Model-Driven Development of Adaptable Service-Oriented Business Processes*

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

[1] Dipartimento di Informatica, Università di Pisa, {monta,semini}@di.unipi.it
[2] Department of Computer Science, University of Leicester, srm13@le.ac.uk

Abstract. Businesses typically structure their activities with workflows, which are often implemented in a rather static fashion in their IT systems. Nowadays, system requirements change rapidly as businesses try to maintain their competitive edge, calling for similar agility of the IT systems. To this end, we present StPowla, an approach that marries service oriented architecture, policies and workflows to support the agile execution of business workflows. In StPowla, the business is modelled by workflows, whose tasks are eventually carried out by services. Adapatation is obtained by allowing the stakeholders to define policies that establish the quality levels required of the services. The prototype StPowla support architecture comprizes the transformation of the workflow model into executable WS–BPEL to be deployed in the ODE–BPEL execution engine, the generation of default policies from the model, and the enactment of the policies by the Appel policy server. The SENSORIA Finance Case Study is used throughout the paper.

1 Introduction

It is common practice, to reduce time-to-market, that enterprises federate their operations by networking via Web services, and these federations can change to follow evolving business goals. On a smaller scale, processes may need to adapt to temporary shortage of resources by simplifying, or even skipping, some steps. These environmental changes need to be supported while the software system is operating. The integration of Business Process Management (BPM) and Service Oriented Architecture (SOA) has been recognized as a promising approach in this respect [17].

However, the integration of BPM and SOA still requires large efforts by highly skilled personnel. Currently, the business rules introduced by business roles like sales or technical managers need to be mediated by business analysts who, thanks to their knowledge of the business processes, transforms them into directives to the programmers for updating the workflows, e.g. in WS-BPEL.

* This work has been partially sponsored by the project SENSORIA, IST-2005-016004. The authors would also like to thank Hong Qing (Harry) Yu for his contributions towards the implementation of the approach and his input to a draft of section 4.

Charfi and Mezini [10] discussed the integration of rule–based languages and process–based service composition, considering either "to adapt one of the languages to be more compatible with the other by extending e.g., the rule-based language with process-oriented features, or the other way around", or "to enhance one of the languages with an interface to the other language, so that the features of the latter can be used in programs written in the former". They concluded that both approaches suffer from the lack of seamless integration due to the paradigm mismatch which the programmer is confronted with, and privilege an Aspect Oriented approach [11].

In the Service-Targeted Policy-Oriented WorkfLow Approach (StPowla – to be read like "Saint Paula") [12], we have integrated the two paradigms seamlessly, via the SOA: the workflow composes coarse–grain business tasks, and the policies control the fine–grain variations in the service level of each task. The integration occurs at the conceptual level and in the supporting environment, rather than at the linguistic level.

Being policy based, the approach naturally distinguishes between a *core* description of the process and its *variations*, which can be specified by declarative rules, and can be dynamically deployed or removed. This fosters Business Process flexibility, by raising the abstraction level at which the variations are specified, while at the same time providing an efficient implementation technique.

In the approach, business tasks are 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* (SL) parameters, related to the resources they exploit to carry out their job: Stakeholders can adapt the core workflows by requiring higher or lower quality of service (QoS), therefore consuming more or less resources.

The kind and granularity of the 'resources' that are identified in the business domain are often more abstract than bandwidth and power, i.e. those usually addressed in service level agreements. For instance, a task of a given type may need higher levels of authorization in given circumstances, and lower levels in others. In StPowla, the authorizing business roles are seen as resources, ordered along an *AuthorizationLevel* dimension: the identification of these *dimensions* is a key design activity in the approach.

The combination of workflows, SOA, and policies can be exploited at its best, if a coherent design strategy is adopted to foster flexibility. In a nutshell, such a strategy is to find the best balance between (i) keeping the workflows simple, i.e. without explicit choices that depend on the quantity/quality of resources available to the tasks, and (ii) providing large and foreseeing ranges of choices to the policies, to support modelling the business rules as they emerge.

In this paper we present the embodiments of the StPowla concepts in UML4SOA [18], the UML profile that introduces stereotypes for the relevant concepts (workflow, tasks, service level, etc.) in the standard framework of UML (classes, interfaces, activities, etc). The main contribution of the paper is the design of an environment to model, deploy, and run StPowla business processes. Note that, besides supporting the use of services with different service levels in the business process, the environment itself is based on a service oriented architecture, orchestrating a workflow engine, a policy server and a service broker.

2 The Modelling Concepts

StPowla is a workflow based approach to business process modelling that integrates:

– a standard graphical notation, to ease the presentation of the core business process;
– policies, to provide the desired adaptation to the varied expectations of all the business stakeholders;
– the SOA, to coordinate the available services in a business process.

More specifically, workflows are used in StPowla to define the business process core as the composition of building blocks called *tasks*, à la BPMN. Each task performs a meaningful step in the business, whose purpose is well understood at an abstract level by the stakeholders. That is, a task is understood as to its effects in the business, regardless of the many details that need to be fixed in its actual enactment.

Policies are used to express finer details of the business process, by defining Service Level (SL) requirements of task *executions*. The added value is that policies can be updated dynamically, to adapt the core workflow to the changing needs of the stakeholders.

Tasks are the StPowla units where BPM, SOA and policies converge, and adaptation occurs: the intuitive notion of task is revisited to offer the novel combination of services and policies.

