Available online at www.sciencedirect.com

SCIENCE DIRECT®

Electronic Notes in Theoretical Computer Science 154 (2006) 139–158

Electronic Notes in Theoretical Computer Science

www.elsevier.com/locate/entcs

Designing a BPEL Orchestration Engine Based on ReSpecT Tuple Centres

Michele Cabano, Enrico Denti, Alessandro Ricci, Mirko Viroli

DEIS, Alma Mater Studiorum, Università di Bologna,
via Venezia 52, I-47023 Cesena, Italy
{mcabano,edenti}@deis.unibo.it
{a.ricci,mirko.viroli}@unibo.it

Abstract

We present the design of a BPEL orchestration engine based on ReSpecT tuple centres, a coordination model extending LINDA with the ability of declaratively programming the reactive behaviour of tuple spaces. Architectural and linguistic aspects of our solution are discussed, focussing on how the syntax and semantics of BPEL have been mapped to tuple centres. This is achieved by a translation of BPEL specifications to set of logic tuples, and conceiving the execution cycle of the orchestration engine in terms of ReSpecT reactions.

Keywords: Web Services, Orchestration, Worflow, Multiagent Systems, Tuple Centres

1 Introduction

Studying Web service orchestration theory and practice is currently a hot research topic: while accessing a single Web service via the appropriate standard protocols is quite straightforward, suitably coordinating multiple Web services so as to build a composite, workflow-like service with a known semantics is a fairly more complex issue [12]. In this context, the BPEL language (Business Process Execution Language) [14] is deserving increasing attention, as it is becoming the *de facto* standard for the specification of complex activities, such as Web-based business processes. Several attempts are currently made to provide a grounded semantics to language features related to orchestration,

doi:10.1016/j.entcs.2005.12.037

including compensation [3,2], correlation [15], and fault handling [8] — mostly developed in the framework of process algebras [1].

On the other hand, the research field of coordination models and languages have typically studied similar issues, even though in the more general context of parallel and distributed systems, and yielded interesting technologies and solutions applicable to Web scenarios as well. The ReSpecT *tuple centres* coordination model is an example of such solution [10], extending the LINDA model [6] with the idea of programmable tuple spaces, which can be used as true *coordination virtual machines* — full-expressive virtual machines for supporting coordination-oriented tasks. As a special case, tuple centres can be exploited for building *workflow engines*, realising workflow management tasks [13]. Thus, it appears natural to evaluate the applicability of this framework for implementing activities specified as BPEL processes, with the ultimate purpose of defining a complete *BPEL orchestration engine*.

So, instead of dealing with semantic aspects as in most current research on orchestration languages, in this work we mean to tackle "abstract" implementation issues. We report on a ongoing project developed in our department to design and implement a full featured BPEL orchestration engine based on a multi-agent system coordinated through ReSpecT tuple centres. As agents realise very simple, basic activities of the engine, the key role of supporting the BPEL orchestration semantics is played by ReSpecT tuple centres, which are declaratively programmed using the ReSpecT logic-based language [9]. This project allows us to prove the effectiveness of this coordination framework, which showed an intrinsic ability to scale up with the complexity of BPEL language. In the end, our current prototype design realises the complete workflow behaviour of BPEL in terms of a ReSpecT program of less than 200 rules (around 50 Kbytes of code) — significantly smaller than usual monolythical implementations of orchestration engines. This solution allows for rapid prototyping and ease maintenance; further advantages expected from our implementation are the ability to dynamically track interactions, debug the execution of the orchestration activity, and dynamically adapt the orchestration behaviour to tackle unpredictable events such as overloads.

In particular, in this paper our purpose is to stress the issue of designing a BPEL orchestration engine based on ReSpecT tuple centres, both from the architectural and the linguistic viewpoint. After shortly introducing BPEL basics (Section 2), and ReSpecT tuple centres (Section 3), we first present a tuple centre-based architecture for the BPEL engine: there, we discuss the BPEL aspects that are mapped onto architectural items (Section 4). Then we analyse how BPEL language syntax and semantics are mapped onto suitable tuples and ReSpecT rules (Section 5). Conclusions and related works are

reported in Section 6.

