StPowla: SOA, Policies and Workflows

S. Gorton[1], C. Montangero[2], S. Reiff-Marganiec[1], and L. Semini[2]

[1] Department of Computer Science, University of Leicester
[2] Dipartimento di Informatica, Università di Pisa

Abstract. We introduce StPowla, a workflow based approach to business process modelling that integrates a simple graphical notation, to ease the presentation of the core business process, a natural policy language, Appel, to provide the necessary adaptation to the varied expectations of the various business stakeholders, and the Service Oriented Architecture, to assemble and orchestrate available services in the business process. We illustrate the approach with a loan approval process.

1 Introduction

The integration of Business Process Management (BPM) and SOA is a promising solution to the design and development of the software systems of the future, as clearly stated in [13]: " [..] The BPM-SOA combination allows services to be used as reusable components that can be orchestrated to support the needs of dynamic business processes. The combination enables businesses to iteratively design and optimize business processes that are based on services that can be changed quickly, instead of being 'hard-wired'. This has the potential to lead to increased agility, more transparency, lower development and maintenance costs and a better alignment between business and IT."

Besides, there is an obvious need for flexibility in modelling business process. Flexibility permits to customize a core model to adapt it to various requirements, and to accommodate the variability of a business domain.

To cope with flexibility and variability, we define StPowla – to be read like "Saint Paula" – a Service-Targeted Policy-Oriented WorkfLow Approach. StPowla is designed to support policy-driven business modelling over general Service Oriented Architectures (SOAs).

In presenting StPowla, rather than showcasing all the details, we are concentrating on the benefits of having an approach that combines workflows with policies and services. In particular we will choose an example with a simple workflow that captures the essential process requirements and show how a number of policies can be attached to this to introduce the specialisation required for different business situations.

2 StPowla

StPowla integrates three main ingredients: a graphical workflow notation, a policy language, and the SOA.

The business process core is defined in terms of sequential, parallel and decision-based composition of building blocks called *tasks*, à la BPMN [16]. In this paper we use the *graphical workflow notation* of [9], due to its expressive power and our familiarity with it. However, StPowla is intended to be independent of the workflow language. E.g., UML activity diagrams possibly extended to accommodate useful workflow operators, can be used.

The finer details of the business process are expressed by policies. These can define functional and non-functional requirements of a *task execution*. Furthermore, they can provide *overarching business constraints*, that is a set of rules specified at a global, enterprise or project level and applicable to the whole workflow. The last part that policies might be play is concenrned with suggesting resolutions to particular problems in the process execution (e.g. "do not use a service from X, rather use an equivalent service from Y").

So, the second ingredient is a *policy language*: here we use Appel [19, 21]. Though developed in the context of telecommunications, Appel is a general language for expressing policies in a variety of application domains. It is conceived with a clear separation between the *core* language and its specialization for concrete *domains*, which turns out very useful for our purposes.

The third notable characteristics of StPowla is that it is targeted to SOAs (see, e.g. [2]). Its users, though informatically naïfs, should be aware that the business is ultimately carried out by *services*, i.e. computational entities that are characterized by two series of parameters: the *invocation* parameters (related to their functionalities), and the *Service Level Agreement* (SLA) parameters. Stakeholders can adapt the core workflow by modifying these agreements.

StPowla addresses integration of business processes with Service Oriented Architectures at a high (that is close to the business goals) level of abstraction. It is natural to think that any of the languages used with StPowla , e.g. the polcies or workflows are compiled into XML notations abiding Web service standards. In fact we see this compilation to existing standards were appropriate as key to the interoperaibility at an implementation level.

2.1 Appel

Appel is a policy language designed for end-users: its style is close to natural language, permitting ordinary users to formulate and understand policies readily. Appel formal semantics [14] underpins integration with workflows.

