HAL open science

From StPowla processes to SRML models

Laura Bocchi, Stephen Gorton, Stephan Reiff-Marganiec

▶ **To cite this version:**

Laura Bocchi, Stephen Gorton, Stephan Reiff-Marganiec. From StPowla processes to SRML models. Formal Aspects of Computing, 2009, 22 (3), pp.243-268. 10.1007/s00165-009-0118-7 . hal-00504646

HAL Id: hal-00504646
https://hal.science/hal-00504646

Submitted on 21 Jul 2010

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Under consideration for publication in Formal Aspects of Computing

From StPowla Processes to SRML Models

Laura Bocchi[1] and Stephen Gorton[2] and Stephan Reiff-Marganiec[1]

[1]Department of Computer Science, University of Leicester, Leicester, UK
[2]ATX Technologies Limited, London, UK

Abstract. Service Oriented Computing is a paradigm for developing software systems as the composition of a number of services. Services are loosely coupled entities, that can be dynamically published, discovered and invoked over a network. The engineering of such systems presents novel challenges, mostly due to the dynamicity and distributed nature of service-based applications. In this paper, we focus on the modelling of service orchestrations. We discuss the relationship between two languages developed under the SENSORIA project: SRML as a high level modelling language for Service Oriented Architectures, and STPOWLA as a process-oriented orchestration approach that separates core business processes from system variability at the end-user's level, where the focus is towards achieving business goals. A fundamental challenge of software engineering is to correctly align business goals with IT strategy, and as such we present an encoding of STPOWLA to SRML. This provides a formal framework for STPOWLA and also a separated view of policies representing system variability that is not present in SRML.

Keywords: Service Modelling, Policies, Workflows, Service Oriented Architecture.

1. Introduction

Service Oriented Computing (SOC) is a paradigm for developing software systems as the composition of a number of services. Services are loosely coupled entities that can be dynamically published, discovered and invoked over a network. A service is an abstract resource whose invocation triggers a possibly interactive activity (i.e. a session) and that provides some functionality meaningful from the perspective of the business logic [HA04]. A Service Oriented Architecture (SOA) allows services with heterogeneous implementations to interact relying on the same middleware infrastructure. Web Services and the Grid are the most popular implementations of SOA. Exposing software in this way means that applications may outsource some functionalities and be dynamically assembled, leading to massively distributed, interoperable and evolvable systems.

The engineering of service-oriented systems presents novel challenges, mostly due to this dynamicity [WBC+07]. In this paper we focus on the modelling of orchestrations. An orchestration is the description of the executable pattern of service invocations/interactions to follow in order to achieve a business goal.

Considering this view, we encounter both: more typical Computer Science issues as well as a business oriented perspective – each with their own challenges – but also a need to bridge the gap between the two. In the light of this, we discuss the relationship between two modelling languages for service oriented systems developed in the context

Correspondence and offprint requests to: Laura Bocchi, Department of Computer Science, University of Leicester, University Road, Leicester LE1 7RH, UK. e-mail: bocchi@mcs.le.ac.uk

of SENSORIA, an IST-FET Integrated Project on Software Engineering for Service-Oriented Overlay Computers: the Sensoria Reference Modelling Language (SRML) [FLB06, ABFL07] and STPOWLA: the Service-Targeted, Policy-Oriented WorkfLow Approach [GMRMS07]. The former addresses the Computer Science needs of providing models at a high level of abstraction but with the possibility of presenting quite refined descriptions. The latter is oriented towards the business user. Their combined use provides an approach to bridge the gap between Computer Science and Business needs.

SRML is a high-level modelling language for SOAs whose goal is "to provide a set of primitives that is expressive enough to model applications in the service-oriented paradigm and simple enough to be formalised" [FLB06]. SRML aims at representing the various foundational aspects of SOC (e.g. service composition, dynamic reconfiguration, service level agreement, etc.) within one integrated formal framework. A declarative semantics has been provided in [AF08, LFB07] that maps SRML to mathematical domains that make precise the meaning of the different constructs made available in SRML. In particular, [AF08] provides a formal computational model for SRML which is being mapped into a logic adapted from μUCTL, a formalism being developed within SENSORIA for supporting qualitative analysis [GM05].

STPOWLA is an approach to process modelling for service-oriented systems. It has three ingredients: workflows to express core processes, services to perform activities and policies to express variability. Workflows are expressed using a graphical notation, such as in [GRM06c] or UML activity diagrams[1]. Policies can make short-lived changes to a workflow instance, i.e. they last for the duration of the workflow instance and usually will be made during the execution of the instance, rather than applied to the overall workflow model. STPOWLA allows for both functional and non-functional changes to a workflow, the former have been introduced in [BGR08]. The non-functional changes are called *refinement* and provide a means to select the most appropriate service based on the current environment when a service is invoked. The functional changes are termed *reconfiguration* and are short lived structural changes to a workflow instance.

G2

Having considered the two languages in more detail, we can now add a few words to expand on the motivatation why both are needed as a way to address the needs of the two communities mentioned above. One can say that SRMLis complete in its expressive power with respect to the systems we intend to model. While expressivity is clearly an issue to a Computer Scientist, usability is the more important factor for Business Analysts. STPOWLA addresses usability partly in making use of graphical notations and more crucially in being modular in that the basic workflow and the policies capturing variability are kept seperate while SRMLis essentially flat in that it merges both into the same description.

The encoding of STPOWLA into SRML, on the one hand provides a formal framework to STPOWLA. Business processes modelled in STPOWLA can be then represented as SRML models and either being analysed alone or as part of more complex modules, where they are composed with other SRML models with heterogeneous implementations (e.g. SRML models extracted from existing BPEL processes [BHLF07]).

A second reason for the encoding is providing a higher layer to the modelling of orchestrations in SRML that includes a process-based approach to the definition of a workflow schedule, a separated view of policies, that had not been yet considered in SRML, and the inter-relation between workflow and policies.

This paper extends on [BGR08] by introducing a mapping for the refinement aspects of STPOWLA and providing more detail on the reconfiguration mapping to enhance clarity. Furthermore we consider an example for the use of SRML and STPOWLA in combination based on an industrial case study from the SENSORIA project [sen]. Using the example we present the envisaged methodology of using the two languages.

In this paper, we give an overview of the STPOWLA approach, including the extension for workflow reconfigurations initially proposed in [BGR08] in section 2. We describe the main concepts of SRML, with respect to STPOWLA in section 3. We then provide an encoding of basic workflow control flow constructs in section 4, and proceed to describe the mapping of STPOWLA reconfiguration and refinement policies to SRML in section 5. We present the forseen methodology for the combined use of STPOWLA and SRML in 6.1 by presenting a case study. Related work and our position relative to these efforts in section 7 and a summary and conclusion in section 8 round the paper off.

2. Specifying and Reconfiguring StPowla Workflows

In this section, we give a brief introduction to the main concepts of STPOWLA, highlighting the motivation and then focusing on workflows, and both refinement and reconfiguration policies.

[1] http://www.agilemodeling.com/artifacts/activityDiagram.htm

```
policyName
appliesTo task_id
     when task_entry
       do req(main, params, [])
```

Fig. 1. A STPOWLA task's default policy.

2.1. Overview

STPOWLA has been designed with the business user in mind – typically businesses follow some business process which is usually described as a workflow. However, the processes are often subject to temporary changes (for example different procedures for holiday cover or large orders) as well as overarching rules applicable to all processes (for example a travel policy requiring employees to travel 2nd class). Of course the tasks in the process need to be executed, which is often done by automated procedures. With this in mind STPOWLA has three ingredients: workflows, policies and SOA. Workflows specify core business processes, in which all task requirements are satisfied by services, that is a service (possibly human) can be invoked to complete the task.

We model a workflow using a graphical notation presented in [GRM06c]. However, for the purpose of this paper we also adapt the more concise textual grammar presented in [BGR08] which shows how complex processes can be composed:

WF	$::=$	$start; P; end$	root process
P	$::=$	T	simple task
	$\mid$	$P; P$	sequence
	$\mid$	$\lambda^? P : P$	condition and simple (XOR) join
	$\mid$	$FJ(m, \{P, \mathcal{B}\}, ..., \{P, \mathcal{B}\})$	split and complex (AND) join
	$\mid$	$SP(P, ..., P)$	strict preference
	$\mid$	$RC(P, ..., P)$	random choice
	$\mid$	$Scope(P)$	scope

Briefly, a workflow has a start and end node nesting the process. The simplest process consists of just one task, more interesting processes contain operators that allow sequencing of processes, a conditional branching of the flow with a simple join or a split into parallel processes with a complex merging operation. There are also operators to specify strict preferences or random choice. A more formal description of the semantics of each construct is presented together with with a description of the relevant SRML transition in section 4. We have mentioned in an earlier paper [GMRMS07] that the choice of workflow notation in STPOWLA is essentially insignificant.

Policies are either Event-Condition-Action (ECA) rules (in which case they require a trigger), or goals (essentially ECAs without triggers). The purpose of policies is to express system variability. Policies are written in APPEL [RMTB05], a policy description language with formal semantics via a mapping to $\Delta DSTL$ [MRMS07, MRMSon]. They are written by the end (business) user and are combined with the workflow at execution time. Policies can express refinement and reconfiguration and we will return our attention to each of these types in the next two subsections. Generally policies have the following shape:

```
policyName
appliesTo task_id
     when trigger
       if condition
       do action
```

Additionally to the user specified policies, each workflow task has a default policy as in Fig. 1. The semantics of the `req` function are essentially to execute the processing of the task, as specified with functional requirements described in the `main` argument, in accordance with invocation parameters in the second argument and keeping to default SLR (Service Level Request) constraints in the third argument (that is no value is specified in the policy).

Workflows are executed by an enhanced workflow engine and at specific trigger points the policy engine is invoked; the policy engine will check for applicable policies and apply the relevant actions on the workflow. Clearly the identification of a common set of triggers for ECA policies is of interest, as these present the interaction point between the policies and the workflow at runtime. We have identified the following as valid triggers for reconfiguration policies as this typeof policy can make changes to the workflow at large:

Function Syntax	*Informal Description*
`req(action, [args], [SLR])`	Request a service to fulfil *action* that satisfies Service Level Requirements *SLR*.

Table 1. Appel policy: refinement action

- Workflow entry/success/failure/abort;
- Task entry/success/failure/abort;
- Service entry/success/failure.

Refinement policies are more local: they only influence the selection of the service for the task to be executed. Hence the only trigger that needs to be considered for refinement policies is 'Task entry'.

Note that in StPowla, we view services as a black box, i.e. they are invoked with the required data, but we cannot intervene in their processing until they respond (or maybe time out).

