Engineering Service Oriented Applications: From StPowla Processes to SRML Models [*]

Laura Bocchi, Stephen Gorton, and Stephan Reiff-Marganiec

Department of Computer Science, University of Leicester
University Road, Leicester LE1 7RH, UK
{bocchi,smg24,srm13}@mcs.le.ac.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, 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. We also extend the current status of STPOWLA to include workflow reconfigurations. 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.

1 Introduction

Service Oriented Computing (SOC) is a paradigm for developing software systems as the composition of a number of services, that 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 [8]. 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 [20]. 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.

[*] This work has been partially sponsored by the project SENSORIA, IST-2005-016004.

We discuss the relationship between two modelling languages for service oriented systems developed in the context of SENSORIA, an IST-FET Integrated Project on Software Engineering for Service-Oriented Overlay Computers: the Sensoria Reference Modelling Language (SRML) [5,1] and STPOWLA: the Service-Targeted, Policy-Oriented WorkfLow Approach [6]. We also discuss the advantages of their combined usage.

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 formalized" [5]. SRML aims at representing, in a technology agnostic way, the various foundational aspects of SOC (e.g. service composition, dynamic reconfiguration, service level agreement, etc.) within one integrated formal framework.

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 [7]. 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.

So far, STPOWLA has been limited to non-functional changes to a workflow. In this paper, we extend the concept of workflow change to include reconfigurations: short lived structural changes to a workflow instance. We substantiate this extension by defining a further encoding of these advanced control flow aspects into SRML.

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 analyzed 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 [3]).

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.

In this paper, we give an overview of the STPOWLA approach, including the extension for workflow reconfigurations 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 STPOWLA reconfigurations as advanced control flow encodings in section 5. We describe related work and thus our position relative to these efforts in section 6, before discussing and concluding in sections 7 and 8.

2 Specifying and Reconfiguring StPowla Workflows

In this section, we give a brief introduction to the main concepts of STPOWLA. In addition, we present the concept of workflow reconfiguration, which is an extension to the current state of STPOWLA.

```
policyName
appliesTo task_id
     when task_entry
       do req(main, Inv, SLA)
```

Fig. 1. A StPowla task's default policy. 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 SLA constraints in the third argument.

2.1 Overview

StPowla has three ingredients: workflows, SOA and policies. Workflows specify core business processes, in which all task requirements are satisfied by services. Each workflow task has a default policy as in Fig. 1.

We describe a workflow, according to [7], with the following grammar to show how complex processes can be composed:

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

We describe the semantics of each construct with a description of the relevant SRML transition in section 4.

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 [14], a policy description language with formal semantics via a mapping to $\Delta DSTL$ [10]. They are written by the end (business) user and are added and removed at any time to the workflow. In addition to default policies, other policies can be added to the workflow to express system variability in terms of refinement and reconfiguration. The former type express constraints over runtime execution and service selection, but is out of the scope of this paper.

We have mentioned in an earlier paper [6] that the choice of workflow notation in StPowla is of small significance. What is of interest is the identification of a common set of triggers for ECA policies. We have identified the following as valid triggers:

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

Note that in StPowla, we view services as a black box, i.e. we cannot intervene in their processing between invocation and (possible) response.

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 1. Policy reconfiguration functions

2.2 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 1. 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:

```
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

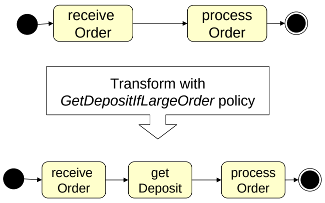


Fig. 2. A simple reconfiguration example where a core business process is transformed via the insertion of the *getDeposit* task after the *receiveOrder* task. The transformation rule comes from a policy.

module. A number of wires establish interaction protocols among the components and between the components and the external interfaces.

Components, external interfaces and wires are specified in terms of Business Roles, Business Protocols and Interaction Protocols, respectively. The specifications define the type of the nodes.

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. Fig. 3 illustrates the structure of the SRML module representing workflow and policies in the procurement example described earlier in this section.

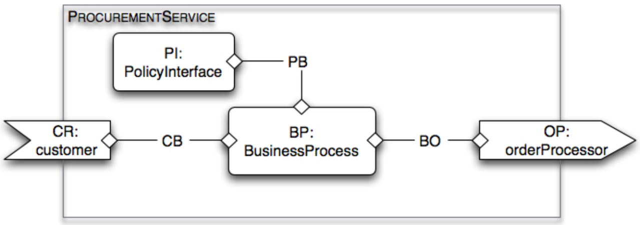


Fig. 3. The structure of a SRML module for the procurement service example

Components are instances of Business Roles specified in terms of (1) the set of supported interactions, (2) the way in which the interactions are orchestrated. We provide in the rest of this section an overview of Business Roles. The overview will not involve the other types of specification as they are not concerned in the encoding.

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 involved in the encoding.

It follows the specification of the interactions supported by $policyInterface$, corresponding each to one of the StPowla functions in Table 1. The Business Role $businessProcess$ supports the complementary interactions (i.e. **r&s** instead of **s&r**). Each interaction can have 🔔-parameters for transmitting data when the interaction is initiated and ✉-parameters for carrying a reply (the example below does not make use of the latter). 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.

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

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]
```