2 BPEL Orchestration

2.1 BPEL as a Language

BPEL is an XML-based specification language for describing business processes orchestrating the interaction of different, existing and possibly dynamically emerging Web services. As such, it builds on top of the WSDL language for describing the interface of Web services [4] — in terms of ports, actions, and message types. A BPEL specification is made of four declaration parts: the *partner links*, the *variables*, the *correlation sets*, and the *activity* realising the business process.

Partner links identify the relationship of the business process with the other Web services it interacts to, by specifying the port types for both process/Web-service and Web-service/process interactions. Variables can be defined that carry XML data values and messages, thus defining the state of each process instance — a working session responsible for orchestrating a given client request. Correlation sets — basically group of fields in messages [15] — are then introduced to identify those interactions that are pertinent to a given process instance, which is necessary in order to correctly dispatch messages between the various concurrent sessions. Finally, an activity is specified that describes the precise behaviour of the business process — the part of a BPEL specification we mostly focus on. Activities are generally built by composing basic ones through structured ones. Basic activities include the acts of sending and receiving requests and replies (`invoke`, `receive`, and `reply`), which can specify one or more existing correlation sets they must adhere to, or new correlation sets to be initialised. Among other basic activities, there are variable assignment (`assign`), waiting for a timeout (`wait`), and raising faults (`throw`). Structured activities realise sequential composition (`sequence`), guarded choice (`pick`), parallel composition (`flow`), iteration cycles (`while`), and multiple cases (`switch`). A (private) link mechanism is introduced to let two activities in different branches of a `flow` construct to synchronise one another, by executing the basic activities `source` and `target`, respectively. To scale up with the complexity of specifications, a basic activity can also take the form of a *scope*, that is, a separately-defined subprocess with its own activity, variables and correlation sets.

A BPEL specification can actually provide the definition of a number of fault handlers, which are subprocesses similar to scopes: they can be executed by an explicit `throw` or when a synchronous interaction fails, and can be used to recover to the fault or simply to terminate the whole process instance.

2.2 A Case Study

As a reference case study, in this paper we consider the shipping service described in the official specification of BPEL [14] (Section 16.1). Since this example actually describes an abstract process, that is, the protocol (or choreography) of interactions between customers and the business process — where interactions with shipping services is abstracted away —, we specialise it to an actual executable process orchestrating customers and shipping servers. In spite of its simplicity, the example we show covers most of the language features we are interested in, including basic activities, structured activities and fault-handling.

We describe a Web service handling the shipment of orders, made by customers to shipping services — both of them modelled as Web services. Customers specify orders containing more items, and shipping services could ship only a subset of them at a time: the business process is then in charge of invoking the shipping service more times until all items have been shipped, sending a ship notice callback to the customer each time. If a single shipment cannot be completed, we suppose the whole shipment is aborted and a notification is sent to the customer. The activity specifying the business process is of the kind shown in Figure 1. There, underlined parts do not represent actual XML code, but are rather placeholders informally describing a more complex XML code, whose details are not reported for the sake of brevity.

The workflow behaviour realised is as follows. The request message `shipRequest` is initially received from the `customer`. While its `itemsTotal` part is greater than the `itemsShipped` variable (initialised to zero), message `shipRequest` is sent to the `storeService`. If the `shipResponse` reply is completed (`shipComplete` part), a `shipNotice` message is prepared and sent to the `customer`, and variable `itemsShipped` is updated. Otherwise, the fault named `shippingError` is thrown specifying variable `error`, which causes a message to be sent to the `customer` before terminating the process instance (`terminate`).

3 ReSpecT Tuple Centres for Orchestration

The Tuple Centre Coordination Model

The tuple centre coordination model is based on the notion of *programmable coordination media* [10] — sort of general-purpose virtual machines executing coordination specifications encoded in some specification language.