2.2. Refining Tasks with Policies

Refinement policies in StPowla have been the focus of previous work [GMRMS07]. The overall idea is that policies can specify additional criteria on a service. In StPowla we refer to such extra requirements as Service Level Requests (SLR) – in many ways they are similar to what one would understand as Service Level Agreements (SLA). The similarities are focused on what is intended through them and the kind of issues that are expressed in these requests. The overall aim of an SLR is that the chosen service provides the expected criteria; the criteria are often non-functional attributes of a service. The main difference to SLAs is that no agreement has been found: there might not be a service that fulfils the requested service levels, in which case negotiation or a suitable other technique needs to be applied to resolve the issue.

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.

Action `req` 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.

Informally `req` request to find a service as described by the first and third arguments, bind it, and invoke it with the values in the second argument. 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.

2.3. Reconfiguring Workflows with Policies

A workflow reconfiguration is the structural change of a workflow instance. In StPowla, a policy can express a reconfiguration rule based on a number of available functions, as described in Table 2. These changes are short-lived, i.e. they only affect the workflow instance and not the overall workflow model.

As an example, consider a supplier whose business process is to receive an order from a registered customer, and then to process that order (which includes collecting, packing and shipping the items, plus invoicing the client). There are no extra constraints on each task, therefore the default task policies are effectively "empty".

Now consider that under certain conditions (e.g. financial pressure), a financial guarantee is required from all customers whose order is above a certain amount. We may have the following policy:

Function Syntax	*Informal Description*
`fail()`	Declare the current task to have failed, i.e. discard further task processing and generate the `task_failure` event.
`abort()`	Abort the current task and progress to the next task, generating the `task_abort` event.
`block(s, p)`	Wait until predicate `p` is true before commencing scope `s`.
`insert(x, y, z)`	Insert task or scope `y` into the current workflow instance after task `x` if `z` is true, or in parallel with `x` if `z` is false.
`delete(x)`	Delete scope `x` from the current workflow instance.

Table 2. APPEL policy: reconfiguration actions

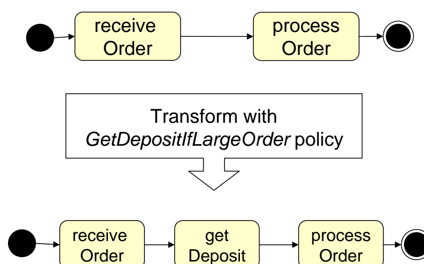


Fig. 2. A simple reconfiguration example where a core business process is transformed via the insertion of the *getDeposit* task after the *receiveOrder* task. The transformation rule comes from a policy.

```
GetDepositIfLargeOrder
      appliesTo receiveOrder
           when task_completion
             if receiveOrder.orderValue > 250000
             do insert(requestDeposit, receiveOrder, false)
```

Intuitively, this policy (named *GetDepositIfLargeOrder*) applies to the *receiveOrder*. It says that when the task completes successfully and the attribute *orderValue* (bound to that task) is above £250000, then there should be an action. The action in this case is the insertion of a task *requestDeposit* into the workflow instance after (not in parallel to) the *receiveOrder* task. The workflow instance thus undergoes the transformation as shown in Fig. 2.

3. Encoding of StPowla to SRML - Foundational Concepts

In SRML composite services are modelled through *modules*. A module declares one or more components, that are tightly bound and defined at design time, a number of requires-interfaces that specify services, that need to be provided by external parties, and (at most) one provides-interface that describes the service that is offered by the module. A number of wires establish interaction protocols among the components and between the components and the external interfaces. Figure 3 shows the SRML module ProcurementService which includes: one provides-interface CR (i.e., the interface of the service provided to the customer), one requires-interface (i.e., the interface of the service that we will discover at run-time) and two components, BP and PI, that orchestrate the interactions among the parties.

The internal nodes of a SRML module can reside at three different layers (top layer, service-oriented layer and bottom layer). Layers in SRML are architectural abstractions that reflect different levels of organisation and change. Each layer uses the layer underneath. The top layer uses the service-oriented layer to achieve some business goal. For example, an application could be designed for the same organization that intend to use it and not for being published as a service. We call this type of applications *activity*. The creation of the instance of an activity is triggered by a node that belongs to the top layer and not by a provides-interface. The service oriented layer uses the bottom layer which typically includes entities which are persistent as far as the life cycle of the activities is concerned, and can be shared by multiple instances of the same activity (e.g., a database shared by all the instances of a service).

Here we focus on SRML modules where nodes reside only at the service-oriented layer. This simplification is

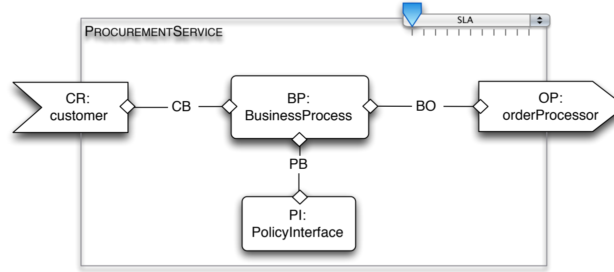


Fig. 3. The structure of a SRML module for the procurement service example

done without loss of generality since it does not have any influence on the encoding. In fact, StPowla does not make any distinction about the type of entity that performs a task (e.g., dynamically discovered service vs persistent resource). The role of the different layers will become interesting in the discussion on the methodology in Section 6.1. The interested reader can refer to [BFL08b] for more details on the layered structure of SRML modules.

Components, external interfaces, wires and interfaces of the different layers are specified in terms of *Business Roles*, *Business Protocols*, *Interaction Protocols*, and *Layer Protocols* respectively. The specifications define the type of the nodes. Each specification provides a slightly different style of behavioural description. A business role defines an execution pattern involving the interactions that it declares in its signature, what we call an orchestration. Business protocols provide a set of properties that abstracts from details of the executable process implemented by the orchestration (e.g., the local state) and describe the behaviour that can be expected of the service (in case of provides-interface) or specify the behaviour that is expected (in the case of requires-interface) of the external party. The interaction protocols define a collection of properties that establish how the interactions are coordinated, which may include routing events or transforming sent data to the format expected by the receiver. However, each language has been captured in the computational model presented in [AF08], which defines the activity of a configuration of SRML components in terms of transition systems where transitions represent the sending, receiving and processing of events by the entities involved in the business activity. The logic $UCTL$ [GM05] is being used to reason about such transition systems. The aim is to provide (1) a notion of correctness for service modules (i.e., the properties of a provides-interface are entailed by the body of a module, assuming the properties described in the requires/uses-interfaces), (2) a way of formalising the matching of provides/requires-interfaces, and (3) a means for validation of activity and service design.

3

In this paper we provide an encoding to derive, from a business process specified in StPowla, an SRML component that we call BP, of type BusinessProcess and a second component PI, of type PolicyInterface that is connected to BP and represents the interface through which it is possible to trigger policies that modify the control flow. PI supports the set of interactions used to trigger a workflow modification in the component BP. Figure 3 illustrates the structure of the SRML module representing the workflow and policies in the procurement example described earlier in this section.

Components are instances of business roles specified in terms of (1) the set of supported interactions, and (2) the way in which the interactions are orchestrated. The remainder of this section provides an overview of business roles. The overview will not include the other types of SRML specification as they are not concerned in the encoding.

3.1. Business Roles: the Interactions

SRML supports asynchronous two-way conversational interactions: **s&r** denotes interactions that are initiated by the co-party, which expects a reply, **r&s** denotes interactions that are initiated by the party, which expects a reply from its co-party. SRML supports also asynchronous one-way and synchronous interactions that are not discussed here as they are not required for the work presented in this paper.

This is followed by the specification of the interactions supported by PolicyInterface, each corresponding to one of the StPowla functions in Table 2. The business role BusinessProcess supports the complementary interactions (i.e. **r&s** instead of **s&r**) plus, other interactions that occur with the external parties. Each interaction can have 🔔-parameters for transmitting data when the interaction is initiated and ✉-parameters for carrying a reply (which in this case are not used as the reply event does not carry data). The index i represents a key-parameter that allows us to handle occurrences of multiple interactions of the same type (as in SRML every interaction event must occur at most once). In this case, we allow PI to trigger more instances of policy functions of the same type.

```
BUSINESS ROLE PolicyInterface is
_____________________________________
INTERACTIONS
        s&r delete[i:natural]
                🔔 task:taskId
        s&r insert[i:natural]
                🔔 task:taskId
                   newTask:taskId
                   c:condition
        s&r block[i:natural]
                🔔 task:taskId
                   c:condition
        s&r fail[i:natural]
                🔔 task:taskId
        s&r abort[i:natural]
                🔔 task:taskId
```

3.2. Business Roles: the Orchestration

The way the declared interactions are orchestrated is specified through a set of variables that provide an abstract view of the state of the component, and a set of transitions that model the way the component interacts with its co-parties. For instance, the local state of the orchestrator is defined as follows:

```
local
start[root],start[x],start[ro],...:boolean, ...
state[root],state[x],state[ro],...:[toStart,running,exited]
```

A module can define an initialisation condition for the each component. For example, the module ProcurementService may define the following initial state for the component OR:

```
start[root]=true
∧ start[x]=start[ro]=...=false
∧ state[root]=state[x]=state[ro]=...=toStart ∧ ...
```

Similarly, a termination condition may specify the situations in which the component has terminated any activity. The behaviour of components is described by transition rules. Each transition has a name, and a number of other features:

```
transition policyHandlerExample
    triggeredBy samplePolicy🔔[i]
    guardedBy state[samplePolicy🔔[i].task] = toStart
    effects policy[samplePolicy🔔[i].task]' ∧ ......
    sends samplePolicy✉[i]
```

5, 6

triggeredBy is a condition, typically the occurrence of a receive-event or a state condition, which triggers the execution of the transition. In the example we engage in the *policyHandlerExample* transition when we receive the initiation of the interaction *samplePolicy* (i.e., $samplePolicy$🔔$[i]$).

guardedBy is a condition that identifies the states in which the transition can take place. For instance, the *policyHandlerExample* transitions should only be taken when the involved task is in state $toStart$ (i.e. is not in execution and it has not been executed yet). The involved task is identified by the parameter $task$ of the interaction $samplePolicy$ (i.e., $samplePolicy$🔔$[i].task$).

effects concern changes to the local state. We use var' to denote the value the state variable var has after the transition.

sends is a sentence that describes the events that are sent and the values taken by their parameters. In the example we invoke the *samplePolicy* reply event (i.e., $samplePolicy$✉$[i]$) to notify of the correct management of the policy.