An initialisation condition may define a specific initial state such as:

```
initialization 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]!
```

1. A trigger is a condition: typically, the occurrence of a receive-event or a state condition. In the example we engage in the *policyHandlerExample* transition when we receive the initiation of the interaction *samplePolicy*.
2. A guard 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., $samplePolicy$🔔$[i].task$).

3. The effects concern changes to the local state. We use var' to denote the value the state variable var has after the transition.
4. The sends sentence describes the events that are sent and the values taken by their parameters. In the example we invoke the *samplePolicy* reply event to notify the correct management of the policy.

4 Basic Control Flow Encoding

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.

For every 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 constructs 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.

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). A local variable $state[x]$

identifies the state of the execution of x by taking the 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 state variable $state[x]$ will be used to ensure that policies act on a task in the correct state of execution.

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
```

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.

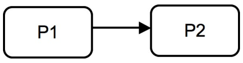


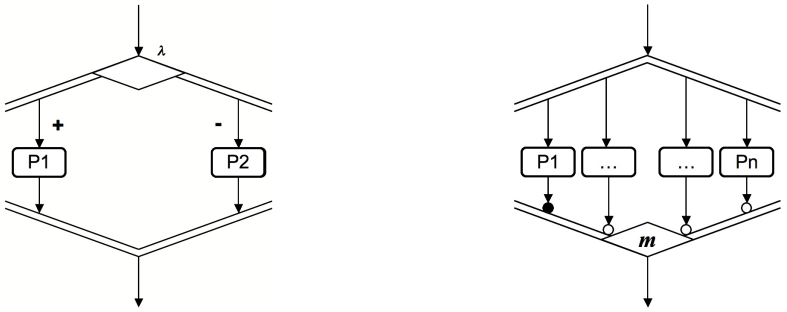
Fig. 4. The sequence control construct in StPowla

The encoding of the sequence construct in SRML is as follows. The sequence activity 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 $p2$. The sequence is encoded in the following SRML transition, with task identifier x:

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

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:



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

Fig. 5. Multiple branches constructs in STPOWLA

```
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
```

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).

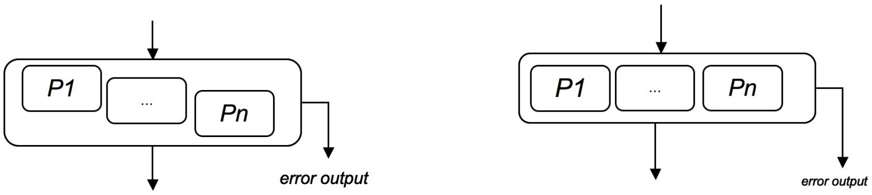
The encoding is as follows: Let S be the set, with cardinality n, of the task indexes associated to the branches of the split/join. Let the identifiers for the subtasks of x to range over $p1, \ldots, pn$. Let N be the set of indexes of the necessary tasks and $m \in \mathbb{N}$ be the minimum number 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] ∨ (∧_{i∈N}done[pi]∧(∨_{K∈Kcomb}(∧_{k∈K}done[pj])))
    guardedBy ¬ policy[x]
    effects start[x] ⊃ ¬ start[x]′ ∧ state[x]′=running ∧_{i∈[1,...,n]} start[pi]′
    ∧ ¬ start[x] ⊃ done[x]′ ∧ state[x]′=exited ∧_{i:[1..n]}(¬ done[pi]′)
```

The transition is executed: (1) when the task x is triggered or (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., $\vee_{K \in Kcomb}(\wedge_{k \in K} done[pj])$).

