our treatment about how to solve this problem exploiting the policies, such that we demonstrate to provide a complete process engine based on policies.

The exchange of input parameters and return values described in Section 3 is a suitable mechanism provided by COOPER to determine the way an activity has to be performed based on the outcome of the one(s) previously performed: with respect to the scenario described in Section 4.1, the activity of the employees could be modeled as a parameterized COOPER activity whose execution depends on the value of a parameter which is meant to represent the list of conferences to attend. The instantiation of this parameter could be provided by the COOPER activity modeling the team leader's activity.

On the other hand, as we mentioned in Section 1, COOPER does not natively provide any means to identify whether an activity should be performed based on the outcome of the activity/ies pre-

[1]http://code.google.com/apis/documents/overview.html

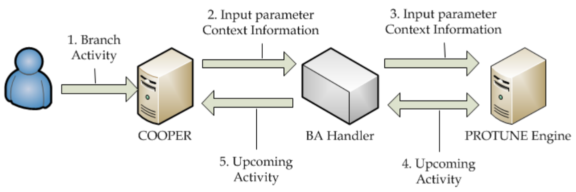


Figure 3: Extension of COOPER by means of inter-activity policies.

viously performed. For this reason we decided to exploit COOPER's extension mechanism described in Section refsec:cooper in order to add branches to COOPER workflows: we defined a new activity type (namely, *branch activity*) and an handler for them (namely, *branch-activity handler* or *BA-handler*).

Fig. 3 shows how a generic branch activity is processed by COOPER extended with our BA-handler. According to COOPER's generic extension mechanism (see Section 3), as soon as a branch activity comes in (1), it is redirected to the BA-handler, together with input parameters and context information (2). The BA-handler acts as a mediator and forwards such information to the PROTUNE engine (3). The PROTUNE Engine uses (among else) such information in order to answer the BA-handler's question about the upcoming activity (4), which is eventually forwarded back to COOPER (5).

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

allow(*getNextActivity*(Activity)) ←
  conditions()

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. *Conditions* have to be fulfilled in order for the given instantiation to take place. As described in Section 2, the PROTUNE policy language allows to define expressive condition involving, among else, properties of the current user, activity and environmental properties (e.g., time).

6 INTRA-ACTIVITY POLICIES

As we argued in Section 1 and Section 4, in real-world scenarios it is often the case that the atomic activities a business process consists of differ according to the actor who has to carry them out. Therefore it is needed: (i) providing workflow designers with the ability to specify variants in the definition of an activity; and (ii) enforcing at run-time the right variant according to the actor who is carrying it out.

A trivial solution would be adopting the approach we discussed in Section 5 to this case as well: each variant of an activity could be modeled as an activity itself and we could identify the right variant at run-time by resorting to inter-activity policies. Although this solution is in principle feasible, it has at least a couple of drawbacks: it is unnatural and (hence) user-unfriendly.

For this reason it makes sense trying to identify a different mechanism to specify and enforce fine-grained activity tuning. We chose to use the Mash-Up technology.

A mashup combines services coming from heterogeneous sources not initially conceived to coexist together. Especially if supported by an easy-to-use development tool, the mashup paradigm can favor the on-the-fly construction of applications that quickly allow users to have insights and make decisions based on the resulting combined data. Within the COOPER framework, mashups can certainly facilitate the definition and integration of new activity types by activity designers. However, the most tangible advantage is that they can enable process designers to create on-the-fly new activity types during process definition, as well as process actors to easily customize already defined activities during process execution. As explained in the following, after its definition, the mashup can be integrated as an activity in a process. A process will be able to incorporate one or more mashups, even different versions of a same mashup.

To achieve the goal of integrating mashups within process definition, we extend the activity type library, as defined in COOPER, with a new type, the container activity (CA), based on two main concepts: (1) as any other activity type, it exposes some properties that are required for its integration within a process (e.g., start and end time, enrolled process actors, etc.), and that must be set during process definition; (2) it is a generic container, aimed at encapsulating externally defined web resources.

Moreover, a CA can be associated with several mashups, even different versions of a same mashup, each of them associated with a particular policy to be satisfied. While defining a process, one or more CAs can be used when the ready-to-use activity types do not match the requirements of some user activities. The process designer can then define a new service in a mashup wise (see next) and encapsulate it within the container. To enable the integration of this new service within the process flow, s/he can characterize the output resource to be produced. Leveraging the template-based paradigm and the flexibility of COOPER, the encapsulation of the actual application supporting the activity execution can even be delayed at run-time: the process designer just configures the container properties that are necessary for process en-

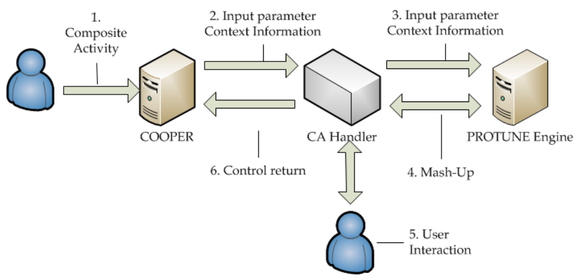


Figure 4: Extension of COOPER by means of intra-activity policies.

actment, e.g., the enrolled user, the resources to be produced as activity output, and so on. The actual mashup for the activity execution can be defined and encapsulated during the process execution by the enrolled actor.