In Appel a *policy* consists of a number of *policy rules*, grouped using a number of operators (**seq**uential, **par**allel, **g**uarded and **u**nguarded choice). A policy rule is a variant of an ECA (event-condition-action) rule, consisting of an optional *trigger*, an optional *condition*, and an *action*. To help the user, a wizard has been presented to formulate policies [21]. The applicability of a rule depends on whether its trigger has occurred and whether its conditions are satisfied. A condition expresses properties of the state and of the trigger parameters. Conditions may be combined with **and**, **or** and **not** with the expected meaning. Actions have an effect on the system in which the policies are applied. Several operators are available to compose actions: **and** leads to the execution of both actions in either order, **andthen** speicifies that the first action precedes the

second in any execution, **or** specifies that either one of the actions should be taken, and orelse that is like **or** but prescribes that the first option is preferred. Triggers and actions are domain specific atoms. Conditions are either domain specific or a more generic (e.g. time) predicates.

2.2 Attributes

The principal means to adapt a workflow to the needs of a stakeholder, is by describing the behaviour of tasks using policies.

The StPowla user can refer to the state of the execution of the workflow, in terms of *attributes*, i.e. properties of individual tasks or of the whole workflow. Attributes can be introduced at different times: a few of them are *predefined*, i.e. they come with StPowla and are applicable to any task or workflow. Most of the attributes are part of the *domain specific* specialization of the Appel component of StPowla, i.e. they come from the ontology of a particular business domain. Finally, each task can have its own attributes.

Predefined attributes include `automation`, which permits to constrain how the task has to be executed, `actorRole`, which is bound to the role interacting with the system to fulfill a task, if any, and `actorId`, which is bound to the identity of the actor playing the requested role, if any. Attributes have types. For instance, `automation` takes values in {`automatic`, `interactive`}, and the first value excludes the involvement of humans in the fulfillment of the task. `actorId` is a `String`, as is `actorRole`, which has `''user''` as default value.

In general, attributes can be overridden by more specific definitions. For instance, the `actorRole` type may be redefined according to the specifics of the business, in a domain dependent definition section. Moreover, in two workflows they may be different, as defined in a workflow dependent definition section.

Some attributes related to a task may be already bound and available on task entry, as task parameters; some other may depend on the results of the invoked service[1]. They are used in policies attached to subsequent tasks.

Similarly, some workflow attributes are available at activation, as workflow parameters, while other are computed along execution. For instance, a workflow relating to a bank, may have an attribute `branchSize` ranging in {`small`, `medium`, `large`}, which is bound when the workflow is instantiated.

An attribute is accessed by a policy, with syntax <prefix>.<attributeName>, where the prefix is either the name of a task, or is left empty, in which case the current task or the current workflow is assumed. In case of ambiguity, the current task may be referenced as `thisTask`, and the current workflow as as `thisWF`.

Finally, here we assume that the standard operators for a type are available as policy actions. For instance, the `totalCost` attribute of a workflow may have an operation `inc` that can be used to accumulate the costs of its tasks.

[1] In this case, the attributes will be eventually refined along the development process to become part of the type of the value that the invoked service returns. However, at the business level of abstraction, these are seen as related to the task, for their use in policies.

2.3 Tasks and services

Tasks are the units where BPM and SOA converge, and where adaptation occurs, by using policies: the intuitive notion of task is revisited to offer the novel combination of services and policies. To specify tasks, we specialize Appel to deal with services, by introducing a special action for service discovery and invocation.

A task has a *name*, which is intended to convey its purpose. For instance, "makeCoffee" is the name of a task where a coffee is expected to be prepared. The details of the working of the task are detailed as the understanding of the workflow grows. Obvioulsy, in well established domains, each task name will identify precise requirements. The task also has an associated policy in Appel that uses the name of the task to express the functional requirements in the service choice, and also specifies non-functional requirements.

A *default* policy is associated with each task. It says that when the control reaches the task, a service is looked for, bound and invoked, to perform the main functionality of the task:

```
appliesTo <taskName>      when taskEntry(<args>)
                            do req(main, <args>, [])
```

For instance, the default policy of a task (with no arguments) where a coffee has to be made is the following one:

```
appliesTo makeCoffee     when taskEntry([])
                            do req(main, [], [])
```

With `taskEntry` we denote the policy trigger, whose arguments are the task parameters, if any. Action `req( _,_,_ )` is the essential bit of the Appel specialization to deal with selection and invocation of services. It is generic, i.e. independent of the business domain. This action takes three arguments:

- the type of the service, expressing its basic functionality. By default it coincides with the name of the task, and is denoted simply as `main`. Anyway, the type must be known in the domain description;
- the list of service parameters, in terms of the task parameters and attributes;
- the specification of the constraints on service selection: they express Service Level Agreements. In the default policy the list of constraints is empty: any service of the required type will do.