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:



(a) Strict preference (b) Random choice

Fig. 6. Other constructs in STPOWLA

```
transition X
    triggeredBy  start[x] $\vee_{i:[1..n]}$(done[pi] ∨ failed[pi])
    guardedBy ¬ policy[x]
    effects start[x] ⊃ ¬ start[x]′ ∧ state[x]′=running ∧ start[p1]′
    $\wedge_{i:[2..n-1]}$failed[pi] ⊃ ¬ failed[pi]′ ∧ start[p(i+1)]′
    ∧ failed[pn] ⊃ ¬ failed[pn]′ ∧ failed[x]′ ∧ state[x]′=exited
    ∧ $\vee_{i:[1..n]}$done[pi] ⊃ done[x]′ ∧  state[x]′=exited $\wedge_{i:[1..n]}$done[pi]′
```

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] $\vee_{i:[1..n]}$(done[pi])∨ ($\wedge_{i:[1..n]}$(failed[pi]))
    guardedBy ¬ policy[x]
    effects start[x] ⊃ ¬ start[x]′ ∧ state[x]=running $\wedge_{i:[1..n]}$start[pi]′
    ∧ ($\wedge_{i:[1..n]}$failed[pi])⊃ failed[x]′∧ state[x]′=exited $\wedge_{i:[1..n]}$¬ failed[pi]′
    ∧ ($\vee_{i:[1..n]}$done[pi]) ⊃ done[x]′ ∧ state[x]′=exited
     $\wedge_{i:[1..n]}$(¬ done[pi]′ ∧ ¬ failed[pi]′)
```

5 Advanced Control Flow Encoding

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 such transitions in detail when discussing individual interactions, in the rest of this section.

A policy related to a task can have 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 along 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 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 $done$.

Delete Task The deletion of task (i.e. $\mathtt{delete(x)}$ in STPOWLA) skips the execution of x. The policy manager prevents the execution of x by signaling a policy exception (i.e. $policy[x] = true$). When the signal for the execution of x is received, the policy handler signals the successful termination of x. The condition $P\_delete[i]$🔔? is $true$ when the event $delete[i]$🔔? occurred in the past.

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

Block Task The function $\mathtt{block(x,p)}$ in STPOWLA blocks a task until p is $true$. In SRML the policy handler prevents x from executing (i.e. *policy[x]* becomes $true$) temporarily 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_2
    triggeredBy block[i]🔔?.condition
    guardedBy P_block[i]🔔?
    effects ¬ policy[block[i]🔔.task]'
    sends block[i]✉!
```

Insert The insertion of a task, represented by the function $\mathtt{insert(x,y,z)}$ in STPOWLA, inserts the task y in sequence or in parallel with respect to x depending on 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.

The transition $policyHandler\_insert\_1$ prevents the execution of the task on which the policy applies (i.e., $insert[i]$✉.$task$) and of the successive one. 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_1
    triggeredBy insert[i]🔔?
    guardedBy  state[insert[i]🔔.task]=toStart
    effects policy[insert[i]🔔.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]'

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)]'

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)]'
```

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]✉!
```

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]✉!
```

5.1 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
```

In case of a receive event of type $insert[i]$✉?, 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 Related Work

SRML is inspired by the Service Component Architecture (SCA) [9]. 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.

Process modelling at a business level is generally achieved using graphical languages such as BPMN [19] or UML Activity Diagrams. However, they do not cater for all workflow patterns [17], as described in [21] and [15], respectively. YAWL [16] caters for all workflow patterns and has a graphical syntax with formal semantics, based on petri nets, so it is a good candidate for the process notation. As we have previously mentioned, the syntax is not significant, but rather the places where a policy can interact is. We have used the language of [7] for its simplicity, expressive power and our familiarity with it. 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.

Policies have generally been used as an administration technique in system management (e.g. access control [11] or Internet Telephony [13]). In addition, a methodology has been proposed to extract workflows from business policies [18]. 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. The closest we know of are AgentWork [12], which is dynamic only because of the choice of rules to follow under failure, and Worklets [2], the YAWL module for dynamic processes. Worklets though are based on a set of processing rules that are predefined, and of which one is selected.

7 Discussion

The benefits of this work are twofold. The mapping between SRML and StPowla creates a formal framework for the latter (which only benefitted from a formal semantics for the Appel policy language). Since applications are often designed based on the business process, StPowla is the ideal vehicle for this step. Transformation to SRML modules allows for analysis of the modules, either on their own or as part of more complex modules. 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.