More specifically, tuple centres are *programmable tuple spaces*. Similarly to tuple spaces — as found e.g. in the LINDA model [6] — they accept and serve

```
<sequence>
  <receive customer:  shipRequest> ...  </receive>
  <assign> itemsShipped:=0 </assign>
  <while condition = itemsShipped < shipRequest.itemsTotal>
    <sequence>
      <invoke storeService:  shipRequest,shipResponse> ...  </invoke>
      <switch>
        <case shipRequest.shipComplete>
          <sequence>
            <assign> shipNotice.itemsCount:=storeService.itemSent </assign>
            <invoke customer:  shipNotice> ...  </invoke>
            <assign> itemsShipped+=shipNotice.itemsCount </assign>
          </sequence>
        </case>
        <otherwise>
          <throw shippingError.error>
        </otherwise>
      </switch>
    </sequence>
  </while>
</sequence>

<faultHandlers>
  <catch shippingError.error>
    <sequence>
      <invoke customer:  itemsShipped> ...  </invoke>
      <terminate>
    </sequence>
  </catch>
</faultHandlers>
```

Fig. 1. BPEL specification schema for the shipping service

requests from external agents for inserting a tuple (`out` primitive), removing a tuple matching a template (`in` primitive), and reading a tuple matching a template (`rd` primitive). Differently from basic tuple spaces, tuple centres can be programmed so that whenever an external communication event occurs, a computation reactively starts which may affect the state of the inner tuple space. This feature makes it possible to fully *encapsulate* coordination policies inside a tuple centre, in terms of behaviour and dynamic state of the coordination activities, and to provide an high degree of separation between computation (embedded in agents) and coordination (embedded in tuple centres). In the overall, the tuple centre model enables system designers to effectively balance the coordination burden in a system, charging coordination media with it by suitably programming their behaviour according to the required needs. For a more detailed discussion, the interested readers can refer to [10].

```
                Spec  ::=  { Reaction }
            Reaction  ::=  reaction( Event, ( Body ) ).
               Event  ::=  CommunicationEvent | InternalEvent
   CommunicationEvent ::=  out(T) | in(T) | rd(T)
       InternalEvent  ::=  out_r(T) | in_r(T) | rd_r(T)
                Body  ::=  { Goal{, Body} }
                Goal  ::=  out_r(T) | in_r(T) | rd_r(T) | no_r(T)
```

Fig. 2. Syntax of the ReSpecT language core

ReSpecT *Tuple Centres*

The particular type of tuple centre we adopt in this paper is referred to as ReSpecT tuple centre [10]. It features a communication language based on primitives `out`, `rd` and `in` working on first-order logic tuples, and using logic unification as tuple-matching criterion. As specification language it uses ReSpecT (Reaction Specification Tuples [9]), by which the tuple centre reactive behaviour is programmed through a set of *specification tuples*: first-order (Prolog-like) facts specifying a *reaction* to certain interaction events. In this framework, while the tuple set content of a tuple centre can be framed as a logic theory on the interactions (communication) occurred among the agents using the tuple centre, a ReSpecT specification can be framed as a theory about the management of such interactions — by definition, as the theory of the *system coordination*.

Figure 2 reports the syntax of the core of the ReSpecT language — in this paper, for simplicity we described only the subset of language features we directly exploit. A ReSpecT program is a set of reactions, each with a head and a body. When a communication event occurs, all the reactions with a matching head are activated, that is, their bodies — each specifying an atomic computation over the tuple centre — are used to spawn a pending reaction waiting to be executed. Such reactions are composed by a sequence of primitives, used to remove a tuple (`in_r`), read a tuple (`rd_r`) and insert a tuple (`out_r`). Pending reactions are non-deterministically picked and executed, by atomically executing all their reaction primitives — if a single primitive fails, e.g. a tuple to be read is absent, the whole effect of the reaction is rolled back. The execution of each such primitive fires a new communicated event, which can be again intercepted by another reaction, and so on, recursively. Therefore, the computation starts with an external communication event (the reception of an `out`, `rd`, or `in`) and can possibly recursively proceed due to internal communication events (execution of `out_r`, `rd_r`, or `in_r`), until reaching a fixpoint where no more pending reactions exist. At that stage, the ReSpecT tuple centre reached a new state of its tuple set, and simply waits for another external communication event to occur. This recursive firing of reactions is the mechanism by

which ReSpecT achieves expressiveness, reaching Turing-completeness [5].