When the control reaches the task, a service is looked for, bound and invoked, to perform the main functionality of the task. Functional requirements of the task are described in the task specification. Conversely, service invocation is always local to task execution, i.e., a service is invoked to satisfy the requirements of a task, not to satisfy some overarching business requirement.

A task can be associated to a policy. Indeed, the principal means to adapt a workflow to the needs of a stakeholder, is by intervening on the behaviour of the tasks using policies. To define a policy StPowla users can refer to the state of the execution of the workflow, as described by task and workflow specification.

In the following, terms in "guillemets" are the UML4SOA stereotypes for the StPowla concepts: A ≪workflow≫ is an activity action that calls the specified behavior, i.e., a lower level workflow; A ≪Task≫ is an activity action that calls the specified main operation.

Next, we present the StPowla concepts with the support of a *loan negotiation* process, part of the *Finance Portal* case study (Chapter 0-3).

2.1 Model Specification

In StPowla a ≪Task≫ is characterized by a ≪Taskspecification≫ via a *name*, a *description*, an *interface*, and a set of service level *dimensions*. The name and description convey the purpose of the ≪Task≫: in well established domains, they identify precise, even if informal, functional requirements for the task. The interface provides the formal signature of the operation carried out by the task. As already mentioned, a task is actually carried out by a service: the interface includes an operation called main, with the same parameters and return type of the required service.

The ≪Taskspecification≫ can specify a number of *service level dimensions* (≪NFDimension≫) that specify the non-functional dimensions that characterize the service to invoke. Besides specifying the type of each dimension, the designer can define:

- the ranges within which the service level can vary. In the case study the non functional dimensions are specified as enumerations, and the ranges are the enumeration literals: manual and automatic; supervisor and branchManager (see Fig. 1).
- a default value. For instance, manual and supervisor in Fig. 1.

Then, the stakeholders can specify the service levels they require along each dimension, by installing *policies* for a given task, overriding the default value, as discussed below.

Finally, a ≪Taskspecification≫ can have *attributes*: they define properties of a ≪Task≫ that depend on the state of the workflow, and can be used in the policies to access the execution state and select the most appropriate service levels when the ≪Task≫ is activated. Attributes are specified at design–time and bound at run–time, e.g. on task/workflow entry, as a function of the inputs, and of the other attributes.

To sum up, from a behavioural perspective, when the control reaches the task, operation main is executed. The execution of main triggers the search and invocation of a suitable service, and returns the computed result. The search identifies a service implementation that satisfies the current policies, i.e., the policies to be applied in the current state of the workflow, or the default values for the service level, when not overridden.

Just like tasks, ≪workflow≫s have a ≪WfSpecification≫ defining their attributes and signature. Moreover, differently from tasks, their behaviour is defined explicitly, via an associated UML activity, whose nodes are either tasks or workflows.

2.2 Case Study: Loan Approval

In this scenario, a customer uses a web portal to request a loan from a bank. The request is forwarded to and handled by the *local branch*, i.e. the closest one to the customer's residence. At the local branch, to process the loan request, and

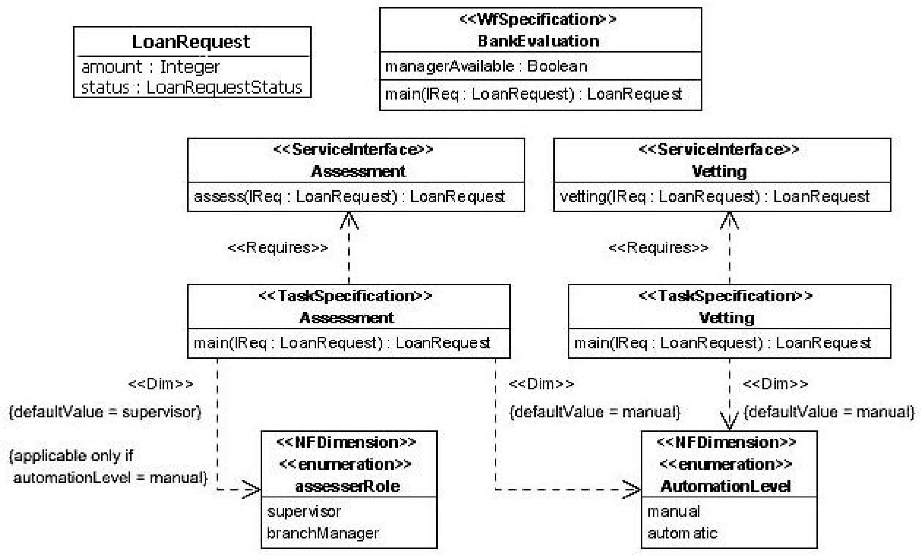

Fig. 1. The specification of BankEvaluation.

before a contract proposal is sent to the customer, there are two necessary steps: a preliminary evaluation (*vetting*), to ensure that the customer is credible, and a subsequent step (*assessment*), where the contract proposal can be approved or rejected.

We concentrate on an inner workflow of the *LoanApproval* business process, *Bank Evaluation*. The diagrams in Figures 1 and 2 specify this workflow. As indicated by the main operation in the ≪WfSpecification≫, the *Bank Evaluation* workflow processes a *LoanRequest*, that is, the document collecting all the information on the loan being worked on . The actual process is in Figure 2, and shows the steps to accept or reject the request. The attribute managerAvailable reflects part of the state of the bank's branch enacting the workflow, and can be used to state the business policies.