3.3. Constraints for Service Level Agreement in a SRML Module

SRML offers primitives for modelling the dynamic aspects concerned with session management and service level agreement, which together we call configuration policies. The *external configuration policy* concerns the constraints that the process of discovery, negotiation and binding must satisfy to establish service level agreements (SLA) with service providers. The external configuration policy models an orthogonal aspect with respect to the orchestration. Specifically, it defines a set of non-functional properties to be considered when, in a specific point of the orchestration process, the run-time discovery of an external service for outsourcing the execution of a specific task is required.

7

In SRML, we use the algebraic approach developed in [SUF97] for constraint satisfaction and optimization. In the following example we use a constraint system where the degree of satisfaction has fuzzy values, i.e. it takes value in the interval $[0, 1]$.

EXTERNAL POLICY

```
SLA VARIABLES
        OP.LOCATION, CR.LOCATION
CONSTRAINTS
        Closeness is <{OP.LOCATION, CR.LOCATION},def2> s.t.
                 if distance(OP.LOCATION, CR.LOCATION)<50 then def2(n,m)=1,
                 otherwise def2(n,m)=500/n
```

In order to define the constraints that we wish to apply to the module ProcurementService, we use the SLA variables OP.LOCATION and CR.LOCATION which are the locations of the order processor and customer, respectively. We define only one constraint Closeness, which minimises the distance between the customer CR and the order processor OP. The best degrees of satisfaction are when the distance is less than 50 miles. Otherwise they are inversely proportional to the distance. The function distance returns the distance between two locations.

For each potential order processor (i.e., the service, among the published ones, whose provides-interface matches with OP, of type OrderProcessor), the set of constraints has to be solved. The solution assigns a degree of satisfaction to each possible tuple of values for the SLA variables. Negotiation in our framework consists in finding an assignment that maximizes the degree of satisfaction. Hence, the outcome of the negotiation between ProcurementService and the potential partner is any tuple that maximizes the degree of satisfaction. Selection then picks a partner with a service level agreement that offers the best degree of satisfaction.

4. Encoding StPowla Workflows in SRML

In this section we present an encoding from the control constructs of StPowla to SRML orchestrations. Our focus is on the control constructs and we abstract from the interactions of the service and from the semantics of the simple activities of the workflow tasks.

StPowla represents a business process as the composition of a number of tasks, either simple (e.g. interactions with services) or complex (e.g. coordinating other tasks by executing them in sequence, parallel, etc.). In SRML we associate an identifier, of type $taskId$, to any task. We denote with T the set of all the task indexes in the workflow schedule.

8

For each task identifier x we define the following local variables, used to handle the control flow and coordinate the execution of the tasks:

- $start[x]$ is a boolean variable that, when true, triggers the execution of x,
- $done[x]$ is a boolean variable that signals the successful termination of x and triggers the continuation of the workflow schedule,
- $fail[x]$ is a boolean variable that signals the termination with failure of x and triggers the failure handler.

In general, the next activity in the control flow is executed when the previous one terminates successfully. In case of task failure the flow blocks (i.e. the next task is waiting for a signal of successful termination from the previous task) and the failure signal is collected by a failure handler that possibly involves a number of policies. According to the failure handler, the execution of the process can be terminated, resumed, altered, etc. We leave the specification of the failure handling mechanisms as a future work. Anyway, the construct of strict preference and random choice, that try a number of alternative tasks until one terminates with success, handle the failure signal directly, within the workflow. The scope construct can be extended in the future to support failure handling, as discussed in Section 4.6.

G1,9

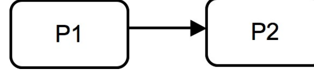


Fig. 4. The sequence control construct in STPOWLA

We will introduce in section 5 a set of transitions, as a part of the orchestration of BP that model the policy handler. The policy handler has the responsibility to enact the modifications of the control flow induced by the policies triggered by PI. The policy handler blocks the normal flow by setting the variable $policy[x] = true$, where x is the identifier of the first task involved in the modification. The variable $policy[x]$ is a guard to the execution of x. We will describe the policy handler more later in this paper, by now it is important to know that when a policy function has to be executed on a task, the task has to be blocked. It is responsibility of the policy handler to reset the flow of execution.

Some policies can be applied only on running processes (e.g. abort) and some others only on tasks that have not started yet (e.g., the deletion). We define a local variable $state[x]$ for every task identifier x which identifies the state of the execution of the transition associated to x by taking one of the following values: $toStart$ (i.e., the execution of the task has not started yet), $running$ (i.e. the task is in execution) and $exited$ (i.e. x has terminated). The initialization conditions for the module set, for each task identifier x, $state[x] = toStart$, see for example the initialization condition for ProcurementService presented in Section 3.1. The state variable $state[x]$ isl be used to ensure that policies act on a task in the correct state of execution (i.e., the deletion of task x can be performed only if $state[x] = toStart$).

12

We consider the simple tasks as black boxes: we are not interested in the type of activity that they perform but only on the fact that a task, for example task x, is activated by $start[x]$, signals its termination along either $done[x]$ or $failed[x]$ and notifies its state along $state[x]$.

The execution of the workflow is started by a special transition *root* that sets $start[x] = true$ where x is the first task in the workflow schedule. The local variables are initialized as follows: $\forall i \in T \setminus root, start[i] = false \wedge start[root] = true, \forall i \in T, done[i] = failed[i] = policy[i] = false$ and $\forall i \in T, state[i] = toStart$.

It follows the encoding of the workflow template $start; P; end$ where P is associated to the task identifier x:

```
transition root
    triggeredBy start[root] ∨ done[x]
    guardedBy ¬ policy[root]
    effects
    start[root]⊃ ¬ start[root]' ∧ state[root]'=running ∧ start[x]'
    ∧ done[x] ⊃ ¬ done[x]' ∧ done[root]' ∧ state[root]'=exited
```

1

The guard of transition x ensures the execution of the transition only if no policy has been triggered on task $root$ (i.e., $policy[root]$ is false). According to the trigger, $root$ is executed twice:

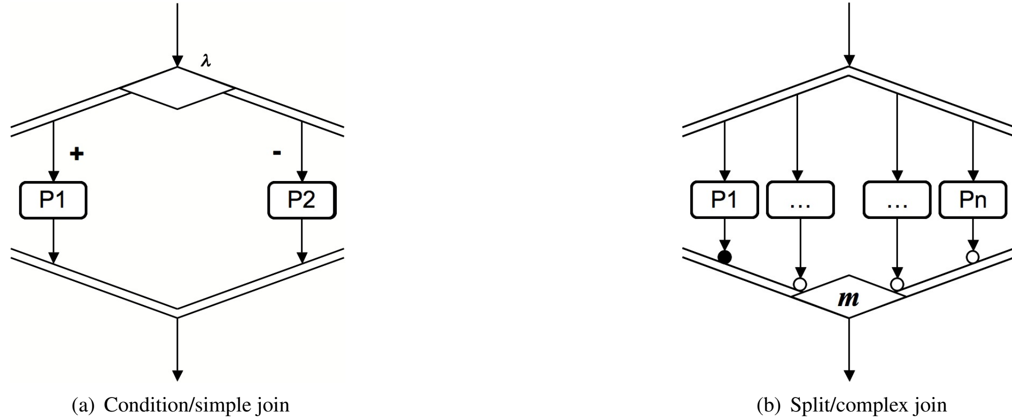
1. at the beginning of the workflow (recall from Section 3.1 that the initialization condition of ProcurementService includes the assignment $start[root] = true$). The transition in this case has the following effects: (1) disabling the the triggering condition of $root$ (i.e., $start[root]$ is set to $false$, (2) setting the state of task $root$ to $running$ and (3) triggering the transition for task x by setting $start[x]$ to $true$.
2. when task x terminates (i.e., $done[x] = true$). The transition in this case has the following effects: (1) disabling the termination signal for x is disabled (i.e., $done[x]$ is set to $false$), (2) enabling the termination signal for $root$ iand (3) setting the state variable for $root$ to $exited$.

11

4.1. Sequence

The sequence operator $P_1; P_2$, illustrated in Figure 4, first executes P_1 and, after the successful termination of P_1, executes P_2. We remark that failures are not handled in this document and will be addressed in the future. The encoding of the sequence construct in SRML is as follows. The sequence is encoded in the following SRML transition, with task identifier x, which triggers the execution of the first task, with task identifier $p1$, then collects the termination signal from $p1$ and triggers the execution of the second subprocess, with task identifier $p2$:

```
transition X
    triggeredBy start[x] ∨ done[p1] ∨ done[p2]
    guardedBy ¬ policy[x]
```



(a) Condition/simple join (b) Split/complex join

Fig. 5. Multiple branches constructs in STPOWLA

```
effects
start[x] ⊃ ¬ start[x]′ ∧ state[x]′=running ∧ start[p1]′
∧ done[p1] ⊃ ¬ done[p1]′ ∧ start[p2]′
∧ done[p2] ⊃ ¬ done[p2]′ ∧ done[x]′ ∧ state[x]′=exited
```

11

Transition X is executed three times:

1. when $start[x]$ is true. The transition in this case has the following effects: (1) disabling the triggering condition $start[x]$, (2) changing the state of task x to $running$ and (3) enabling the triggering condition $start[p1]$.
2. when $done[p1]$ is true (i.e., after $p1$ has been executed). The transition in this case has the following effects: (1) disabling the termination signal $p1$ and (2) enabling the triggering condition $start[p2]$.
3. when $done[p2]$ is true. The transition in this case has the following effects: (1) disabling the termination signal for $p2$, (2) enabling the termination signal for x and (3) setting the state of task x to $exited$.

11

4.2. Condition and Simple Join (XOR)

The condition and simple join construct $\lambda^? P_1 : P_2$, illustrated in Figure 5(a), consists of the combination of the flow junction, that diverts the control flow down one of two branches P_1 and P_2, represented by the task identifiers $p1$ and $p2$, respectively, according to a condition λ, and the flow merge of a number of flows where synchronization is not an issue. The condition and simple join are encoded into the following SRML transition:

```
transition X
    triggeredBy start[x] ∨ done[p1] ∨ done[p2]
    guardedBy ¬ policy[x]
    effects
    start[x] ⊃ ¬ start[x]′ ∧ state[x]′=running
                   ∧ (λ ⊃ start[p1]′) ∧ (¬ λ ⊃ start[p2]′)
    ∧ done[p1] ⊃ ¬ done[p1]′ ∧ done[x]′ ∧ state[x]′=exited
    ∧ done[p2] ⊃ ¬ done[p2]′ ∧ done[x]′ ∧ state[x]′=exited
```

11

Transition X is executed twice:

1. when $start[x]$ is true. The transition in this case has the following effects: (1) disabling triggering condition $start[x]$, (2) setting the state of x to $running$ and (3) triggering either $p1$ or $p2$ depending on the condition λ.
2. when either $done[p1]$ or $done[p1]$ is true (either $p1$ or $p2$ was executed)). The transition in this case has the following effects: (1) disabling the termination signal for $p1$ or $p2$, (2) enabling the termination signal of x and (3) setting the state variable of x to $exited$.