The ReSpecT coordination model is at the core of TuCSoN coordination infrastructure for multi-agent systems [11]: in TuCSoN, tuple centres are distributed over the network, collected and managed by infrastructure nodes – which corresponds to Internet nodes. The infrastructure enables agents to access and use tuple centres either in a local network-unaware fashion, when agents reside on the same node hosting the tuple centres to be exploited, or in a global network-aware fashion, when they reside on any other Internet node.

ReSpecT *Tuple Centres as Orchestration Engines*

The features of ReSpecT tuple centres make them a natural choice for the design and development of those parts of a system tailored to coordination aspects. Workflow and orchestration engines are exemplar cases, being complex components centralising the responsibility of the coordination of distributed activities (services) [13]. Therefore, tuple centre programmability and ReSpecT expressive power can be naturally exploited to design an orchestration engine upon one or multiple tuple centres, programmed to dynamically interpret and execute a BPEL specification suitably encoded in forms of logic tuples.

The properties of this coordination model can be exploited to realise orchestration engines with advanced functionalities, in particular concerning the dynamic management of coordination activities, in terms of observation, control and adaptation. For instance, some specific ReSpecT rules could be added to trace any interaction of interest inside the engine, in order to monitor the communication among the orchestrated parties and the state of orchestration activities. As another example, the model supports the dynamic adaptation of the orchestration activities through changes on the set of tuples – required, for instance, to face unpredictable events happening in the environment.

4 Orchestration Engine Architecture

Since in our approach we are concerned with coordination, and see it as being mostly orthogonal to communication (partner links) and data representation (variables and properties) aspects, we shall focus our discussion on the activities specified in BPEL. Whereas a BPEL process is externally seen as an orchestrator of Web services, when looking at the internal details of an engine for BPEL specifications another similar view is possible and fruitful. At that level, a BPEL process instance is seen as coordinating a set of basic activities — such as `invoke`, `receive`, `assign`, `terminate`, and the like —, to be performed in the order prescribed by the BPEL semantics [14]. In the context of

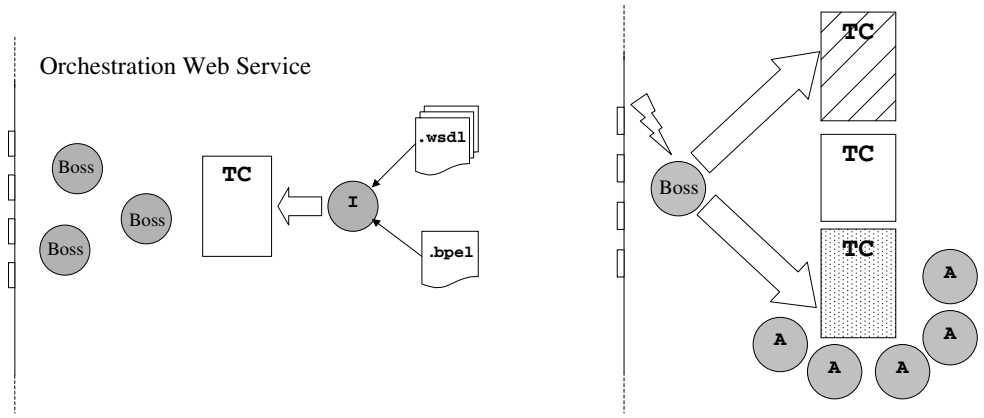

Fig. 3. *Left:* setup phase by the ***interpreter*** agent. *Right:* creation of agents and tuple centres (***bpel_variables***, ***bpel_engine***, ..) for the process instance.