Let us now have a look at ≪Taskspecification≫ *Assessment*, which characterizes the second step of this workflow. Its ≪ServiceInterface≫ identified by the ≪requires≫ association specifies that this task needs a service able to transform a LoanRequest[3]. This ≪ServiceInterface≫ is implemented by a service invoked by the task and can be adapted along two dimensions: AutomationLevel and AssesserRole. The former is a standard dimension that roughly distinguishes two kinds of implementations: those that exploit only machine resources, automatic, and those that need human resources, manual. The second dimension may vary from

[3] The description of the transformation is not shown in the diagram, but should appear in the report containing it, or in a suitable pane in the supporting environment (for instance, in the *property* pane in the IBM Rational Software Architect –RSA– where the figure comes from).

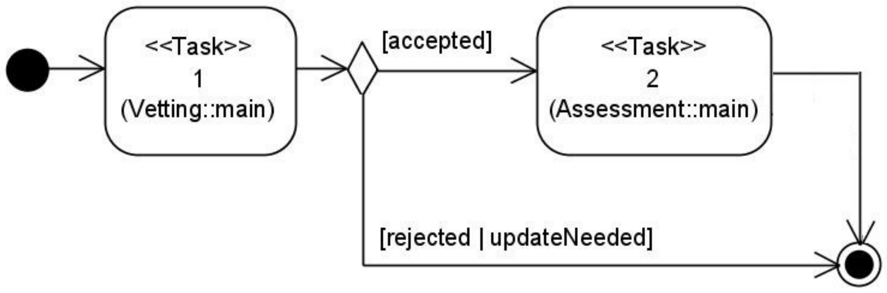

Fig. 2. The BankEvaluation activity.

one ≪Taskspecification≫ to another, since it classifies the different roles that, in different situations, can be involved in the ≪Task≫. Here, we have two such roles, branchManager and supervisor, defined as the default.

To deal with service levels, another stereotype has been introduced, ≪Dim≫, with a tagged value default to specify the default level. In the figure it is shown how the default values can be set in the model. Two dimensions of the same ≪Taskspecification≫ need not be independent: for instance, in our example, AsesserRole makes sense only if AutomationLevel is set to manual.

Besides its ≪Taskspecification≫, a ≪Task≫ also has a *name*, which is only used to distinguish different occurrences of the same task type in the same workflow. Therefore, we simply use integers as names for ≪Task≫s. The BankEvaluation workflow simply states that the request is first subject to Vetting and then, if accepted, to Assessment. In either step, the request may be rejected; after Vetting, more information may be requested from the applicant. The default service levels imply that a supervisor will perform Assessment. Similarly, a clerk will vet the request by default – not shown here. Variations can be specified by policies, as shown next.

2.3 Policies

A task may have associated policies, which come in two flavours: those that adapt the workflow by constraining the task behaviour along its SL dimensions, and those that modify the workflow structure, adding and/or deleting tasks. The latters are discussed in [7]; here we concentrate on the formers, and call them simply policies. For instance, the generic *BankEvaluation* process can be adapted to specific situations via policies, like:

P1: In case of loans of small amount, both vetting and assessment are performed automatically.

P2: In a small branch, the branch manager has to approve all applications.

P3: If the branch manager of a small branch is out of office, loan applications are approved by the manager's representative.

In StPowla, the policies act on the process by specifying the requested service levels as a function of the state of execution as expressed in the attributes. To this purpose, we use is Appel [36]. 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*, a separation which turns out very useful for our purposes.

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 has the following syntax

$$[\textbf{when}\ \textit{trigger}]\ [\textbf{if}\ \textit{condition}]\ \textbf{do}\ \textit{action} \qquad (1)$$

The core language defines the structure but not the details of these parts, which are defined in specific application domains. Base triggers and actions are domain-specific atoms. An atomic condition is either a domain-specific or a more generic (e.g. time) predicate. This allows the core language to be used for different purposes.

The applicability of a rule depends on whether its trigger has occurred and whether its conditions are satisfied. Triggers are caused by external events. Triggers may be combined using **or**, with the obvious meaning that either is sufficient to apply the rule. Conditions may be negated as well as combined with **and** and **or** with the expected meaning. A condition expresses properties of the state and of the trigger parameters. Finally, actions have an effect on the system in which the policies are applied. A few operators (**and**, **andthen**, **or** and **orelse**) have been defined to create composite actions.

In StPowla, to specify tasks, we specialize Appel. In this paper we only consider the specializations relevant to refinement policies, additional extensions exists for reconfiguration policies and they are introduced in [7]. The only possible trigger of a policy is the activation of the associated task (reconfiguration policies allow for a number of other triggers). To deal with services, we introduce a special action, `req(-, -, -)`, for service discovery and invocation. The semantics of this action is to *find* a service as described by the first and third arguments (specifying service type and SLA constraints), *bind* it, and *invoke* it with the values in the second argument (the invocation parameters).