11

(a) Strict preference (b) Random choice

Fig. 6. Strict preference and random choice in STPOWLA

4.3. Split and Complex Join (AND)

The split and complex join construct $FJ(m, \{P_1, \mathcal{B}_1\}, \ldots, \{P_n, \mathcal{B}_n\})$ consists of the combination of the flow split, that splits the control flow over many branches, and the conditional merge, that synchronizes two or more flows into one. The value of m, that is statically determined, represents the minimum number of branches that have to be synchronized. Furthermore, any branch is associated to a boolean $\mathcal{B}_i$ that determines whether the $i-th$ branch is mandatory in the synchronization. The graphical notation of the construct is illustrated in Fig. 5(b).

The encoding is as follows. Let S be the set, with cardinality n, of the task indexes associated to the branches of the split/join. Let the identifiers for the subtasks of x to range over $p1, \ldots, pn$. Let N be the set of indexes of the necessary tasks and $m \in \mathbb{N}$ be the minimum number of branches that have to be synchronized. We assume that $m \leq |N|$. The complex join is encoded in the following SRML transition, where $Kcomb$ is the set of $(m-|N|)-subsets$ of $S \setminus N$.

transition X
  triggeredBy start[x] $\vee$ ($\wedge_{i\in N}$done[pi]$\wedge$($\vee_{K\in Kcomb}$($\wedge_{k\in K}$done[pj])))
  guardedBy $\neg$ policy[x]
  effects
  start[x] $\supset$ $\neg$ start[x]′ $\wedge$ state[x]′=running $\wedge_{i\in[1,\ldots,n]}$ start[pi]′
  $\wedge$ $\neg$ start[x] $\supset$ done[x]′ $\wedge$ state[x]′=exited $\wedge_{i:[1..n]}$($\neg$ done[pi]′)

The transition above is parametric with respect to N and K_{comb} in order to model the general case. In a real workflow schedule the general construct would be instantiated (at design-time) and the parameters in the conjuction/disjunctions would disappear (e.g., if $N = \{1, 2\}$ the term $\wedge_{i\in N} done[pi]$ becomes $done[p1] \wedge done[p2]$. Transition X is executed twice:

1. when $start[x]$ is true. The transition in this case has the following effects: (1) disabling the triggering condition $start[x]$, (2) setting the state of task x to $running$ and (3) enabling the triggering condition for each sub-task i by setting $start[i]$ to $true$.
2. in case of successful termination of all the necessary subtasks (i.e., $\wedge_{i\in N} done[pi]$) and of a number of tasks greater or equal to m (i.e., $\vee_{K\in Kcomb}(\wedge_{k\in K} done[pj])$). The transition in this case has the following effects: (1) enabling the termination signal for x, (2) setting the state of x to $exited$ and (3) disabling the successful termination of all the subtasks.

4.4. Strict Preference

The strict preference $SP(P_1, \ldots, P_n)$, illustrated in Figure 6(a), attempts the tasks $P_1, \ldots, P_n$ one by one, in a specific order, until one completes successfully. In this case, with no loss of generality we consider the tasks ordered by increasing index numbers.

The strict preference is encoded in the following SRML transition:

transition X
  triggeredBy start[x] $\vee_{i:[1..n]}$(done[pi] $\vee$ failed[pi])
  guardedBy $\neg$ policy[x]
  effects
  start[x] $\supset$ $\neg$ start[x]′ $\wedge$ state[x]′=running $\wedge$ start[p1]′
  $\wedge_{i:[1..n-1]}$failed[pi] $\supset$ $\neg$ failed[pi]′ $\wedge$ start[p(i+1)]′

```
∧ failed[pn] ⊃ ¬ failed[pn]' ∧ failed[x]' ∧ state[x]'=exited
∧ ∨_{i:[1..n]}done[pi] ⊃ done[x]' ∧  state[x]'=exited ∧_{i:[1..n]}¬ done[pi]'
```

11

Transition X is executed a number of times in the following cases:

1. when the task x is triggered. The transition in this case has the following effects: (1) disabling the triggering condition $start[x]$, (2) setting the state of x to $running$ and (3) triggering the first sub-task $p1$ of x.
2. when either any of the tasks pi terminates with failure ($failed[pi] = true$) or success ($done[pi] = true$).
   - If the task failed (i.e., $failed[pi] = true$) and it was not the last task pn, then the transition has the following effects: (1) disabling the termination variable of pi and (2) enabling the next task by setting $start[p(i+1)]$ to true.
   - If the last task failed ($failed[pn] = true$) then the transition has the following effects: (1) disabling the signal of failed termination of pn, (2) enabling the signal $failed[x]$ of failed termination for x and (3) setting the state of x to $exited$.
   - If any of the sub-tasks terminated successfully ($done[pi] = true$) then the transition has the following effects: (1) enabling the signal of successful termination for x, (2) setting the state of x to $exited$ and (3) disabling the successful terminations of all the sub-tasks.

11

4.5. Random Choice

The random choice $RC(P_1, \ldots, P_n)$, illustrated in Figure 6(b), attempts the tasks $P_1, \ldots, P_n$ simultaneously and completes when one completes successfully. The random choice is encoded in the following SRML transition:

```
transition X
    triggeredBy start[x] ∨_{i:[1..n]}(done[pi])∨ (∧_{i:[1..n]}(failed[pi]))
    guardedBy ¬ policy[x]
    effects
    start[x] ⊃ ¬ start[x]' ∧ state[x]=running ∧_{i:[1..n]}start[pi]'
    ∧ (∧_{i:[1..n]}failed[pi])⊃ failed[x]'∧ state[x]'=exited ∧_{i:[1..n]}¬ failed[pi]'
    ∧ (∨_{i:[1..n]}done[pi]) ⊃ done[x]' ∧ state[x]'=exited
                                  ∧_{i:[1..n]}(¬ done[pi]' ∧ ¬ failed[pi]')
```

11

Transition X is executed a number of times, in the following cases:

1. when the task x is triggered. The transition in this case has the following effects: (1) disabling the triggering condition $start[x]$, (2) setting the state variable of x to $running$ and (3) enabling the triggering condition of all the sub-tasks.
2. when any of the tasks pi terminates with success (i.e., $done[pi] = true$). The transition in this case has the following effects: (1) enabling the successful termination of x, (2) setting the state of x to $exited$ and (3) disabling the successful and faulty terminations for all the sub-processes.
3. when all the sub-tasks pi terminates with failure (i.e., $failed[pi] = true$). The transition in this case has the following effects: (1) enabling the faulty termination of x, (2) setting the state of x to $exited$ and (3) disabling the faulty terminations for all the sub-processes.

11, 15

4.6. Scope

The scope construct $Scope(P)$, where P is associated to the task identifier y behaves similarly to the root process. It is illustrated in Figure 7. Within larger business processes it often makes sense to group parts of the process together as these are controlled by one division in the company or the tasks are more intrinsically linked together. We represented scopes separately, even though at the moment little is done with them. The scope construct will in the future be extended to include notions of compensation and fault handling (that is they will form a way to express long running transactions), however this aspect is beyond the scope of this paper. An idea is to support the mechanism of fault handling by taking into account the semantics given in [BLZ03] through an extension of the asynchronous $\pi-$ calculus. The intuition is to associated at design time a scope with two process: a compensation process and a fault handler.

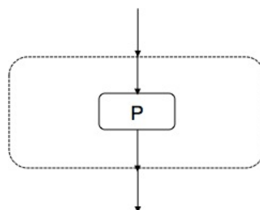


Fig. 7. The scope control construct in STPOWLA

The compensation process can be triggered only after the successful completion of the activity of the scope, in order to compensate the effects of the scope. The failure handler is triggered in case of failure during the execution of the scope and include (among other activities that depend on the specific process that is being modelled) the triggering of the compensations of all the scopes, enclosed in the failing scope, that have already terminated successfully.

The Scope choice is encoded in the following SRML transition:

```
transition X
    triggeredBy start[x] ∨ done[y]
    guardedBy ¬ policy[x]
    effects
    start[x] ⊃ ¬ start[x]' ∧ state[x]'=running ∧ start[y]'
    ∧ done[y] ⊃ ¬ done[y]' ∧ done[x]' ∧ state[x]'=exited
```

11

Transition X is executed twice:

1. when the triggering condition $start[x]$ is enabled. The transition in this case has the following effects: (1) disabling the triggering condition $start[x]$, (2) setting the state of task x to $running$ and (3) enabling the triggering condition for the sub-task y.
2. when y terminates with success (i.e., $done[y] = true$). The transition in this case has the following effects: (1) disabling the successful termination of y, (2) enabling the signal of successful termination for x and (3) setting the state of x to $exited$.

11

5. Encoding StPowla Policies in SRML

5.1. Refinement Policies

Recall that in STPOWLA refinement policies are those requesting the `req(action, [args], [SLR])` as action. The element that requires encoding in SRML is the Service Level Requirements list. This list contains expressions that specify acceptable values for a domain specific attribute: e.g. `[cup_temperature = warm]` which specifies that the chosen service (here a beverage service) should offer warm cups.

In SRML we have seen that the `EXTERNAL POLICY` element allows to express such issues. The expression language in SRML is much more powerful than what STPOWLA offers currently: in SRML complex behaviour such as that seen in the example in Section 3.3 can be expressed. The example did show a case where complete satisfaction was achieved by a delivery distance of less than 50 miles, and otherwise satisfaction did decrease in line with distance. In STPOWLA currently only simpler relations such as less, equal or more can be expressed but there will be ongoing work in enhancing the mechanisms in STPOWLA.

In general the mapping from STPOWLA to SRML involves creating an external policy for SLRs where for each STPOWLA attribute an `SLAVARIABLE` is created in SRML. The relation and values are then captured as a `CONSTRAINT` in the SRML policy rule.

Let us consider, for example, the following STPOWLA policy for the procurement service (see workflow in Figure 2):

```
CheapService
appliesTo processOrder
     when on_task_entry
     req(mail,[],[ServiceCost<10])
```

The policy *CheapService* ensures that the cost of the transaction with the service for processing the order is less then $10£$. In SRML, we can express *CheapService* as a constraint that applies to the requires-interface OP of the module ProcurementService (see Figure 3). Since STPOWLA, at the moment, only allows to express sharp requirements, we use a constraint system where the degree of satisfaction has boolean values (i.e. $\{0, 1\}$).

```
EXTERNAL POLICY
________________________________________
SLA VARIABLES
        OP.SERVICECOST
CONSTRAINTS
        CheapService is <{OP.SERVICECOST},def1> s.t.
                if n<10 then def1(n)=1,
                otherwise def1(n)=0
```

The constraint uses the SLA variable OP.SERVICECOST (i.e., the price for enacting a transaction with the order processor), which assign degree of satisfaction of 1 if the service costs less then $10£$ and degree of satisfaction of 0 otherwise. The constraint ensures that any service which does not ensure a degree of satisfaction of 1 will not be selected.