ReSpecT tuple centres, this view naturally promotes the idea of realising the single basic activities through agents, and the coordination of such activities via tuple centres, which require agents to execute activities "on-demand". This multi-agent system will therefore be perceived as the (conceptually atomic) orchestration server realised by the engine.

The life-cycle of an orchestration service in our engine is then as follows. At bootstrap, an ***interpreter*** agent in the engine takes the description of both the BPEL process (a `.bpel` file) and the single Web services to be orchestrated (the `.wsdl` files), and produces a representation in terms of ReSpecT logic tuples, stored in tuple centre ***bpel_spec***. When this set-up phase is over, the system configuration completes by creating a pool of ***boss*** agents, in charge of actually launching the BPEL process instances (Figure 3, *left*). This happens only in response to a request from a client of the Web server: in BPEL a new process instance is actually created only due to a `receive` or a `pick` (involving a set of possible message receptions), for these are the only activities which can expose the `createInstance` attribute [14,15]. When a ***boss*** agent receives one such request, it retrieves from the ***bpel_spec*** tuple centre the information required to set up a running instance of the BPEL process, and creates the agents and the tuple centres needed to execute it (Figure 3, *right*). As the process instance terminates the ***boss*** agent releases all the used resources.

Whether for a given instance there should be one or more tuple centres and one or more agents, this is a design choice which might impact the solution performance and simplicity, but is mostly orthogonal to the general engine architecture; in this stage of our development process we are mostly concerned with simplicity, and hence will neglect performance issues. On the one hand,