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

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

where `taskEntry` denotes the policy trigger, whose arguments are the task parameters, if any. Adaptation occurs by overriding the default policy. For instance, to satisfy the requirements expressed by policy P2, we associate the following policy to task Assessment:

```
P2: when taskEntry([]) if thisWF.branchSize = small
```

```
do req(main, [], [AutomationLevel = manual,
                  AssessorRole = branchManager])
```

To ease the policy designer task, policies can also be defined by tables, whose structure is derived from the UML4SOA model of the workflow. A default table is automatically derived, which corresponds to the default policy: no discriminator appears, and the default value is assigned to each SL, as in Table 1. Then, the designer can redefine the default policy, by adding discriminators and SL values. For each new discriminator, the table is automatically extended, by building the decision tree, and by assigning the default value to the SLs. Finally, the designer can override any SL with the intended value. An example, relative to ≪Task≫ 2

Policies for BankEvaluation.2: Assessment	
Requested SLs	
Automation level	AssesserRole
default: manual	default: supervisor

Table 1. The policy table for task 2 – automatically derived from the workflow model.

Policies for BankEvaluation.2: Assessment			
Discriminators		**Requested SLs**	
iReqAmount<5000 ⋆	*branchSize=small* ⋆	Automation level	AssesserRole
true	true	P1: automatic ⋆	N/A
true	false	P1: automatic ⋆	N/A
false	true	default: manual	P2: branchManager ⋆
false	false	default: manual	default:supervisor

Table 2. The policy table for task 2 – interactively extended by the designer.

of the *BankEvaluation* workflow, is given in Table 2, which reflects the informal policies P1 and P2 of Section 2. In a policy, task and workflow attributes are accessed by name, while the usual OO dot notation allows accessing the attributes of the task data, like in *lReq.amount*. The left side columns encode a decision tree, for the two discriminators $lReq.amount < 5000$ and $branchSize = small$: each row on the right side lists the required service level for each dimension (one per column on the right). For instance, if neither condition holds, the default values are requested for the service levels. The policy names are there for traceability, and the stars denote the only parts of the table that are input by the stakeholders.

3 Design and Deployment

We distinguish two roles in the design of a system integrating BPM and SOA: the *BP Designer* dealing with workflow and policy specification, and the *Service Producer*, who is in charge of designing, implementing, and registering the services. We can also distinguish between *Workflow* and *Policy Designer*, since they deal with different aspects of the business process. However, we note that they normally work in the same organization, they both specify the requirements from a business point of view, they share the modelling of the task types like the one in Figure 1, and often they are the same person, namely the Business Analyst.

In this section we describe the process to apply the StPowla approach, and the tools we propose to support the designers job.

3.1 Workflow Design

The Workflow Designer defines the task types and orchestrates different tasks into an executable process to achieve a business goal which is requested by the end-users. To do that, he uses the UML4SOA profile as notation and the IBM Rational Software Architect (RSA) as editor. Once the workflow model is created (or updated), it is transformed into executable WS-BPEL [29] and deployed in the ODE BPEL execution engine [25]. Besides, policy tables templates, with the adaptable service levels and the default values are automatically derived from the workflow model, as discussed in the last part of the previous sections. The policy definition is also supported by RSA, which has been extended via the *PolicyDesign* plug–in, This way the designer is naturally offered the context for policy definition, that is task types definitions, including attributes and service level dimensions. Once specified and deployed, the policies affect all subsequent workflow enactments.

3.2 Service Design

For the moment being, StPowla makes a sort of "closed world" assumption: whenever a new task or dimension is introduced, new refinement services need be designed, implemented and deployed. The discussion that follows describes a method to specify these services.

Any service refining a ≪Taskspecification≫ implements the same ≪ServiceInterface≫ interface, but offers a specific kind of QoS, defined in an associated *capability document* (capDoc). For instance, an "automatic" implementation, and one that involves the BranchManager can be specified as shown in Figure 3 for *Assessment*. The ≪ServiceInterface≫ interface of the ≪Taskspecification≫ is refined respectively by the interfaces *AutomaticAssessment* and *BranchManagerManalAssessment*. The capDoc tag of ≪TaskRefinement≫ specifies the capDoc describing which service levels the implementation must offer, that is it constrains the possible implementations. For instance, Table 3 shows the two documents referred to in Figure 3. So, the implementer has all the information he needs:

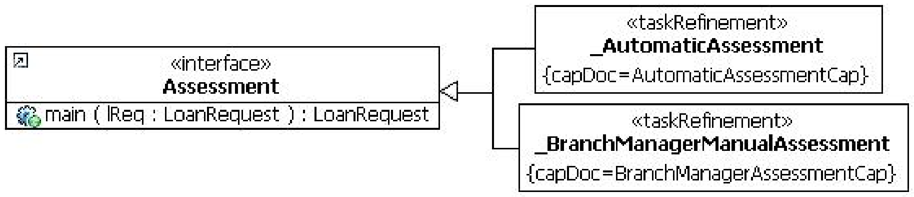
«interface»
Assessment
main (lReq : LoanRequest) : LoanRequest
«taskRefinement»
_AutomaticAssessment
{capDoc=AutomaticAssessmentCap}
«taskRefinement»
_BranchManagerManualAssessment
{capDoc=BranchManagerAssessmentCap}

Fig. 3. Service specifications to refine ≪Task≫ 2: Assessment.

```
<capDoc name="AutomaticAssessmentCap" serviceType="Assessment"> <and>
  <qos name="AutomationLevel" enum="automatic" confidence="1"/>
</and> </capDoc>
<capDoc name="BranchManagerAssessmentCap" serviceType="Assessment"> <and>
  <qos name="AutomationLevel" enum="manual" confidence="1"/>
  <qos name="AssesserRole" enum="BranchManager" confidence="1"/>
</and> </capDoc>
```

Table 3. Service capabilities.

functionality from domain knowledge and enterprise standards, service ≪ServiceInterface≫, and capabilities from the capDoc.