Definition 1. (*Semantics of the* `req` *action*) Find a service as described by the first and third arguments, bind it, and invoke it with the values in the second argument[2]. The *action succeeds* if a service is found, and its invocation is successful. It *fails* if either no service is found or if the bound service fails. The binding acts as a commit: only one service is invoked, and if its invocation fails no other found service is invoked.

Adaptation occurs when the user overrides the default policy with his own, by specifying the SLA constraints, or by using the composition operators of Appel.

[2] We assume an automatic search and matching of services to tasks, thus allowing the user to work without the need for detailed service knowledge.

Let us show some examples. In case of task makeCoffee, a constraint on service invocation might deal with `CupTemperature`, which can take values in {`cold`, `warm`}. The request for coffee can be refined, to request that the coffee is served in a warm cup, by introducing the following policy.

```
appliesTo makeCoffee     when taskEntry([])
                            do req(main, [], [CupTemperature = warm])
```

Service discovery must now take into account the SLA constraints, in the third argument of `req`: the invoked service must be able to satisfy the given constraint on CupTemperature.

SLA constraints usually address different kinds of concerns, or SLA *dimensions*. A dimension is specified in the domain description by giving it a name, the set of values, and the applicable operators (if different from the generic ones, like equality and inequality, which we always assume). In the coffee example, one dimension is `CupTemperature`, with values in {`cold`, `warm`}. Essentially, a dimension defines a type.

Since APPEL permits actions to be combined, we can provide the customer with a glass of water (at no cost) before the coffee:

```
appliesTo makeCoffee     when taskEntry([])
                            do req(glassOfWater, [], [Cost = 0])
                               andthen
                               req(main, [], [] )
```

The operator `andthen` lets the two services be invoked in a row.

It is useful to define repairing actions, to cope with failures (e.g. no service found or service invocation failure). Operator `orelse` permits to introduce repairing actions: if it is not possible to have a coffee then a tea is looked for.

```
appliesTo makeCoffee     when taskEntry([])
                            do req(main, [], [] )
                               orelse
                               req(makeTea, [], [])
```

In all the previous examples the policy has no conditional clause, i.e. it is always applicable. In practice, one may want to subordinate the invocation of a service, to some condition, like in

```
appliesTo makeCoffee     when taskEntry([mood])
                            if mood=sleepy
                              do req(main, [], [] )
```

where the task is passed through, without doing anything, if the customer is not sleepy and hence does not need a coffee. So, the actual workflow can depend on the state of the system, inspected by exploiting attributes.

We can now provide the definition of task success and failure.

Definition 2. A task *succeeds* if the associated policy is either not applicable, i.e. its conditions are not satisfied, or if its action succeeds. The task *fails* if the policy is applicable but fails.

In the former example the task fails if we are sleepy, and the `req` action fails: no service is found at our SLA conditions or a service is found but its execution fails. Conversely, the task succeeds if a service is found and correctly executed, or if the policy is not applicable, since we are nicely awake.

In APPEL policies can be combined in groups that control the order in which applicability is checked. Operator `seq` checks its second argument only if the first one is not applicable: caffeine comes in a coke only if the customer is thirsty.

```
appliesTo makeCoffee     when taskEntry([mood]) if mood != thirsty
                              do req(main, [], [] )
                         seq when taskEntry([])
                              do req(glassOfCoke, [], [])
```

In STPOWLA, we exploit this feature to allow the user to assign priorities to the policies attached to the same task. We envisage an extension of the graphical interface to APPEL, known as the APPEL wizard [21], to support the user of STPOWLA in policy management, e.g. with respect to priorities among policies.