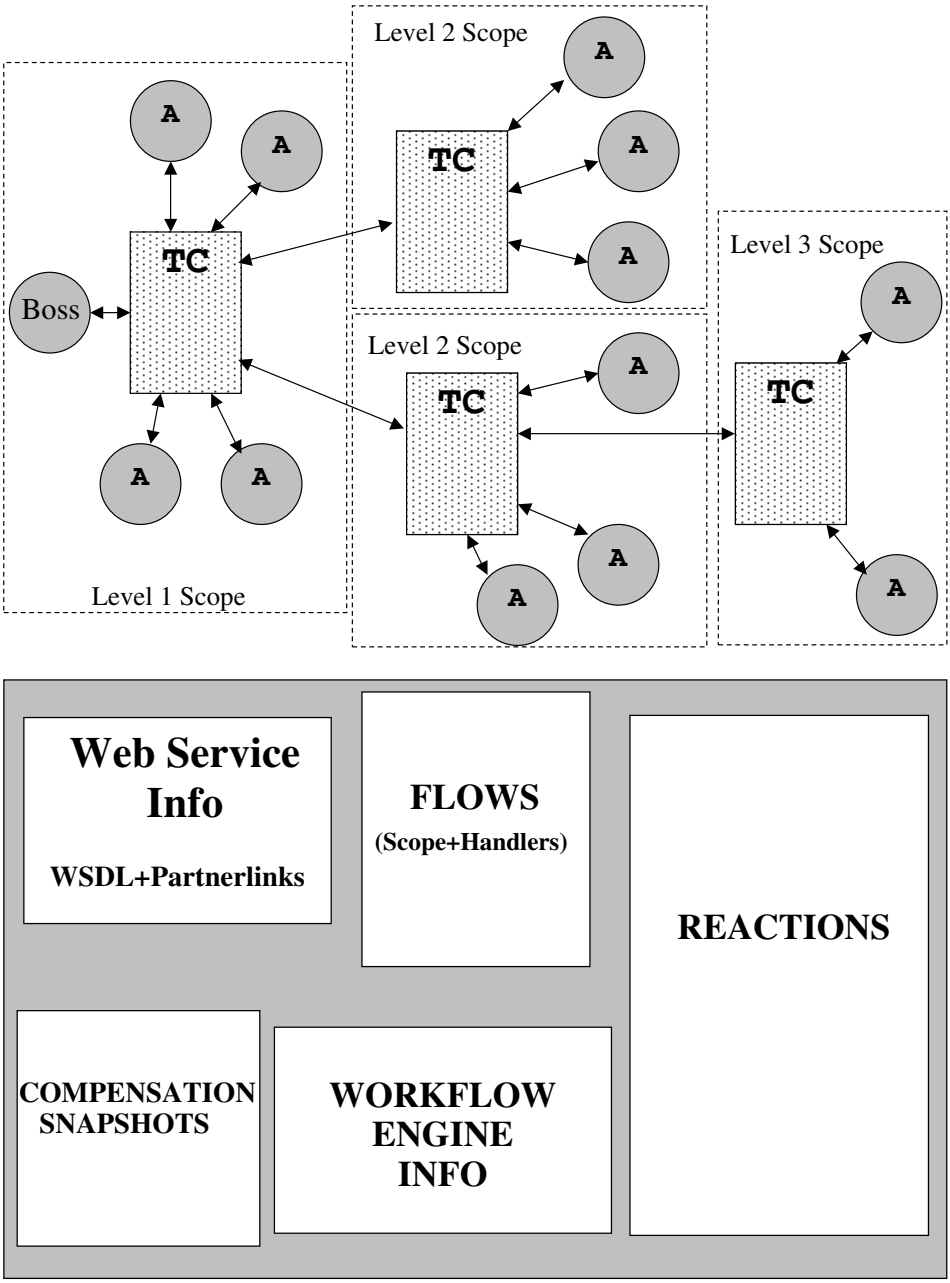

Fig. 4. *Top:* the run-time architecture with multiple `bpel_engine` tuple centres. *Bottom:* the different kinds of tuple categories inside each tuple centre.

given one process instance, we associate precisely one agent to each basic activity tag — so we will have one agent handling all the `invoke` activities, one the `receive`'s, one the `assign`'s, and so on. As shown in the next section, this guarantees a very simple interaction protocol between agents and tuple centres, while promoting separation of concerns in agents' implementation. On the other hand, a process instance is associated with many tuple centres (Figure 3, *right*): one ***bpel_variables***, holding the value of the process instance's variables, and one ***bpel_engine*** for each different "flow" expressed in the BPEL specification — one for the main activity process (called *normal flow*), one for each `scope` (Figure 4, *top*), and one for each compensation, fault, and event handler defined. During the normal flow execution, the other tuple centres can be activated, either to read/update variables and properties, when entering scopes, or when handlers are to be executed.

Accordingly, tuples in each tuple centre conceptually refer to different information categories: the main one describes scope flows, the other described supporting information, such as the data format for message exchange, the addresses and port numbers of nodes providing the Web services, and all the compensation handler snapshots (Figure 4, *bottom*).

The decision of making a single tuple centre responsible for just one flow dramatically simplifies our design: it allows us to develop tuple centre according to the "workflow virtual machines" (WVM) pattern — as discussed in details in [13]. In particular, the actual workflow to be executed is not hardwired into the tuple centre program; rather, it can be specified via a script of a suitable "workflow specification language" — in our case, a set of logic tuples represents the BPEL specification of the flow.

5 Mapping BPEL onto ReSpecT

In this section we describe the mapping of a BPEL orchestration server in ReSpecT tuple centres, both in terms of the representation of a BPEL specification and the WVM behaviour expressed in terms of a ReSpecT program. We focus the discussion on the behaviour of those tuple centres realising the execution of normal flows: for the mere sake of space we shall only briefly sketch aspects related to the management of variables, scopes, and handlers of compensation, events and faults.