Note that the scenario we assume in StPowla entails a strict co-operation between task specifier, policy specifier and service implementer: this is possible since they all share the same UML4SOA model of the business.

3.3 Deployment

The workflow and policy deployment targets three components of the run–time support, namely the three rightmost ones in Figure 4. Steps 1 to 3 occur when a new UML model is deployed: The BPEL representation of the workflow is generated by the central deployment service, the StPowlaDeployEngine, and downloaded to the workflow engine: We currently use Apache ODE (Orchestration Director Engine) [25] to execute WS-BPEL [29] representations of the

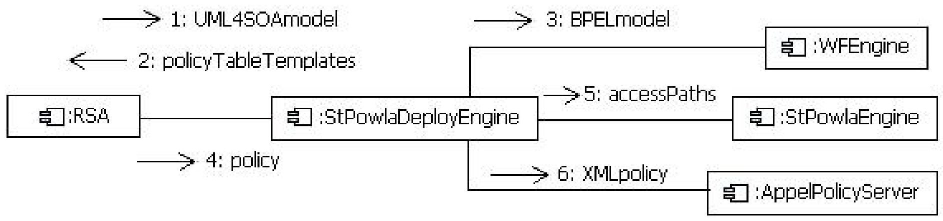

Fig. 4. Workflow and policy deployment.

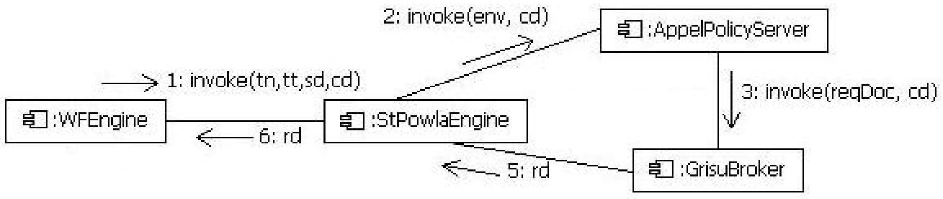

Fig. 5. Runtime StPowla choreography.

workflows. Also, the StPowlaDeployEngine generates the policy tables templates and stores them back into the RSA. Thereafter, whenever a new table instance is deployed, the XML representation of the policy is generated and loaded into the Appel Policy Engine [36,32]. The last component affected by deployment is the StPowlaEngine, that is, the core of the run–time environment. For each policy, it is loaded with the paths it will use to access the run–time values needed to evaluate the policy itself (more details in Section 4).

4 Run–Time Environment

In this section we describe the tools we propose to implement the StPowla approach. There are four cooperating services and six steps in the run-time environment to complete a task (see Figure 5). The WFEngine interacts with the StPowlaEngine, which coordinates the AppelPolicyServer and the GrisuBroker to select a refinement per each task in the workflow, according to the current state and policies, and invoke it. The AppelPolicyServer selects the requirements for service discovery, and the GrisuBroker performs both discovery and invocation.

The **WFEngine** is the interpreter of the business process. All the tasks have the same BPEL behaviour: they invoke the StPowlaEngine, to detect and invoke the task refinement that best suits the current requirements. The StPowlaEngine receives the *task name (tn)*, *task type (tt)*, *state data (sd)*, and *call data (cd)*. The latter is the data for the invocation of the chosen service, while *sd* carries the relevant information on the state of the workflow enactment, i.e., the current values of the task and workflow attributes. In our example, the second task of Figure 2, the WFEngine will pass as arguments “2” , “Assessment”, the current values of the workflow and task attributes (branchSize and managerAvailable), and the loan request, that is, the input argument to the task.

Then, the **StPowlaEngine** builds and sends the environment for policy evaluation to the AppelPolicyEngine. The environment has bindings for the task name and type, and for all the information in the call data and attributes that are used in the policies currently deployed for the task. In the example, the domain of the environment will be {taskName, taskSpecification} united with {branchSize, lReq.amount} or {managerAvailable} according to the deployed policy (P1 or P2, respectively). Remember that, to allow the StPowlaEngine to build such an environment at run–time, whenever a new policy is deployed, the

```
<reqDoc> <serviceType>Assessment</serviceType> <and>
   <qos name="AutomationLevel" enum="automatic" confidence="1"/>
</and> </reqDoc>
<reqDoc> <serviceType>Assessment</serviceType> <and>
    <qos name="AutomationLevel" enum="manual" confidence="1"/>
    <qos name="AssesserRole" enum="branchManager" confidence="1"/>
</and> </reqDoc>
```

Table 4. The reqDocs for policies P1 and P2.

relevant information is stored in the StPowlaEngine itself (step 5 in Figure 4)). To understand how this is done, we need to point out that i) call data are represented in XML, according to schemas that are derived from the UML4SOA model, and ii) state data are also represented in XML in a standard format, shared among the WFEngine, the StPowlaEngine, and the StPowlaDeployEnv. So, the StPowlaEngine knows which data to retrieve from the state and call data, and pairs them to the paths while building the environment.