5.2. Reconfiguration Policies

One of the aims of this paper is to illustrate how policies can influence the control flow and how this can be modelled in SRML. In this section we discuss the encoding of policies, as described in STPOWLA, into SRML orchestrations. Each interaction is handled, in the orchestration of BP, by one or more transitions that model the policy handler. We will see in detail such transitions when discussing the single interactions, in the rest of this section.

A policy related to a task can have an effect (1) on the state prior to the task execution (i.e. $delete$, $block$ and $insert$) or (2) during the execution of a task (i.e. $fail$ and $abort$). The state of a task is notified through the variable $state[y]$. The policy handler must check that the task is in the correct state according to the specific policy that has to be enacted. The policy handler prevents the execution of either (1) the task or (2) the rest of the task by using the variable $policy[x]$: the condition $\neg policy[x]$ guards the transition(s) corresponding to the execution of task. Notice that for most of the control constructs it is not possible to trigger policies of this second type on atomic tasks whose state changed directly from $toStart$ to $done$.

5.2.1. Delete Task

The deletion of task (i.e. `delete(x)` in STPOWLA) skips the execution of x. The policy manager prevents the execution of x by signaling a policy exception (i.e. $policy[x] = true$).

```
transition policyHandler_delete_1
    triggeredBy delete[i]🔔
    guardedBy state[delete[i]🔔.task] = toStart
    effects policy[delete[i]🔔.task]'
```

Transition $policyHandler_delete_1$ is triggered by the event $delete[i]$🔔, sent by PI. The guard ensures that a task can be deleted only if its execution has not started yet (i.e., its state has value $toStart$). The transition has the effect of setting the value of the variable $policy$ to $true$. This will prevent the regular execution of the deleted task (recall that the execution of each task x is guarded by the condition $\neg policy[x]$). When the triggering condition for task x becomes $true$, the transition $policyHandler_delete_2$ is executed instead of the transition of the deleted task.

```
transition policyHandler_delete_2
    triggeredBy start[x]
    guardedBy P_delete[i]🔔? ∧  delete[i]🔔.task=x
    effects ¬ start[x]' ∧ done[x]' ∧ state[x]' = done
    sends delete[i]✉
```

The guard of $policyHandler_delete_2$ ensures its execution only if a deletion policy has been triggered for x. The effects of the transition are: (1) to disable the triggering condition of x, (2) to notify the correct termination of x

(which in fact has not been executed) and (3) to set the state of x to $done$. The transition also sends the reply event to the interaction $delete$ to notify PI of the completed enactment of the reconfiguration policy.

5.2.2. Block Task

The function `block(x,p)` in STPOWLA blocks a task until p is $true$. In SRML the policy handler prevents x from executing (i.e. *policy[x]* becomes $true$) temporary until p is $true$. The policy handler notifies the enactment of the policy to the environment after that the task has been unblocked.

```
transition policyHandler_block_1
    triggeredBy block[i]🔔
    guardedBy state[block[i]🔔.task] = toStart
    effects policy[block[i]🔔.task]'
```

Transition $policyHandler_block_1$ is triggered by the event $block[i]$🔔, sent by PI. The guard ensures that a task can be deleted only if its execution has not started yet (i.e., its state has value $toStart$). The transition has the effect of setting the value of the variable $policy$ to $true$. This will prevent the regular execution of the blocked task. When the condition specified through condition p, which has been communicated by PI, becomes true then the transition $policyHandler_block_2$ unblocks the task.

```
transition policyHandler_block_2
    triggeredBy block[i]🔔.condition
    guardedBy P_block[i]🔔
    effects ¬ policy[block[i]🔔.task]'
    sends block[i]✉
```

Transition $policyHandler_block_2$ is triggered by the condition $block[i]$🔔$.condition$ and the guard ensures its execution only the task has previously been blocked. The transition has the effect of setting the variable $policy$ for the task to which the policy applied to $false$ so that the task can be executed as soon as its triggering condition becomes true. The transition also sends the reply event to the interaction $block$ to notify PI of the completed enactment of the reconfiguration policy.

5.2.3. Insert

The insertion of a task, represented by the function `insert(x,y,z)` in STPOWLA, inserts the task y in sequence or in parallel with respect to x depending by the value of the boolean variable z. In SRML the insertion is triggered by the interaction $insert[i]$✉ with parameter $insert[i]$✉$.task$ representing the task x, $insert[i]$✉$.insertedTask$ representing the task y and $insert[i]$✉$.condition$ representing the condition z. We assume that the set of tasks that it is possible to insert is determined a priori, in this way we assume that the SRML encoding has a set of transitions for each possible task, including the task to possibly insert, that is executed by setting $start[y]$ to $true$. We introduce in this way a limitation on the number of task types that we can insert and on the fact that a task can be inserted only once (we will manage multiple insertions in the future, when we will encode looping constructs) but we do not provide any limitation on the position of the insertion.

We rely on a function $next : taskId \rightarrow taskId$ that returns, given a task, the next task to execute in the workflow. Such function can be defined by induction on the syntax of STPOWLA defined in Section 2.1.

```
transition policyHandler_insert_1
    triggeredBy insert[i]🔔
    guardedBy  state[insert[i]🔔.task]=toStart
    effects policy[insert[i]🔔.task]'
```

The transition $policyHandler_insert_1$ prevents the execution of the task on which the policy applies (i.e., $insert[i]$✉$.task$) by setting the its policy variable to true. When the task on which the policy applies is triggered, $policyHandler_insert_2$ is executed instead of the regular transition for the task.

```
transition policyHandler_insert_2
    triggeredBy start[x]
    guardedBy P_insert[i]🔔 ∧ insert[i]🔔.task=x
    effects
```

```
insert[i]🔔.condition ⊃ ¬ policy[insert[i]🔔.task]′
∧ ¬ insert[i]🔔.condition ⊃ policy[insert[i]🔔.task]′
                                ∧ start[insert[i]🔔.insertedTask]′
```

The transition $policyHandler_insert_2$ starts the execution of the task on which the policy applies (in parallel with the inserted task if $insert[i]⊠.condition = true$). The transitions $policyHandler_insert_sequence$ and $policyHandler_insert_parallel$ coordinate the execution of the tasks (the one on which the policy applies and the inserted one) in sequence or in parallel, according to the condition.

```
transition policyHandler_insert_sequence
    triggeredBy done[x] ∨ done[y]
    guardedBy P_insert[i]🔔 ∧ insert[i]🔔.condition
     ∧ (insert[i]🔔.task=x ∨ insert[i]🔔.insertedTask=y)
    effects
    done[x] ⊃ ¬ done[x]′ ∧ start[y]′
    ∧ done[y] ⊃ ¬ done[y]′ ∧ start[next(x)]′
    sends
    done[y] ⊃ insert[i]⊠
transition policyHandler_insert_parallel
    triggeredBy done[x] ∧ done[y]
    guardedBy P_insert[i]🔔? ∧ ¬ insert[i]🔔.condition
     ∧ insert[i]🔔.task=x ∧ insert[i]🔔.insertedTask=y
    effects ¬ done[x]′ ∧ ¬ done[y]′ ∧ start[next(block[i]🔔.task)]′
    sends insert[i]⊠
```

Transitions $policyHandler_insert_sequence$ and $policyHandler_insert_parallel$ are similar to regular sequence and paralel transitions (see Section 4.6), but they are guarded by the fact that an insertion policy with positive/negative condition has been triggered in the past. The effects are similar to those of a regular sequence/parallel transitions. A reply event for the interaction $insert$ is sent to notify PI of the completed enactment of the reconfiguration policy when, in the case of $policyHandler_insert_sequence$ transition the inserted task terminates (i.e., $done[y] \supset= true$) and in the case of $policyHandler_insert_parallel$ both of the parallel tasks terminate.

5.2.4. Fail Task

The failure of a task must occur during the execution of the task (it has no effects otherwise). The failure can be triggered autonomously, within the task or induced externally by the execution of the policy $fail$. We consider here the second case.

```
transition policyHandler_fail
    triggeredBy fail[i]🔔
    guardedBy state[fail[i]🔔.task]=running
    effects policy[i][fail[i]🔔.task]′ ∧  state[fail[i]🔔.task]′=failed
    sends fail[i]⊠
```

Transitions $policyHandler_fail$ is triggered by the event $fail[i]$🔔, sent by PI. The guard ensures that a task can fail only if it is currently in execution (i.e., its state has value $running$). The transition has the effect of setting the value of the variable $policy$ to $true$ and the state of the task to $failed$ (so that the normal flow of execution blocks). A reply event for the interaction $fail$ is sent to notify PI of the completed enactment of the reconfiguration policy.

5.2.5. Abort Task

The abortion of a task is similar to a deletion, but it involves a running task. An abort of a task occurring not during its execution has no effects.

```
transition policyHandler_abort
    triggeredBy abort[i]🔔
    guardedBy state[abort[i]🔔.task]=running
    effects policy[abort🔔.task]′  ∧ state[abort[i]🔔.task]′=done
    sends abort[i]⊠
```

Transitions $policyHandler_abort$ is triggered by the event $abort[i]$🔔, sent by PI. The guard ensures that a task can fail only if it is currently in execution (i.e., its state has value $running$). The transition has the effect of setting the value of the variable $policy$ to $true$ and the state of the task to $done$ (so that the flow of execution can continue normally). A reply event for the interaction $fail$ is sent to notify PI of the completed enactment of the reconfiguration policy.

18

5.3. An Example: the Reconfiguration of the Procurement Scenario

The orchestration of the business role BusinessProtocol would consist of the sequence of the tasks request order (i.e. task ro) and process order (i.e. task po).

```
transition X
    triggeredBy start[x] ∨ done[ro] ∨ done[po]
    guardedBy policy[x]
    effects
    start[x] ⊃ ¬ start[x]' ∧ state[x]'=running ∧ start[ro]'
    ∧ done[ro] ⊃ ¬ done[ro]' ∧ start[po]'
    ∧ done[po] ⊃ ¬ done[po]' ∧ done[x]' ∧ state[x]'=exited
```

18

In case of a receive event of type $insert[i]\boxtimes$, triggered by the component PI, with parameter $task$ equal to po, parameter $insertedTask$ equal to gbd (i.e. get deposit), and the parameter $condition$ equal to $true$. The policy handler would: (1) block the execution of ro (preventing in this way ro to trigger its continuation $po = next(ro)$) by setting $policy[ro] = policy[po] = true$, (2) wait for the condition $start[ro] = true$ that is triggered by transition X, (3) since the parameter $condition$ is $true$, the policy handler would unblock ro, (4) the transition $policyHandler_insert_sequence$ would handle the execution of gd after ro and, finally, trigger po by setting $start[po] = true$.