2.4 The SLA language

The purpose of the third argument of `req` is to specify a *list* of constraints on service selection. We can constrain a dimension to a single value, or to a range of values. SLA constraints are expressed using parameters, attributes, and *getter* functions that permit to inspect the state of the workflow. Differently from the conditions in the if clause, which are evaluated when executing a policy, these conditions are evaluated by the `req` action itself, against each candidate service. Should the condition be independent from the current state, as in

```
Automation = automatic
```

we could simply consider the type of the third argument to be a String, and let `req` interpret it. However, one may want to express constraints that depend on the state of the computation, e.g. that the automation kind requested is that defined by the current value of the corresponding attribute of the task. We need a mechanism to force partial evaluation of the condition in the third argument of `req` . The situation is similar to what happens with SQL queries in JDBC. We use the convention of prefixing a question mark to those part of a constraint that need evaluation. For instance, a constraint like

```
Automation = ?(thisTask.automation)
```

will entail a search for an `automatic` or `interactive` service, according to the value of `thisTask.automation` when the policy is applied.

2.5 Pragmatics of the customization

Our present attitude is that, when a new policy is introduced, the user should define its relation with other policies applying to the same task This can be done using the APPEL operator `seq`, which introduces an enforcement. That is, we traverse the structure, determining whether the first policy is applicable: if so we apply it, otherwise we check the second one.

Looser attitudes, like composing policies in parallel, may lead to policy conflicts. Conflicts can be detected with a supporting tool [14], and their resolution

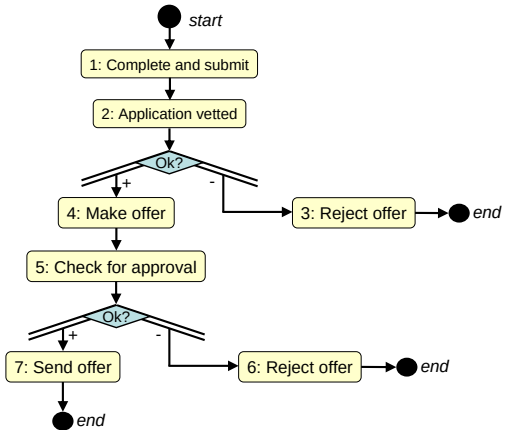

Fig. 1. The workflow for loan approval

can be manual or automatic. Manual resolution can include re-formulation of policies. A technique to automatically solve conflicts is to give a priority to the most recently introduced policies.

3 Case Study

As an example we consider a loan approval process; this realistic case study has been provided by an industrial partner (the German S&N) of the EU–IST–GC2 Sensoria project. A customer uses a web portal to *complete and submit* a loan application. Requests will be forwarded to and handled by the respective local branch, that is the one closest to the customer's location. The data on the form is checked before submission, as to ensure that all essential items such as customer data and evidence for credibility are entered. The submitted application is *vetted* to ensure that the customer is credible and the provided evidence is suitable (the process might terminate here with a rejection of the loan request). If the customer is found to be credible, a loan *offer will be made*. The created offer needs to be *checked and approved* and if found ok will be *sent* to the customer. If the offer is not approved, the application will be rejected and the process ends. Figure 1 shows the StPowla representation of this workflow.

This example is rather simple when considering the structure of the workflow (in that it is mostly sequential with some decision points). However, it shows nicely how the capability to express policies can enhance workflow techniques. The above process is rather generic: it shows stages of submission, checking and offer creation and final approval – these stages are essential in the process as specified by the bank to ensure transparency. However, we can imagine a number of refinements that adapt the process to given situations. It is these that we express as policies, and here are some examples:

P1: Likely, in a big branch the request should be vetted and approved by two distinct members of staff.

P2: In a small branch the branch manager has to approve all applications.
P3: If the customer has a current account, loans up to €5000 must be approved automatically by the computer system.

Of course it would be possible to write a very detailed business process diagram that encapsulates all these options, but it is typical that while the essential process remains the same, the policies change. In particular we have already shown specialisations depending on branch size or loan amount. We can further imagine that there are policies added lately "by need", such as:

P4: If the branch manager of a small branch is out of office, his representative signs all applications.

Overall policies will hence refine the general process to adapt it to specific environments, but they will also allow for adjustments to handle current situations.