The **AppelPolicyServer** determines the requirements of the task refinement service to call, in the current state of workflow enactment, and according to the policies currently installed in the policy engine. We are using the Appel policy engine [36,32] to reap the benefits of its architecture. Indeed Appel neatly distinguishes between the core mechanisms for policy evaluation and the extensions mechanisms that allow tailoring the engine to particular domains. Tailoring is done by defining the relevant triggers, predicates and actions: for StPowla, we defined i) the trigger related to task entry, which reacts to the invocation from the StPowlaEngine, and ii) the action that builds and returns a specific requirements document (reqDoc) for the service broker. For instance, when policy P2 above is triggered, the policy server returns the upper reqDoc in Table 4. Similarly, the policy server generates the lower document for P1.

The **GrisuBroker** uses the input reqDoc to discover a matching service, i.e. one of the correct ≪Taskspecification≫ and offering service levels that match the request. The discovered endpoint is used to invoke the main operation, with the call data as argument. The data returned by the service is then passed back to the workflow engine, which carries on with the workflow. Note that the lack of a matching service denotes a flaw in the design/deploy process. Indeed, due to StPowla's "closed world" assumption, whenever a new service is deployed the needed data access paths are stored in the StPowlaEngine.

For the two policies in our example, Grisu would discover respectively the capDocs shown in Table 3. Grisu can discover these services by comparing reqDocs (like the one in Table 4) and capDocs: no other information is needed. The format of reqDoc and capDoc is taken from the service broker DINO[26]: For this reason, our service broker is called Grisu[4].

[4] Grisu is a popular Italian cartoon character, a small dragon: since DINO is a dinosaur, we see Grisu as DINO's child.

5 Barbed Model Driven Development

When thinking of model driven development, the immediate understanding is that models drive software development, in the sense that the software is constructed by transforming models from higher levels of abstraction to the point where we reach a model which is executable with the desired degree of quality characteristics. What tends to be less evident, is that, precisely in order to reach the desired quality, many other models are used in the verification and assessment of the solutions under consideration at the various stages of development. That is, looking at the development process, besides a spine of model transformations moving from highly abstract, domain related models down to concrete platform related models (programs), we can see a number of barbs, relating models in the spine to specialized models that permit specific, often very sophisticated, analysis of parts of the software under development, usually in the early stages. We called this approach Barbed Model–Driven Software Development [24].

Within the work on StPowla, we applied this idea to address the detection of possible conflicts among policies. Indeed, when several policies are composed (or applied simultaneously) they might contradict each other: a phenomenon referred to as *policy conflict*. Policy conflict has been recognised as a problem [31] and there have been some attempts to address this, mostly in the domain of access or resource control. In the case of end-user policies the problem is significantly increased by a number of factors. To name a few:

- the application domains are much more open and hence more difficult to be modelled,
- there will be many more end-user policies than there are system policies (sheer number of policies),
- end-users are not necessarily aware of the wider consequences of a policy that they formulate.

To provide the user with confidence that the rules are conflict free, we propose to filter his/her input to detect those policies that, if entered in the policy engine, would originate conflicts. The advantages include that we can anticipate conflict detection—traditionally performed at run-time—at design-time. Indeed, the well-known advantages of early verification apply to policies as well. In [20,21], we take a logic–based approach to this end: conflicts are detected by deducing specific formulae in a suitable theory. A translation function has been defined to derive the logical representation of Appel policies in the temporal logic ΔDSTL(x) [22,23]: as a side effect, this function defines a formal semantics for Appel, which before was only defined informally, like most of the policy languages. The translation maps a group of policies into a logical theory expressing its meaning. The temporal features of ΔDSTL(x) permit the expression of the dynamics of the rules, the event operator facilitates dealing with the triggers, and the spatial features permit addressing the localization of the policies.

More specifically, the filter maintains a logical theory representing

1. the relevant information on the domain, that is, interesting facts and inference rules valid in the application domain,
2. the set of policies currently *installed*, i.e. contained in the Policy Base,
3. a representation of the state space of the system, restricted to the part accessed when selecting the policies, and
4. the definition of what constitutes a conflict.

Then , it is sufficient to equip the filter also with a deduction engine for the logic in use: before a new policy is added to the Policy Base, its logical representation is added to the filter theory, and then the deduction engine is run: if one of the formulae identifying a conflict is derived, the user is informed and he can resolve the detected conflict.

Taking a similar direction, we designed a barb towards UML state machines to model check whether policies are free of conflicts [5]. To this aim, we have defined a semantics-preserving compositional mapping from APPEL to UML, suitable for model checking with UMC [19,37]. Since UMC operates on UML state machines, the target of the mapping happens to be a subset of UML state machines: policies and policy groups are defined using composite states, i.e. states with structure reflecting the one imposed by the APPEL operators onto policies and actions.

A policy in APPEL is built with triggers, conditions, and actions, just like state machine transitions. Indeed, triggers, conditions and basic map onto the UML triggers, conditions and actions that decorate the machine state transitions, in the natural way. This is fortunate, since they are domain dependent, and we can exploit the flexibility that UML provides w.r.t the language in which to express them, to best fit the domain peculiarities. Some more work is needed to map combination of actions since action combinators are defined in terms of the outcome of the actions under composition. However, this is true in a very broad sense that need not consider the details of the action semantics, but only an abstract notion of *success* and *failure*. Intuitively, these notions entail that an action may complete normally (success) or may abort for some reason (failure). Again, APPEL leaves the specifics of when an action succeeds or fails to the domain, and simply defines the success or failure of a composed action as a combination of the successes and failures of the actions under composition.