5.1 A BPEL Specification as a Net

Whereas a BPEL specification can be easily understood as a process-algebraic specification [15], as far as execution/interpretation by ReSpecT tuple centres is concerned it is more fruitfully described in terms of a net — or automaton

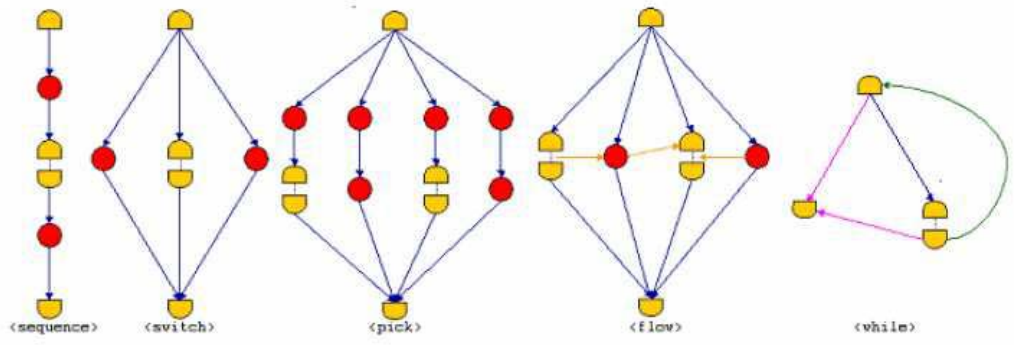

Fig. 5. Structured activities as nets

[7].

Its static structure is described by a directed graph with two kinds of nodes — some examples of graphs are shown in Figure 5. Nodes are called *tasks*, and can be *(i)* basic BPEL activities, drawn as circles, or *(ii) tags* identifying begin and end of complex control structures, drawn as half circles. Edges are called *links*, and express dependencies between the execution order of tasks. Figure 5 shows how the basic structured activities of BPEL are mapped to graphs — subgraphs of a whole specification in between a start and stop tag.

The `sequence` construct expresses a sequential ordering to the activities it specifies, hence links connect such activities through a chain. E.g. in the Figure 5 a sequence of one basic activity, one structured activity, and again one basic activity is shown. The `switch`, `pick`, and `flow` constructs express instead activities without direct ordering, hence links simply connect the start tag with each activity, and each activity with the end tag. In the Figure 5, the graph corresponding to the `pick` construct shows that if a single activity in it is a sequence, we do not need further start and end tags for it; the graph corresponding to the `flow` construct shows that links can in this case even exist between the activities, as expressed by the `link` construct of BPEL specification. The `while` construct expresses a conditioned iteration cycle, hence the single activity can be executed many times due to the loop.

Concerning links, we find it useful to distinguish four kinds of semantically-different ones: *(i) order links*, imposing a sequential ordering, *(ii) flowsynch links*, modelling BPEL links between activities in a `flow`, *(iii) loopback cycle links* (or `cycle1`), dealing with `while` construct loops, and *(iv) escape cycle links* (or `cycle2`), dealing with `while` construct breaks. In Figure 5, horizontal links in the `flow` graph are flowsynch links, the looping link in the `while` graph is a loopback cycle link, the two links towards the end tag in the `while` graph are escape cycle links; all the others are order links.