Fig. 4 shows how a generic container activity is processed by COOPER extended with our CA-handler. According to COOPER's generic extension mechanism (see Section 3), as soon as a container activity comes in (1), it is redirected to the CA-handler, together with input parameters and context information (2). The CA-handler forwards such information to the PROTUNE engine (3). The PROTUNE Engine uses (among else) such information in order to answer the CA-handler's question about which mash-up should be used (4). Differently than the BA-handler described in Section 5, the CA-handler does not limit itself to forwarding such information back to COOPER, but is responsible for: (i) retrieving the mash-up based on the information provided by the PROTUNE engine; (ii) coordinating the interaction of such mash-up with the user; and in particular (iii) identifying when the user completed the mash-up activity (5). At this point the control is returned to COOPER (6).

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

allow(*getMashUp*(MashUp)) ←
  conditions()

Against query $getMashUp(MashUp)$ issued by the CA-handler, the PROTUNE engine instantiates the variable $MashUp$ with the identifier of the mash-up to be provided to the user. Again, $Conditions$ have to be fulfilled in order for the given instantiation to take place.

7 CONCLUSIONS

In this paper, we have presented an integrated framework where the definition of collaborative processes is augmented through policies aimed at enhancing process flexibility to enlarge or restrict the user access to services or contents. Collaborative processes are highly characterized by the need of adapting the process flow and the execution of single tasks to the variability of the needs, access rights and background of the involved actors. To respond to this need, the COOPER platform has been specifically conceived to empower end-users to self-define and modify processes through an easy-to-use Web environment. However, COOPER still shows some limits, especially related to the customization of task instances targeting individual process actors. This paper has shown how such limits can be overcome thanks to the adoption of the PROTUNE policy manager, still ensuring the full interoperability with COOPER as to easy the global integration task.

An initial prototype has allowed us to prove the feasibility of the concepts illustrated in this paper. 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 a library of pre-defined and reusable policies, expressing frequent rules for adaptation and/or access restriction, as well as of policy-enhanced process templates covering the most frequent collaboration requirements.

ACKNOWLEDGEMENTS

This research has been co-funded by the European Commission within the eContentplus targeted project OpenScout, grant ECP 2008 EDU 428016 (http://www.openscout.net).

REFERENCES

Bardram, J. E., Bunde-Pedersen, J., and Søgaard, M. (2006). Support for activity-based computing in a personal computing operating system. In Grinter, R. E., Rodden, T., Aoki, P. M., Cutrell, E., Jeffries, R., and Olson, G. M., editors, *CHI*, pages 211–220. ACM.

Birsan, D. (2005). On plug-ins and extensible architectures. *ACM Queue*, 3(2):40–46.

Ceri, S., Daniel, F., Matera, M., and Raffio, A. (2009). Providing flexible process support to project-centered learning. *IEEE Trans. Knowl. Data Eng.*, 21(6):894–909.

Coi, J. L. D. and Olmedilla, D. (2008). A review of trust management, security and privacy policy languages. In *SECRYPT*, pages 483–490.

Cuzzocrea, A., De Coi, J.L., Fisichella, M., and Matera, M. (2011). Graph-based matching of composite OWL-S services. In *DASFAA Workshops*, pages 28–39.

Cuzzocrea, A. and Fisichella, M. (2011). Discovering semantic Web services via advanced graph-based matching. In *IEEE SMC*, pages 608–615.

Fisichella, M. and Matera, M. (2011). Process flexibility through customizable activities: A mashup-based approach. In *ICDE Workshops*, pages 226–231.

Kiepuszewski, B., ter Hofstede, A. H. M., and Bussler, C. (2000). On structured workflow modelling. In Wangler, B. and Bergman, L., editors, *CAiSE*, volume 1789 of *Lecture Notes in Computer Science*, pages 431–445. Springer.

Li, N., Tripunitara, M. V., and Wang, Q. (2006). Resiliency policies in access control. In *ACM Conference on Computer and Communications Security*, pages 113–123.

Li, N. and Wang, Q. (2006). Beyond separation of duty: an algebra for specifying high-level security policies. In *ACM Conference on Computer and Communications Security*, pages 356–369.

Lloyd, J. W. (1987). *Foundations of Logic Programming, 2nd Edition*. Springer.

Nodenot, T., Marquesuzaà, C., Laforcade, P., and Sallaberry, C. (2004). Model based engineering of learning situations for adaptive web based educational systems. In Feldman, S. I., Uretsky, M., Najork, M., and Wills, C. E., editors, *WWW (Alternate Track Papers & Posters)*, pages 94–103. ACM.

Sloman, M. (1994). Policy driven management for distributed systems. *J. Network Syst. Manage.*, 2(4).

Stoller, S. D., Yang, P., Ramakrishnan, C. R., and Gofman, M. I. (2007). Efficient policy analysis for administrative role based access control. In *ACM Conference on Computer and Communications Security*, pages 445–455.