UMC is an on-the-fly model checker built to analyze UML state machines for properties expressed in the action- and state-based branching-time temporal logic UCTL [6]. In the case of policies, conflicts arise if a pair of conflicting actions is executed. To prove conflict freeness the full state space must be checked to exclude a path along which both actions are executed (in any order). The approach has been validated with the SENSORIA finance case study.

6 Related Work

Much work has been published in the area of business process specifications, ranging from natural English to structured languages used for expressing processes. BPEL [16] is considered the de–facto standard for SOA-based business processes, despite its initial purpose as a service composition language.

Policies are descriptive and essentially provide information that is used to adapt the behaviour of a system. Most work deals with declarative policies. Notable examples are the formalisms to define access control policies, and to detect conflicts [33,15]; formalisms for modelling the more general notion for usage control [38]; formalisms for SLA, i.e. to specify client requirements and service guarantees, and to *sign* an agreement between them [9,8].

Ideas of introducing flexibility into workflows have been presented by Reichert and Dadam [30] and in the Woklet system by Adams et al [3]. The formers discuss a framework for dynamic process change, but do not include support for changes to 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. The difference here is that our adaptation focuses on changing the Service Levels, thus providing guidance in the design phase.

In *AgentWork* [27], rules can be used to drop or add individual tasks to workflows. This is close to our reconfiguration policies [13,14]. 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.

A policy-driven approach is proposed in [34], to extend BPEL definitions with transactional behaviour, as the one offered by WS-Coordination. To actually enforce the coordination behaviour for the BPEL processes, as specified by the policies, a separate middleware system has been integrated in the architecture.

Among the various types of software tools available in the marketplace for BPM support, several business rules management tools (BR tools) became available in recent years. Among the most complete and promising solutions are Blaze Advisor [1] and JRules [2]. Recently BR tools have been including SOA integration features, such as deploying rule services as part of an SOA [28].

It is worthwhile to locate StPowla in the grid provided by two popular classification of the BR tools [4,35]. Being aimed at business analysts, StPowla falls in the knowledge–based BR tools, and can benefit the people/document intensive processes, which it can support with respect to workflow agility and resource management via its reconfiguration/refinement mechanisms. Historically, the knowledge-based BR tools have been targeted to decision intensive business processes. They foster 'rule–driven programming', with no clear difference between the rules driving the high level behavior of the workflow and those governing the application low level, such as computation and inference rules. In this respect, StPowla improves the overall structure of process representation with its distinction between core process and variations along the SL dimensions.

7 Conclusions and Future Work

StPowla introduces a novel combination of policies and workflows that allows the designer to capture the essence of a business process as workflow and to express variations in a descriptive way.

In this paper we have only considered *static* QoS requests, which involve no run-time assessment of the resources. Consider now P4: "In a big branch, the request should be vetted and approved by different members of staff". Without introducing cross-task requirements, the reqDoc for *Assessment* cannot be completed at design time, that is, it must be parametric and instantiated at run–time as a function of the identity of the vetter. On the capDocs side, one way is to introduce as many different task refinements as assessors, specify each one statically, and let Grisu make the choice. Alternatively, one should change the ≪ServiceInterface≫ of the ≪TaskRefinement≫, adding as a parameter the needed info (the assessor, for P4). In terms of service level, this amounts to characterize the refinement as being able to use any specific resource of the requested type.

We already mentioned that in StPowla we assume a strict co-operation between task specifiers, policy specifiers and service implementers, which share the same UML4SOA model of the business process. Looking for task refinements made available by independent providers, involving e.g. interface adaptation, is left for future work.

References

1. http://www.fico.com/en/Products/DMTools/Pages/Fair-Isaac-Blaze-Advisor-System.aspx. Last visited: March 2009.
2. http://www.ilog.com/products/businessrules/index.cfm. Last visited: March 2009.
3. M. Adams, A.H.M. ter Hofstede, D. Edmond, and W.M.P. van der Aalst. Worklets: A service-oriented implementation of dynamic flexibility in workflows. In R. Meersman and Z. Tari, editors, *On the Move to Meaningful Internet Systems 2006*, volume 4275 of *LNCS*, pages 291–308. Springer, 2006.
4. M. Bajech and M. Krisper. A methodology and tool support for managing business rules in organizations. *Information Systems*, 30:423–443, 2005.
5. M. ter Beek, S. Gnesi, C. Montangero, and L. Semini. Detecting policy conflicts by model checking UML state machines. In S. Reiff-Marganiec and M. Nakamura, editors, *Feature Interactions in Software and Communication System X*, pages 59–74. IOS Press, 2009.
6. M.H. ter Beek, A. Fantechi, S. Gnesi, and F. Mazzanti. An Action/State-Based Model-Checking Approach for the Analysis of Communication Protocols for Service-Oriented Applications. In *Revised Selected Papers of Formal Methods for Industrial Critical Systems (FMICS'07)*, volume 4916 of *LNCS*, pages 133–148. Springer, 2007.
7. L. Bocchi, S. Gorton, and S. Reiff-Marganiec. Engineering Service Oriented Applications: From StPowla Processes to SRML Models. In J.L. Fiadeiro and P. Inverardi, editors, *Fundamental Approaches to Software Engineering*, volume 4961 of *Lecture Notes in Computer Science*, pages 163–178. Springer Verlag, 2008.
8. M.G. Buscemi, L. Ferrari, C. Moiso, and U. Montanari. Constraint-Based Policy Negotiation and Enforcement for Telco Services. In *TASE 2007*, pages 463–472. IEEE Computer Society, 2007.