To proceed to the actual policy definition in the APPEL subset of STPOWLA, we need to attach each of these informal descriptions to the tasks they affect, and make precise, for each task, type and SLA of the services. The first step is to define task and workflow attributes, and SLA dimensions. We focus our attention on those attributes needed to define the policies:

SLA dimensions	`ActorId = String`
	`Automation = enum{automatic, interactive}`
	`ActorRole = enum{clerk, manager, branchManager, managerRepresentative}`
General Attributes	`actorId: ActorId`
	`automation: Automation`
General Attributesn	`actorId: ActorId`
	`automation: Automation`
Domain Dependent Attributes	`actorRole: ActorRole`
Workflow Attributes	`branchSize: enum{small, large}`
	`branchId: String`
	`loanRequest: Integer`
Task CompleteAndSubmit	`applicantAccountBranch: String`

Now, consider P1: we attach it to task 5, *CheckForApproval*, by turning it into the request that the service is an interactive one and that, if possible, the role interacting with the system to fulfill this task is different from the one that was involved in task 2, *ApplicationVetted*. Here, we have a SLA which is a preference. We use the APPEL operator `orelse`, which gives preference to the first action.

```
P1: appliesTo CheckForApproval
        when taskEntry([]) if thisWF.branchSize = large
           do req(main, [], [Automation = interactive,
                              ActorId /= ?VetProposal.actorId])
           orelse do req(main, [][Automation = interactive])
```

This policy has a natural priority with respect to the default policy of task 5, and the policy wizard will help the user to compose the two policies in a sequence. In the case that the branch size is `small`, then the default policy is applied:

```
P1 seq default: appliesTo CheckForApproval
                  when taskEntry([]) if thisWF.branchSize = large
```

```
    do req(main, [], [Automation = interactive,
                      ActorId /= ?VetProposal.actorId])
    orelse do req(main, [Automation = interactive])
  seq when taskEntry([])
    do req(main, [], [ ])
```

We now express P2 and P3 in Appel.

```
P2: appliesTo CheckForApproval
       when taskEntry([]) if thisWF.branchSize = small
         do req(main, [], [Automation = interactive,
                            ActorRole = branchManager])
P3: appliesTo CheckForApproval
       when taskEntry([])
       if CompleteAndSubmit.applicantAccountBranch = thisWF.branchId  and
         thisWF.loanRequest < 5000
       do req(main, [], [Automation = automatic])
```

The user has to compose these policies with P1. Accordingly to what discussed in Section 2.5, the basic choice is to define a priority and to compose the policies in a sequence. Here, there is a natural priority of P3 over both P1 and P2. P1 and P2 are independent and can be combined in any order (and also in parallel). A possible specification of task 5 is the following one.

```
P3 seq P2 seq P1 seq default
```

Note that sequencing may lead to overriding, and parallel composition may lead to conflicts. These circumstances arise if the composed policies overlap, i.e. if there are some states in which more than one of the composed policies is applicable. For instance, P3 and P1 overlap, as well as P3 and P2, while P1 and P2 don't. To help the user avoiding conflicts and undesired overriding when defining priorities, the policy wizard can be extended to integrate the analysis techniques defined in a previous work [14].

Now assume that P4 is added. We turn it into an Appel policy.

```
P4: appliesTo CheckForApproval
       when taskEntry([]) if thisWF.branchSize = small
         and branchManagerState = outOfOffice
        do req(main, [], [Automation = interactive,
                           ActorRole = managerRepresentative])
```

P4 was added lately, and it is clear that it was introduced to override P2 in the case the branch manager is out of office. Hence it is correct to give it priority over P2. P4 is independent of P1, while it has less priority than P3. Hence, the new specification of task 5 is the following one:

```
P3 seq P4 seq P2 seq P1 seq default
```

This case study has exemplified the StPowla approach, and in particular we have also shown how a number of policies can be used together to refine the workflow.

4 Related Work

Apart from natural English, structured languages are often used for expressing processes. BPEL [12] is considered the de facto standard for SOA-based business processes, despite its initial purpose as a service composition language.

More traditional workflow languages are more appropriate for modelling processes. In particular, YAWL [22] is a powerful workflow language with semantics based on Petri-Nets. These solutions may be considered better in terms of describing processes since they abstract away composition details that would be included in those solutions previously discussed. However, they are unable to define high-level requirements for activities or events that occur in the workflow.