6. Methodology by Example – a case study of using StPowla and SRML

In this section we present a methodology for developing services, which involves STPOWLA and SRML. The discussion on the methodology illustrates the rationale of the work presented in this paper by discussing the benefits of a joint usage of SRML, STPOWLA and the encoding.

Some of the methodological aspects concerning SRML have been discussed in previous work [BLF08], which proposes a process to arrive at (formal) service models in SRML starting from informal (or semi-formal) specifications in notations that are typically described in UML. In Section 6.1 we use the extension of UML use-case diagrams presented in [BLF08] to capture requirements in a service-oriented scenario and derive the structure of SRML models.

From the use-case diagram we can determine the services and resources our application relies on. The definition of the internal structure of the SRML module (i.e. the components and wires that define the internal workflow) depends in general on the portfolio of components already available for reuse within the business organisation. The definition of a complex internal structure from scratch, deriving from the decomposition of the orchestration in a number of coordinated units, can be done, in general, using traditional techniques for Component Based Development.

SRML offers primitives based on events that allow to suitably model those scenarios where process-based modelling could result in over-specification. However, the primitives of SRML are general enough to support process based modelling. For example, [BHLF07] presents an encoding from the process-base style of modelling of WS-BPEL to SRML. Another example is the encoding presented in this paper where STPOWLA provides SRML with a means to define, at a higher level of abstraction, a process-based behaviour (i.e. workflow schedule) which is dynamically reconfigured by policies.

In Section 6.1 we use STPOWLA to model the internal behaviour of the SRML module. The workflows shown in this section use the same workflow notation that we have used earlier and are complete in that notation. To ease readability we have added “swimlanes” to highlight the stake holders as often seen in UML Activity diagrams, but note that these are not used in the translation. By using the encoding we derive the internal structure of the SRML module and the SRML specifications for each component to model the business process. As described in Section 3, the resulting internal structure of the SRML module consists of one main component BP that models the behaviour of the workflow, and the policy handler PI which handles the reconfiguration policies.

The methodology presented through the example involves three different languages that suit the needs of different stages of the modelling process:

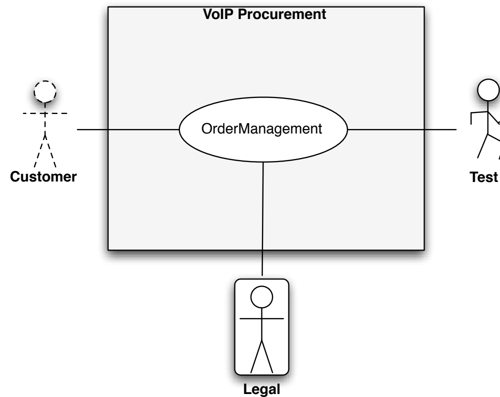


Fig. 8. VoIP Case Study use-case diagram

Requirements UML Use Case diagrams are used to capture the requirements of the service-oriented application that has to be developed. The application may involve a number of related services (i.e., they may share resources and are modelled as a single business unit). The usage of UML Use Case diagrams provides a human friendly notation suitable in this phase which involves the intervention of non experts in the engineering process (e.g., members of the business company commissioning the development of the application). The outcome of this phase is a Use Case diagram. By using the mapping defined in [BLF08] it is possible to derive, from the diagram, the structure for a number of SRML modules where the orchestration and the behavoural specification of the external interfaces is left unspecified. An Eclipse plugin is currently under development to automate this transformation.

Business Modelling StPowla is used to perform the modelling, at the business level, of the orchestration in terms of workflow schedule and reconfiguration policies. This phase can be performed by experts in business modelling who can benefit by the level of abstraction provided by StPowla (i.e., modular definition of control-based process and business policies). The outcome of this phase is a StPowla model of the business process that, by using the encoding to SRML, can provide each of the SRML module structures resulting from the previous phase with an the internal orchestration.

Service-Oriented Modelling The SRML modules obtained in the previous phase can be extended to include the behavioural specifications for the external interfaces, modified to include other components in the internal structure (for example component extracted from implementations in BPEL [BHLF07] or any other language for which an encoding into SRML has been provided), and analyzed with the formal framework provided by SRML.

6.1. Use Case Driven Example

In this section, we present a case study supplied by an industrial partner of the SENSORIA project. The case study is based on a Voice over IP (VoIP) procurement example between a customer and a reseller. If viewed at a higher level, it can represent most, if not all, procurement examples.

The requirements are expressed in Figure 8. Figure 8 uses the notation for use-case diagrams proposed in [BLF08]. The customized icons represent different types of roles that an actor can have in a service-oriented context (e.g., dynamically discovered services, statically bound persistent resources, service requester, etc.).

The diagram models a service, provided to a service requester represented by the actor *Customer*, which manages the order for a VoIP connection. The service relies on the persistent resource *Legal*, which is shared among the different instances of the service, and the external service *Test* which is dynamically discovered and selected according to the customer's location.

As mentioned in Section 3, SRML distinguish among the different types of actors by representing nodes at different layers. The SRML module derived by the diagram in Figure 8 is illustrated in Figure 9. The module has one provides-interface CU of type Customer, one bottom-layer interface LE of type Legal and one requires-interface TE of type Test. The internal structure of the module is defined according to Section 3: the component BP of type BusinessProcess defines the base workflows for orchestrating the service, the component PI of type PolicyInterface handles the reconfiguration requests.

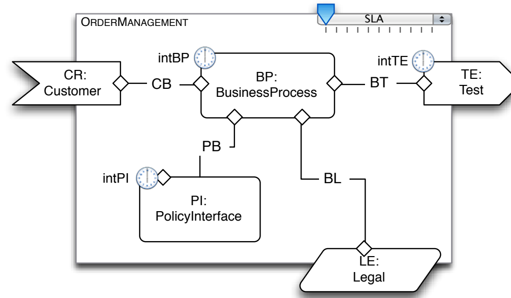


Fig. 9. VoIP Case Study SRML module.

The workflow implemented by BP is modelled with STPOWLA as follows. The STPOWLA base workflow is shown in Figure 10. In addition to the workflow specification, we attach a workflow ontology as follows:

```
STPOWLAWorkflow is
  Invariants:
    LowBusinessValueThreshold: int
    LowOrderValueThreshold: int
  Actors:
    Legal: Actor
    Test: Actor
    OrderManagement: Actor
    Customer: Actor
  Scopes:
    s1: [PerformServiceTest, ..., ResultsAcceptance]
    s2: [RequestLegalAssistance, ..., SendSLA]
  Variables:
    Customer.orderValue: int
    Customer.businessValue :int
```

What is important to note is that there are two workflow invariants `LowBusinessValueThreshold` and `LowOrderValueThreshold`, which refer to the level of spend from a customer for which to be regarded as a small customer and the upper value of an order which would be considered as small, respectively. Furthermore, on invocation of the workflow, two variables attached to the `Customer` actor specifying the customer's spend in the last year and the value of the current order.

The following policies should be activated on the workflow:

```
TelcoPolicy1
when workflow.started
  if   customer.businessValue < LowBusinessValueThreshold
     and
       customer.orderValue < LowOrderValueThreshold
  do delete(s1)

TelcoPolicy2
when workflow.started
  if Customer.orderValue < LowOrderValueThreshold
  do   delete(s3)
     andthen
       insert(PrepareFinalSLA,...,...)
     andthen
       insert(CreateAContract, ..., ...)
```

The first policy *TelcoPolicy1* states that on commencing the workflow, if the customer could be considered small

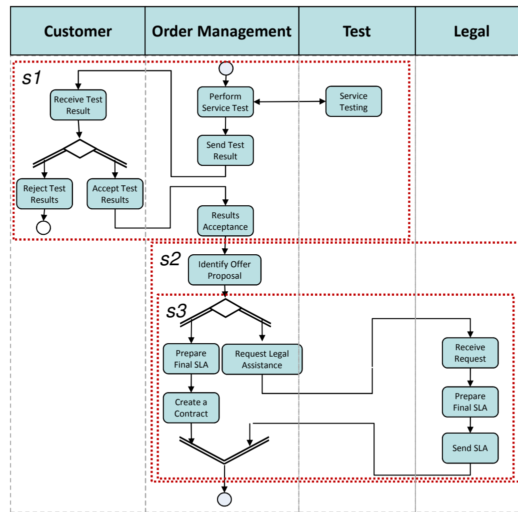


Fig. 10. VoIP Case Study base workflow.

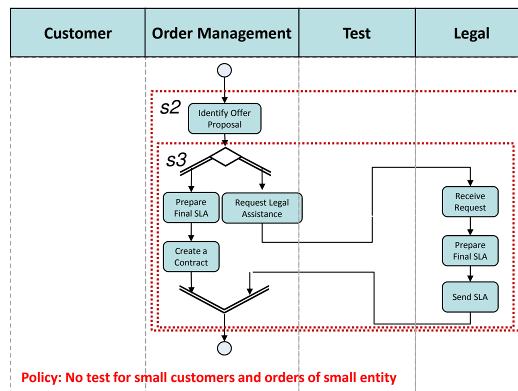


Fig. 11. VoIP Case Study base workflow.

and the order value was also small, the order management system should forgo the testing process. If this were applied, the reconfigured workflow instance would be as in Figure 11.

The second policy *TelcoPolicy2* states that on commencing the workflow, if the order value could be considered small, then the order management system should not invoke legal assistance. Noting the trigger position of the policy and the initial area of effect, the trigger could also have been `IdentifyOfferProposal.task_started` or `IdentifyOfferProposal.task_completed`. The choice of which trigger to use is down to the author and this

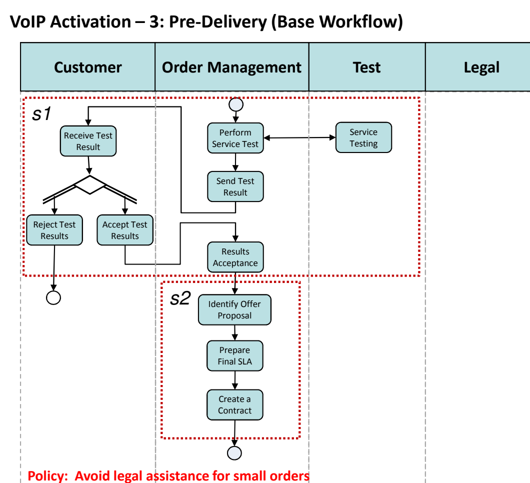


Fig. 12. VoIP Case Study base workflow.

can have a significant effect depending on the existence of other policies. The policy *TelcoPolicy2*, which eliminates the branch at the bottom of the split/join, is implemented by deleting the whole scope `s3` and reinserting the upper branch (this is required as just deleting the tasks along the lower branch will leave the operatoes in place and would provide a direct path bypassing the task that we do want to enforce).