To exemplify the benefits, we describe an application scenario from an industrial case study, provided by a partner in the Sensoria project. The scenario describes the interaction between a VoIP telephony provider and their internal service network, based on a pre-delivery state.

The trigger of the workflow is the supplier receiving an order from a customer. Under normal operating circumstances, this proceeds to a testing phase, where the customer can choose to accept or decline the test results. If they accept the results, then the supplier will proceed to identifying an offer proposal. In order to do this, legal assistance is required from a legal service. This assistance is then received and embedded into a Service Level Agreement between the customer and the supplier. The contract is created and the workflow is then complete.

There are two policies that affect how the workflow is executed. Firstly, if the order is sufficiently small and the customer's order is of sufficiently small value, then the initial testing phase should be bypassed. Secondly, if the order is of sufficiently small value, then no legal assistance should be sought. If both these policies were applied to the workflow, then it would essentially be represented as *ReceiveOrder* → *IdentifyOffer* → *CreateContract*.

All tasks inside the workflow are handled by services, either internal to the supplier or external (e.g. in the case of legal assistance and getting customer acceptance or refusal). The service composition and configuration can then be modelled formally in SRML. The effect of the policies can also be incorporated and through the application to the SRML model, the effects can be analysed and reasoned about.

Any method suggested for software engineering must be considered with respect to its scalability. The term can be interpreted in two different ways here, namely does the encoding mechanism itself scale and also do the StPowla and SRML methods scale. To answer the former, it can be said that it scales as every opeartor of StPowla is mapped into a relatively straight forward transaction in SRML. The only exception is the random choice operator, where the matching transaction is slightly more complex. As for the latter aspect, and in our opinion this is more crucial for practical purposes, we actually gain scalability. This is achieved by using the high-level StPowla notation to capture the business process in a more abstract and more easily understandable way than one would achieve by using SRML directly – however, reasoning for e.g. validation can make use of the formality of SRML in addition of course to SRML being a step towards implementation.

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 a mapping from StPowla to SRML. SRML is a high level modelling language for service-based applications, based on a formal framework. SRML 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 aspect is less clear. StPowla is a SOA aware approach that combines workflows and policies. It allows to define the orchestration according to a business process.

The main contributions of this report are 1) to encode basic StPowla workflow constructs in SRML, 2) to extend StPowla with workflow reconfiguration functions, and 3) to encode these reconfiguration functions in SRML.

Future work includes the application of this work to some larger case studies and consider mapping StPowla refinement policies into SRML.

References

1. 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 (FORTE)*, LNCS. Springer, 2007. To appear.
2. 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.
3. Laura Bocchi, Yi Hong, Antónia Lopes, and José Luiz Fiadeiro. From BPEL to SRML: A Formal Transformational Approach. In *Proc.of 4th International Workshop on Web Services and Formal Methods (WSFM'07)*, LNCS. Springer, 2007.

4. Schahram Dustdar, José Luiz Fiadeiro, and Amit P. Sheth, editors. *Business Process Management, 4th International Conference, BPM 2006, Vienna, Austria, September 5-7, 2006, Proceedings*, volume 4102 of *Lecture Notes in Computer Science*. Springer, 2006.
5. José Luiz Fiadeiro, Antónia Lopes, and Laura Bocchi. A Formal Approach to Service Component Architecture. *Web Services and Formal Methods*, 4184:193–213, 2006.
6. 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.
7. Stephen Gorton and Stephan Reiff-Marganiec. Towards a task-oriented, policy-driven business requirements specification for web services. In Dustdar et al. [4], pages 465–470.
8. 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/`.
9. 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`.
10. 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, 2007.
11. Tim Moses. extensible access control markup language specification. Available from www.oasis-open.org, 2005.
12. Robert Müller, Ulrike Greiner, and Erhard Rahm. Agentwork: a workflow system supporting rule-based workflow adaptation. *Data Knowl. Eng.*, 51(2):223–256, 2004.
13. 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, 2003.
14. Stephan Reiff-Marganiec, Kenneth J. Turner, and Lynne Blair. APPEL: the ACCENT project policy environment/language. Technical Report TR-161, University of Stirling, 2005.
15. 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.
16. Wil M. P. van der Aalst and Arthur H. M. ter Hofstede. Yawl: yet another workflow language. *Inf. Syst.*, 30(4):245–275, 2005.
17. 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.
18. 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.
19. Stephen A. White. Business process modelling notation. Object Management Group (OMG) and Business Process Management Initiative, 2004. Available from www.bpmn.org.
20. 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.
21. 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. [4], pages 161–176.