Process calculi and Petri nets offer a formal method in which to express workflows as processes. The formalisms provide operational semantics allowing for reasoning about the process, e.g. [11] and [7].

The most widely-accepted universal process notation for business processes is the Business Process Modelling Notation [16] (BPMN). This graphical notation also describes process flows, though somewhat more structured through the use of swimlanes. One particular advantage of BPMN is that it can be used to model a BPEL process, although it is still limited by its inability to express service selection criteria including non-functional service properties [17].

Policies are descriptive and essentially provide information that is used to adapt the behaviour of a system. Most work deals with declarative policies. Examples are the formalisms to define access control policies, and to detect conflicts [20, 10]; formalisms for modelling the more general notion for usage control [23]; formalisms for SLA, i.e. to specify client requirements and service guarantees, and to *sign* a contract with an agreement between them [5, 4].

RuleML is a language for rule-based and knowledge-based systems, and allows Web-based rule storage, retrieval and interchange [3]. Like APPEL, it is XML-based and allows for the definition of ECA rules (note that for readability we have not used APPEL 's XML syntax in this paper).

WS-Policy [6] seems the obvious candidate when considering policies in the context of Web Services; however WS-Policy addresses mostly aspects related to access control and encryption which are at a much more technical level then the business concepts that we consider in policies. It might be possible to extend WS-Policy with suitable constructs and then complie the polciies into this framework, which is an avenue worthwhile of future investigation.

Ideas of introducing flexibility into workflows have been presented by Reichert and Dadam [18] and in the form of the Woklet system by Adams et al [1]. The former discuss a framework for dynamic process change, but their approach does not include a flexible external system (like our policies) that can affect the workflow in progress. The latter is based on an extensible repertoire of sub processes aligned to each task, one of which is chosen at runtime.

Possibly *AgentWork* [15] system, where ECA rules can be used to drop or add individual tasks to workflows, is closest to our initial discussions on linking policies with workflows [9, 8]. However, there is no notion of tasks being linked to services in this work, and the policies are concerned with task replacement rather than task implementation or service selection.

5 Conclusion and Future Work

StPowla introduces a novel combination of policies and workflows that adds to each of the concepts being used on their own by allowing to capture the essential requirements of a business process using a workflow notation and at the same time allows for the variability to be expressed in a descriptive way by policies. Additionally, StPowla creates a clear link between this enhanced workflow mechanism and services: tasks are being executed by services, and StPowla permits to specify requirements and SLAs that together specify which service can be chosen and what guarantees it has to provide. A specific policy action has been defined that allows for expression of functional and SLA aspects.

StPowla makes a contribution to the engineering of service oriented systems by capturing essential requirements at a business level and allowing for the inherent variability in these requirements to be expressed at a similar level of abstraction.

Despite its bias towards the final user, i.e. business managers, operators and clients, StPowla has a precise semantics, which we presented in natural language. Future work include the definition of a formal semantics, in the form of a mapping to logical theories, extending the work in [14]. This will be the basis for performing some analysis on the process models, e.g. that the various policies are not conflicting.

Acknowledgements

All authors are partially supported by the EU project SENSORIA IST-2005-16004. Dr Reiff-Marganiec has been partially supported by the Royal Society International Outgoing Short Visit – 2006/R2 programme.