For example, if the trigger of the second policy were bound to an event from a task in scope `s1`, it is possible that the scope would be deleted and the policy never initiated. Whereas instead the required functionality was to initiate the policy regardless.

If the customer could firstly be identified as a small customer and also the order value is below the low order threshold, it is clear that both policies would be applied. The resulting workflow is shown in Figure 13.

According to the encoding, the process triggered by the root transition (see Section 4) is represented in SRML by the transition x which executes in sequence the scope $s1$ and then the scope $s2$.

```
transition X
    triggeredBy start[x] ∨ done[s1] ∨ done[s2]
    guardedBy ¬ policy[x]
    effects start[x] ⊃ ¬ start[x]’ ∧ state[x]’=running ∧ start[s1]’
    ∧ done[s1] ⊃ ¬ done[s1]’ ∧ start[s2]’
    ∧ done[s2] ⊃ ¬ done[s2]’ ∧ done[x]’ ∧ state[x]’=exited
```

Transition X is executed when the sequence task is triggered and when any of the two sub-tasks of the sequence terminate. The guard ensures that no policies have been triggered for x. In the first execution, the state of x is set to running and the first task $s1$ in the sequence is triggered. In the second execution of the transition $s1$ completed successfully and the second task $s2$ in the sequence is triggered. In the third execution of the transition $s2$ completed successfully and the sequence task x terminates successfully.

Fragments of the transitions `s1` and `s2`, which illustrate triggers and guards, are reported below. The variables y and z represent the root process started within each scope.

```
transition scope1
    triggeredBy start[s1] ∨ done[y]
    guardedBy ¬ policy[s1]
     ...
transition scope2
```

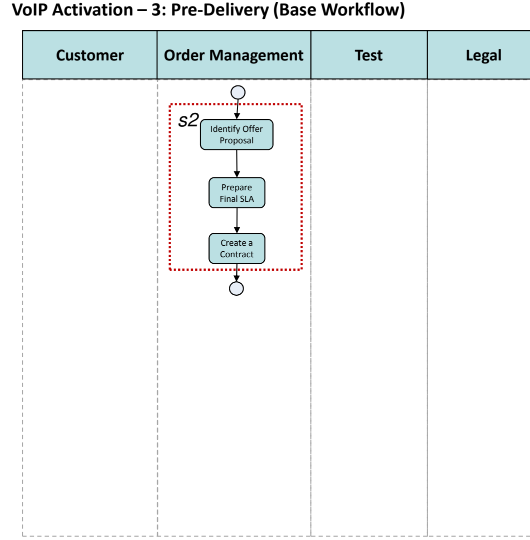


Fig. 13. VoIP Case Study base workflow.

```
    triggeredBy start[s2] ∨ done[z]
    guardedBy ¬ policy[s2]
     ...
```

The business role BusinessProcess provides transitions to handle the reconfiguration for each sub-process. When the customer creates a new instance of the service OrderManagment, the SLA variables concerning the business value of the customer and the order value are set. Depending on those values, PI will either request to apply $TelcoPolicy1$ and $TelcoPolicy2$, or not. In order to apply $TelcoPolicy1$, for instance, PI sends an event $delete[s1]$🔔! to BP. The transitions deleteScope1_1 and deleteScope1_2, defined as described in Section 5, handle the request of the component PI to delete the scope $s1$ (i.e., $TelcoPolicy1$).

```
transition deleteScope1_1
    triggeredBy delete[s1]🔔
    guardedBy state[delete[s1]🔔.task] = toStart
    effects policy[delete[s1]🔔.task]'

transition  deleteScope1_2
    triggeredBy start[s1]
    guardedBy P_delete[s1]🔔 ∧  delete[s1]🔔.task=s1
    effects ¬ start[s1]' ∧ done[s1]' ∧ state[s1]' = done
    sends delete[s1]✉
```

The transition deleteScope1_1 is triggered by the deletion request, only if scope $s1$ has not started its execution yet, and sets the variable $policy[delete[s1]$🔔$.task$ to true. When the transition x sets $start[s1] = true$, the trantision s1 can not be triggered because of the false guard. The transition deleteScope1_2 is triggered instead. The effects of deleteScope1_2 are to set the variables for $s1$ as if the scope had been successfully executed (but in fact it has not been executed at all), and to notify PI of the deletion through the interaction $delete[s1]$✉!. The following transition to be executed is, again, x which triggers the next process in the sequence (i.e., $s2$).

7. Related Work

SRML is inspired by the Service Component Architecture (SCA) [MHD+05]. SCA is a set of specifications, proposed by an industrial consortium, that describe a middleware-independent model for building over SOAs. Similarly to SCA, SRML provides primitives for modelling, in a technology agnostic way, business processes as assemblies of (1) tightly coupled components that may be implemented using different technologies (including wrapped-up legacy systems, BPEL, Java, etc.) and (2) loosely coupled, dynamically discovered services. Differently from SRML, SCA is not a modelling language but a framework for modelling the structure of a service-oriented software artifact and for its deployment. SCA abstracts from the business logic provided by components in the sense that it does not provide a means to model the behavioural aspects of services. SRML is, instead, a modelling language that relies on a mathematical framework and that provides the primitives to specify such behavioural aspects.

A (formal) model of services based on Component Based Development (CBD) can be found in [BKM07]. There, services are seen as “crosscutting elements of the system under consideration”, describing “partial views on the set of components in the system under consideration” [BKM07]. SRML captures a different notion of SOC in that there is no “system under consideration”, conceived a priori, that services crosscut. The configuration, consisting of a set of components and connectors which together provide a certain functionality, can dynamically change to include other components, procured according to given (functional) types and service level constraints from a universe that is not fixed a priori. The underlying middleware (SOA) supports the discovery and selection from a set of services that is itself dynamically changing as, for example, new services are published. SRML supports, in an integrated way, both CBD and SOC-based service provision in terms of its layered architecture that separates the service-oriented (horizontal) layer from the component-based (vertical) one. In this way, SRML captures the positive aspects of SOC (i.e., the service-oriented layer provides extensibility and adaptability to change) without unnecessary overhead when discovery is not needed (i.e., for resources and components).

The declarative primitives of SRML capture a comprehensive view of the foundational aspects of services at a high level of abstraction. SRML provides a more domain-level support for service-oriented modelling with respect to formalisms based on Petri-Nets (e.g., [Rei05]) or process calculi(e.g., [LPT07, GLG+06, BBC+06]). For example, SRML abstracts on the actual process of discovery, selection, binding, reconfiguration and session management, whereas COWS (Calculus for Orchestrating Web Services) [LPT07] addresses a lower level of abstraction in which the dynamic aspects need to be explicitly modelled. SRML has been related to cows in [BFL+08a] to provide an operational semantics of SRMLto make explicit in COWS some of the run-time aspects that the denotational semantics for SRML provided in [AF08, LFB07] abstracts from.

Process modelling at a business level is generally achieved using natural English or, more interestingly, structured languages such as the Business Process Modelling Notation (BPMN) [Whi04] or UML Activity Diagrams. BPMN describes process flows, with additional structure provided 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 [OEtH05]. However, these notations do not cater for all workflow patterns [vdAtHKB03], as described in [WvdAD+06] and [RvdAtHW06], respectively.

YAWL [vdAtH05] caters for all workflow patterns and has a graphical syntax with formal semantics, based on Petri Nets. At a lower business level, languages such as BPEL or WS-CDL are capable of expressing business processes, but with a code-based approach that is not high level enough for the end user.

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. [HB03] and [FBS02].

As we have previously mentioned, the syntax of the workflow language is not significant, but rather the places where a policy can interact. We have used the language of [GRM06c] for its simplicity, expressive power and our familiarity with it

Policies have generally been used as an administration technique in system management (e.g. access control [Mos05] or Internet Telephony [RM03]). 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 [SCZ03, HW03]; formalisms for modelling the more general notion for usage control [ZPPSP05]; formalisms for SLA, i.e. to specify client requirements and service guarantees, and to *sign* a contract with an agreement between them [BM07, BFMM07].

RuleML is a language for rule-based and knowledge-based systems, and allows Web-based rule storage, retrieval and interchange [BTW01]. 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 [(ed06] 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 compile the policies into this framework, which is an avenue worthwhile of future investigation.

In addition, a methodology has been proposed to extract workflows from business policies [Wan06]. However, we are not aware of policies being used as a variability factor in service-targeted processes.

Dynamic processes that are based on the end-user's needs are more difficult to find. Reichert and Dadam [RD98] and Adams et al [AtHEvdA06] introduce ideas for flexibility in workflows. 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, presents Worklets, a system based on an extensible repertoire of sub processes aligned to each task, one of which is chosen at runtime.

Possibly *AgentWork* [MGR04], 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 [GRM06b, GRM06a]. 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, service selection or the more significant workflow restructuring discussed here.

8. Summary and Conclusion

The engineering of Service Oriented applications, as opposed to more traditional application development, is faced with novel challenges arising from the dynamicity of component selection and assembly, leading to massively distributed, interoperable and evolvable systems. Furthermore, a continuing challenge is to correctly align business goals with IT strategy. As such, the development approach must change to accommodate these factors.

In this paper, we have presented SRML - a high level modelling language for service-based applications. SRML is based on a formal framework and can model service compositions and configurations. The orchestration of the services is modelled by a central agent in each SRML module. However, the business process is less clear. The orchestration according to the business process is defined by STPOWLA, an approach that combines workflows, policies and SOA.

The main contributions of this paper are 1) an encoding of basic STPOWLA workflow constructs in SRML, 2) an encoding of STPOWLA reconfiguration and refinement policies in SRML and (3) a methodology by example on a realistic case study showing how the combination of STPOWLA and SRML can be used.

The benefits of this work are twofold. The mapping between SRML and STPOWLA creates a formal framework for the latter (where it only previously had formal semantics for the APPEL policy language). Since applications are often designed based on the business process, STPOWLA can create process models easily, then transformed to SRML modules such that they can be analysed alone or as part of more complex modules. Even further, BPEL processes can be mapped to SRML modules for a clearer path between the defining the process and using the service. Looking at the encoding bottom-up, STPOWLA adds a higher level of modelling to SRML modules in the form of a process-oriented workflow schedule, with system variability separated from the core business concerns, which is considered for the first time here.