9. M.G. Buscemi and U. Montanari. Cc-pi: A constraint-based language for specifying service level agreements. In R. De Nicola, editor, *Programming Languages and Systems (ESOP 2007)*, pages 18–32, 2007.
10. A. Charfi and M. Mezini. Hybrid web service composition: business processes meet business rules. In M. Aiello, M. Aoyama, F. Curbera, and M. P. Papazoglou, editors, *ICSOC*, pages 30–38. ACM, 2004.
11. A. Charfi and M. Mezini. AO4BPEL: An Aspect-oriented Extension to BPEL. In *World Wide Web*, pages 309–344, 2007.
12. S. Gorton, C. Montangero, S. Reiff-Marganiec, and L. Semini. StPowla: SOA, Policies and Workflows. In *Revised Selected Papers of Workshops, ICSOC'07*, volume 4907 of *LNCS*, pages 351–362. Springer, 2007.
13. S. Gorton and S. Reiff-Marganiec. Policy support for business-oriented web service management. In *Web Congress, 2006. LA-Web '06. Fourth Latin American*, pages 199–202, Los Alamitos, CA, USA, Oct. 2006. IEEE Computer Society.
14. 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 *LNCS*, pages 465–470. Springer, 2006.
15. J. Y. Halpern and V. Weissman. Using first-order logic to reason about policies. In *Proceedings of the Computer Security Foundations Workshop (CSFW'03)*, pages 187–201, Los Alamitos, CA, USA, 2003. IEEE Computer Society.
16. IBM. BPEL4WS, Business Process Execution Language for Web Services, version 1.1, 2003.
17. F. Kamoun. A roadmap towards the convergence of business process management and service oriented architecture. *Ubiquity*, 8(14), 2007. ACM Press.
18. N. Koch, P. Mayer, R. Heckel, L. Gonczy, and C. Montangero. UML for service-oriented systems, SENSORIA EU-IST 016004 Deliverable D1.4.a. http://www.pst.ifi.lmu.de/projekte/Sensoria/del_24/ D1.4.a.pdf, 2007.
19. F. Mazzanti. UMC User Guide v3.3. Technical Report 2006-TR-33, Istituto di Scienza e Tecnologie dell'Informazione "A. Faedo", CNR, 2006.
20. C. Montangero, S. Reiff-Marganiec, and L. Semini. Logic-based detection of conflicts in APPEL policies. In F. Arbab and M. Sirjani, editors, *Int. Symp. on Fundamentals of Software Engineering, FSEN 2007, Tehran, Iran*, volume 4767 of *LNCS*, pages 257–271. Springer, 2007.
21. C. Montangero, S. Reiff-Marganiec, and L. Semini. Logic-based conflict detection for distributed policies. *Fundamenta Informaticae*, 89(4):511–538, 2008.
22. C. Montangero and L. Semini. Distributed states logic. In *9^{th} International Symposium on Temporal Representation and Reasoning (TIME'02)*, Manchester, UK, July 2002. IEEE CS Press.
23. C. Montangero, L. Semini, and S. Semprini. Logic Based Coordination for Event–Driven Self–Healing Distributed Systems. In R.De Nicola, G.Ferrari, and G. Meredith, editors, *Proc. 6th Int. Conf. on Coordination Models and Languages, COORDINATION'04*, volume 2949 of *LNCS*, pages 248–262, Pisa, Italy, Feb. 2004. Springer-Verlag.
24. Carlo Montangero and Laura Semini. Barbed model–driven software development: A case study. *Electron. Notes Theor. Comput. Sci.*, 207:171–186, 2008.
25. S. Moser and T. van Lessen. Developing, deploying and running a hello world BPEL process with the Eclipse BPEL designer and Apache ODE, people.apache.org/∼vanto/helloworld-bpeldesignerandode.pdf.
26. A. Mukhija, D. S. Rosenblum, and A. Dingwall-Smith. Dino: Dynamic and adaptive composition of autonomous services. www.cs.ucl.ac.uk/research/dino/, 2007.

27. 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.
28. S. Núñez. ILOG JRules 6.5 brings rules to SOA. InfoWorld: Product Guide: ILOG JRules 2007: Review, 2007.
29. Oasis Organization. Web services business process execution language version 2.0 - primer, 2007.
30. M. Reichert and Peter Dadam. ADEPT flex -supporting dynamic changes of workflows without losing control. *J. Intell. Inf. Syst.*, 10(2):93–129, 1998.
31. S. Reiff-Marganiec and K. J. Turner. Feature interaction in policies. *Comput. Networks*, 45(5):569–584, 2004.
32. 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.
33. F. Siewe, A. Cau, and H. Zedan. A compositional framework for access control policies enforcement. In *FMSE '03*, pages 32–42. ACM Press, 2003.
34. S. Tai. Composing web services specifications: Experiences in implementing policy-driven transactional processes. In *BTW*, volume 65 of *LNI*, pages 547–559. GI, 2005.
35. C. Teubner. The Forrester Wave: Human Centric BPM for Java Platforms, Q3 2007. http://www.forrester.com/Research/Document/Excerpt/-0,7211,38886,00.html, 2007.
36. 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.
37. UMC v3.5. Online: `http://fmt.isti.cnr.it/umc`.
38. 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.