References

1. Michael Adams, Arthur H. M. ter Hofstede, David Edmond, and Wil M. P. van der Aalst. Worklets: A service-oriented implementation of dynamic flexibility in workflows. In Robert Meersman and Zahir Tari, editors, *OTM Conferences (1)*, volume 4275 of *Lecture Notes in Computer Science*, pages 291–308. Springer, 2006.
2. G. Alonso, F. Casati, H. Kuno, and V. Machiraju. *Web Services: Concepts, Architecture and Applications*. Springer Verlag, 2004.
3. H. Boley, S. Tabet, and G. Wagner. Design rationale for ruleml: A markup language for semantic web rules. In I.F. Cruz, S. Decker, J. Euzenat, and D.L. McGuinness, editors, *SWWS*, pages 381–401, 2001.
4. M.G. Buscemi, L. Ferrari, C. Moiso, and U. Montanari. Constraint-based policy negotiation and enforcement for telco services. 2007.
5. M.G. Buscemi and U. Montanari. Cc-pi: A constraint-based language for specifying service level agreements. pages 18–32, 2007.
6. J. Schlimmer (ed). Web services policy 1.2 – framework (WS-Policy). W3C, Apr 2006. http://www.w3.org/Submission/WS-Policy/.
7. X. Fu, T. Bultan, and J. Su. Formal verification of e-services and workflows. In C. Bussler, R. Hull, S. A. McIlraith, M. E. Orlowska, B. Pernici, and J. Yang, editors, *WES*, volume 2512 of *LNCS*, pages 188–202, 2002.

8. S. Gorton and S. Reiff-Marganiec. Policy support for business-oriented web service management. In *Proceedings of the Fourth Latin American Web Congress (LA-WEB'06)*, pages 199–202, Washington, DC, USA, 2006. IEEE Computer Society.
9. S. Gorton and S. Reiff-Marganiec. Towards a task-oriented, policy-driven business requirements specification for web services. In S. Dustdar, J.L. Fiadeiro, and A.P. Sheth, editors, *Business Process Management*, volume 4102 of *Lecture Notes in Computer Science*, pages 465–470. Springer, 2006.
10. J. Y. Halpern and V. Weissman. Using first-order logic to reason about policies. In *16th IEEE Computer Security Foundations Workshop (CSFW'03)*, page 187, Los Alamitos, CA, USA, 2003. IEEE Computer Society.
11. R. Hamadi and B. Benatallah. A petri net-based model for web service composition. In K.-D. Schewe and X. Zhou, editors, *ADC*, volume 17 of *CRPIT*, pages 191–200. Australian Computer Society, 2003.
12. D. Jordan and J. Evdemon et al. Web services business process execution language version 2.0. W3C, Aug 2006. http://docs.oasis-open.org/wsbpel/2.0/wsbpel-specification-draft.pdf.
13. F. Kamoun. A roadmap towards the convergence of business process management and service oriented architecture. *Ubiquity*, 8(14), 2007. ACM Press.
14. C. Montangero, S. Reiff-Marganiec, and L. Semini. Logic-based detection of conflicts in APPEL policies. To appear in IPM–FSEN, LNCS, 2007.
15. R. Müller, U. Greiner, and E. Rahm. Agent work: a workflow system supporting rule-based workflow adaptation. *Data Knowl. Eng.*, 51(2):223–256, 2004.
16. OMG. *Business Process Modeling Notation (BPMN) Specification*, Feb 2006.
17. J. O'Sullivan, D. Edmond, and A. H. M. ter Hofstede. Formal description of non-functional service properties. Technical Report FIT-TR-2005-01, Queensland University of Technology, Brisbane, Feb 2005.
18. M. Reichert and Peter Dadam. $\text{Adept}_{\text{flex}}$-supporting dynamic changes of workflows without losing control. *J. Intell. Inf. Syst.*, 10(2):93–129, 1998.
19. S. Reiff-Marganiec, K.J. Turner, and L. Blair. Appel: The accent project policy environment/language. Technical Report TR-161, University of Stirling, Dec 2005.
20. F. Siewe, A. Cau, and H. Zedan. A compositional framework for access control policies enforcement. In *Proceedings of the 2003 ACM workshop on Formal Methods in Security Engineering*, pages 32–42, NY, NY, USA, 2003. ACM Press.
21. K. J. Turner, S. Reiff-Marganiec, L. Blair, J. Pang, T. Gray, P. Perry, and J. Ireland. Policy support for call control. *Computer Standards and Interfaces*, 28(6):635–649, 2006.
22. W. M. P. van der Aalst and A. H. M. ter Hofstede. YAWL: yet another workflow language. *Inf. Syst.*, 30(4):245–275, 2005.
23. X. Zhang, F. Parisi-Presicce, R. Sandhu, and J. Park. Formal model and policy specification of usage control. *ACM Trans. Inf. Syst. Secur.*, 8(4):351–387, 2005.