References

[ABFL07] João Abreu, Laura Bocchi, José Luiz Fiadeiro, and Antónia Lopes. Specifying and composing interaction protocols for service-oriented system modelling. In *Formal Methods for Networked and Distributed Systems*, volume 4574 of *Lecture Notes in Computer Science*, pages 358–373. Springer Verlag, 2007.

[AF08] João Abreu and José Luiz Fiadeiro. A coordination model for service-oriented interactions. In *In Proceedings of Coordination Languages and Models*, volume 5052 of *Lecture Notes in Computer Science*, pages 1–16. Springer Verlag, 2008.

[AtHEvdA06] 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.

[BBC+06] Michele Boreale, Roberto Bruni, Luís Caires, Rocco De Nicola, Ivan Lanese, Michele Loreti, Francisco Martins, Ugo Montanari, Antonio Ravara, Davide Sangiorgi, Vasco Vasconcelos, and Gianluigi Zavattaro. SCC: a service centered calculus. In *Web Services and Formal Methods*, volume 4184 of *Lecture Notes in Computer Science*, pages 38–57. Springer, 2006.

[BFL+08a] Laura Bocchi, José Luiz Fiadeiro, Alessandro Lapadula, Rosario Pugliese, and Francesco Tiezzi. From Architectural to Behavioural Specification: An encoding of SRML into cows. Technical report, Dipartimento di Sistemi e Informatica, Universit à di Firenze, 2008. Available at `http://rap.dsi.unifi.it/cows/` .

[BFL08b] Laura Bocchi, José Luiz Fiadeiro, and Antónia Lopes. Service-Oriented Modelling of Automotive Systems. *32nd Annual IEEE International Computer Software and Applications Conference (COMPSAC)*, 0:1059–1064, 2008.

[BFMM07] M.G. Buscemi, L. Ferrari, C. Moiso, and U. Montanari. Constraint-based policy negotiation and enforcement for telco services. 2007.

[BGR08] Laura Bocchi, Stephen Gorton, and Stephan Reiff-Marganiec. Engineering Service Oriented Applications: From StPowla Processes to SRML Models. In José Luiz Fiadeiro and Paola Inverardi, editors, *Fundamental Approaches to Software Engineering*, volume 4961 of *Lecture Notes in Computer Science*, pages 163–178. Springer Verlag, 2008.

[BHLF07] Laura Bocchi, Yi Hong, Antónia Lopes, and José Luiz Fiadeiro. From BPEL to SRML: A Formal Transformational Approach. In M. Dumas and R. Heckel, editors, *Web Services and Formal Methods*, volume 4937 of *Lecture Notes in Computer Science*, pages 92–107. Springer Verlag, 2007.

[BKM07] Manfred Broy, Ingolf H. Krüger, and Michael Meisinger. A formal model of services. *ACM Trans. Softw. Eng. Methodol.*, 16(1):5, 2007.

[BLF08] Laura Bocchi, Antónia Lopes, and José Luiz Fiadeiro. A use-case driven approach to formal service-oriented modelling. In *International Symposium on Leveraging Applications of Formal Methods, Verification and Validation*, Lecture Notes in Computer Science. Springer Verlag, 2008. to appear.

[BLZ03] Laura Bocchi, Cosimo Laneve, and Gianluigi Zavattaro. A calculus for long running transactions. In *FMOODS 2003*, Lecture Notes in Computer Science. Springer Verlag, 2003.

[BM07] M.G. Buscemi and U. Montanari. Cc-pi: A constraint-based language for specifying service level agreements. pages 18–32, 2007.

[BTW01] 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.

[DFS06] Schahram Dustdar, José Luiz Fiadeiro, and Amit P. Sheth, editors. *Business Process Management*, volume 4102 of *Lecture Notes in Computer Science*. Springer Verlag, 2006.

[(ed06] J. Schlimmer (ed). Web services policy 1.2 – framework (WS-Policy). W3C, Apr 2006. http://www.w3.org/Submission/WS-Policy/.

[FBS02] 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.

[FLB06] José Luiz Fiadeiro, Antónia Lopes, and Laura Bocchi. A Formal Approach to Service Component Architecture. *Web Services and Formal Methods*, 4184:193–213, 2006.

[GLG+06] Claudio Guidi, Roberto Lucchi, Roberto Gorrieri, Nadia Busi, and Gianluigi Zavattaro. SOCK: A calculus for service oriented computing. In Asit Dan and Winfried Lamersdorf, editors, *International Confetence on Service Oriented Computing*, volume 4294 of *Lecture Notes in Computer Science*, pages 327–338. Springer, 2006.

[GM05] Stefania Gnesi and Franco Mazzanti. A model checking verification environment for uml statecharts. In *AICA, Udine 2005*, 5-7 October 2005.

[GMRMS07] Stephen Gorton, Carlo Montangero, Stephan Reiff-Marganiec, and Laura Semini. StPowla: SOA, Policies and Workflows. In *Proceedings of 3rd International Workshop on Engineering Service-Oriented Applications: Analysis, Design, and Composition, Vienna, Austria (17th September 2007)*, 2007.

[GRM06a] 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.

[GRM06b] S. Gorton and S. Reiff-Marganiec. Towards a task-oriented, policy-driven business requirements specification for web services. In Dustdar et al. [DFS06], pages 465–470.

[GRM06c] Stephen Gorton and Stephan Reiff-Marganiec. Towards a task-oriented, policy-driven business requirements specification for web services. In Dustdar et al. [DFS06], pages 465–470.

[HA04] Hugo Haas and Allen Brown. Web Services Glossary. W3C Working Group Note, World Wide Web Consortium (W3C), 2004. `http://www.w3.org/TR/ws-gloss/`.

[HB03] 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.

[HW03] 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.

[LFB07] Antónia Lopes, José Luiz Fiadeiro, and Laura Bocchi. Algebraic Semantics of Service Component Modules. In *Algebraic Development Techniques*, volume 4409 of *Lecture Notes in Computer Science*, pages 37–55. Springer, 2007.

[LPT07] Alessandro Lapadula, Rosario Pugliese, and Francesco Tiezzi. A calculus for orchestration of web services. In *European Symposium of Programming*, volume 4421 of *Lecture Notes in Computer Science*, pages 33–47. Springer, 2007.

[MGR04] Robert Müller, Ulrike Greiner, and Erhard Rahm. Agentwork: a workflow system supporting rule-based workflow adaptation. *Data Knowl. Eng.*, 51(2):223–256, 2004.

[MHD+05] M. Beisiegel, H Blohm, D. Booz, J. Dubray, A. Colyer, M. Edwards, D. Ferguson, B. Flood, M. Greenberg, D. Kearns, J. Marino, J. Mischkinsky, M. Nally, G. Pavlik, M. Rowley, K. Tam, and C. Trieloff. Building Systems using a Service Oriented Architecture. Whitepaper, SCA Consortium, 2005. `http://www.oracle.com/technology/tech/webservices/standards/sca/pdf/SCA_White_Paper1_09.pdf`.

[Mos05] Tim Moses. extensible access control markup language specification. Available from www.oasis-open.org, 2005.

[MRMS07] Carlo Montangero, Stephan Reiff-Marganiec, and Laura Semini. Logic-based detection of conflicts in APPEL policies. In F. Arbab and M. Sirjani, editors, *FSEN2007*, volume 4676 of *Lecture Notes in Computer Science*. Springer Verlag, 2007.

[MRMSon] C. Montangero, S. Reiff-Marganiec, and L. Semini. Logic-based conflict detection for distributed policies. *Fundamentae Informatica*, 2008, accepted for publication.

[OEtH05] 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.

[RD98] M. Reichert and Peter Dadam. Adept$_{\text{flex}}$-supporting dynamic changes of workflows without losing control. *J. Intell. Inf. Syst.*, 10(2):93–129, 1998.

[Rei05] Wolfgang Reisig. Modeling- and analysis techniques for web services and business processes. In Martin Steffen and Gianluigi Zavattaro, editors, *Formal Methods for Open Object-Based Distributed Systems*, volume 3535 of *Lecture Notes in Computer Science*, pages 243–258. Springer, 2005.

[RM03] Stephan Reiff-Marganiec. Policies: Giving users control over calls. In Mark Dermot Ryan, John-Jules Ch. Meyer, and Hans-Dieter Ehrich, editors, *Objects, Agents, and Features*, volume 2975 of *Lecture Notes in Computer Science*, pages 189–208. Springer Verlag, 2003.

[RMTB05] Stephan Reiff-Marganiec, Kenneth J. Turner, and Lynne Blair. APPEL: the ACCENT project policy environment/language. Technical Report TR-161, University of Stirling, 2005.

[RvdAtHW06] Nick Russell, Wil M. P. van der Aalst, Arthur H. M. ter Hofstede, and Petia Wohed. On the suitability of uml 2.0 activity diagrams for business process modelling. In Markus Stumptner, Sven Hartmann, and Yasushi Kiyoki, editors, *APCCM*, volume 53 of *CRPIT*, pages 95–104. Australian Computer Society, 2006.

[SCZ03] 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.

[sen] Software engineering for service-oriented overlay computers (SENSORIA).
Web site: `http://sensoria.fast.de/`.

[SUF97] Stefano Bistarelli, Ugo Montanari, and Francesca Rossi. Semiring-based constraint satisfaction and optimization. *J. ACM*, 44(2):201–236, 1997.

[vdAtH05] Wil M. P. van der Aalst and Arthur H. M. ter Hofstede. Yawl: yet another workflow language. *Inf. Syst.*, 30(4):245–275, 2005.

[vdAtHKB03] Wil M. P. van der Aalst, Arthur H. M. ter Hofstede, Bartek Kiepuszewski, and Alistair P. Barros. Workflow patterns. *Distributed and Parallel Databases*, 14(1):5–51, 2003. Information also available from www.workflowpatterns.com.

[Wan06] Harry Jiannan Wang. *A Logic-based Methodology for Busines Process Analysis and Design: Linking Business Policies to Workflow Models*. PhD thesis, University of Arizona, 2006.

[WBC+07] Martin Wirsing, Laura Bocchi, Allan Clark, José Luiz Fiadeiro, Stephen Gilmore, Matthias Hölzl, Nora Koch, and Rosario Pugliese. *SENSORIA: Engineering for Service-Oriented Overlay Computers*. MIT, June 2007. submitted.

[Whi04] Stephen A. White. Business process modelling notation. Object Management Group (OMG) and Business Process Management Initiative, 2004. Available from www.bpmn.org.

[WvdAD+06] Petia Wohed, Wil M. P. van der Aalst, Marlon Dumas, Arthur H. M. ter Hofstede, and Nick Russell. On the suitability of bpmn for business process modelling. In Dustdar et al. [DFS06], pages 161–176.

[ZPPSP05] 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.