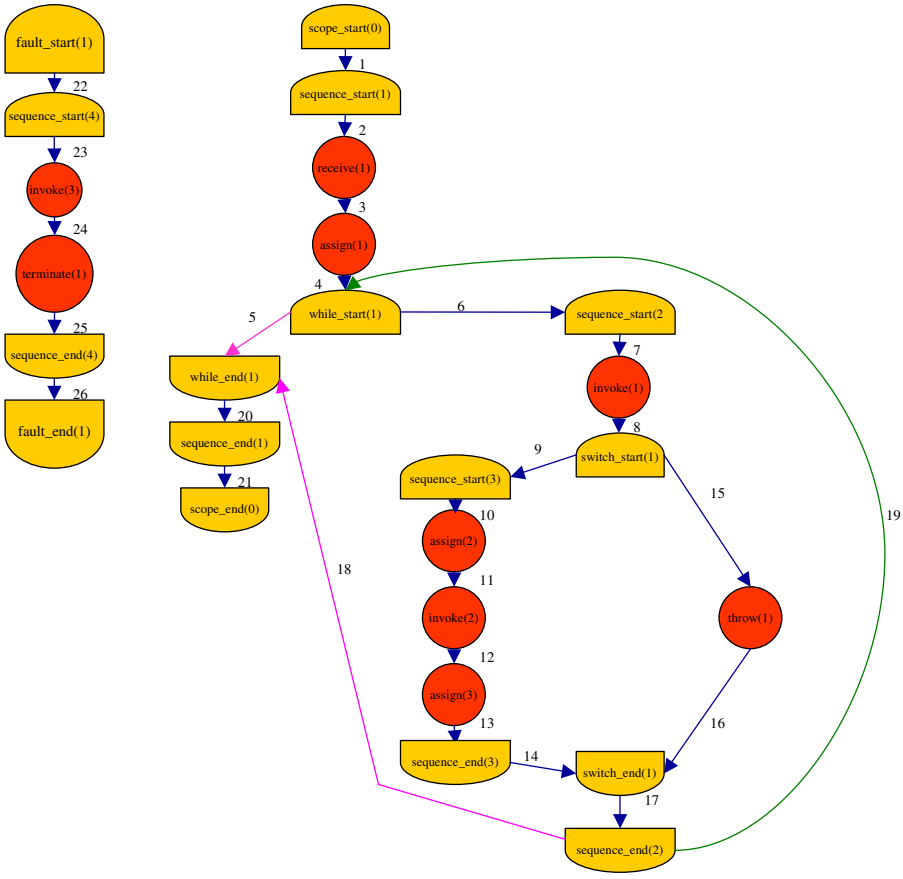

Fig. 6. Shipping Service Net

Since BPEL specifications allow scopes nested inside a flow to specify a whole (possible complex) structured activity, and since we map each scope onto a single tuple centre, their occurrence in the outer flow is tracked by a single activity node — which carries information on the tuple centre actually managing the scope. So, in the end, single activity nodes are used to model the BPEL basic activities `invoke`, `receive`, `reply`, `wait`, `terminate`, `assign`, `throw`, `empty`, and `compensate`, as well as a `scope`, while structured activities are mapped to subgraphs. As a further example of graph, discussed in next section, Figure 6 reports the graph corresponding to the shipping service specification described in Section 2 — where each node is given a unique identifier and each link a unique number — (the side subgraph corresponds to the fault handler).

5.2 Mapping a BPEL Net in ReSpecT tuples

A ReSpecT tuple centre executes a BPEL specification by keeping track of the graph structure, expressed in terms of a set of ReSpecT tuples. Such tuples also carry information on the current execution state, so that the dynamics of the net is obtained by evolving such information during execution. This is achieved through four kinds of tuples respectively representing tasks, links, start-tag/end-tag associations (called structures), and whole flows, as follows:

Task— Each node of a net, either an activity or a tag, is described by a tuple `task/10` of the kind:

```
task( Name,Type,Status,OrderCondition,JoinCondition,
      InLinkList,OutLinkList,TaskInfo)
```

`Name` represents the unique identifier of the task, and is of the kind `TaskName(n)` as shown in Figure 6; `Type` is either `Activity` or `Tag`; `Status` is either `ready`, `started`, `completed` or `stopped`; `OrderCondition` and `JoinCondition` represents conditions over the execution of the task as expressed by the BPEL specification; `InLinkList` and `OutLinkList` are the list of incoming and outgoing link identifiers; `TaskInfo` expresses further information concerning the task as expressed in the XML specification. In particular, the state of a net at a given time is tracked by evolving the argument `Status` of each such tuple, which transitates through the `ready/started/completed` sequence, and can also be reinitialised to `ready` (because of a `while`-loop) or `stopped` (e.g. because of faults).

Link— Each link is similarly described by a tuple `link/6` of the kind:

```
link( LinkName,LinkType,Source,Target,TransCond,LinkStatus )
```

`LinkName` is a unique natural number for the link as shown in Figure 6; `LinkType` is either `order`, `flowsync`, `cycle1`, or `cycle2` as described in previous section; `Source` and `Target` are the identifiers for the source and target tasks; `TransCond` represents the condition over the execution of the task as expressed by the BPEL specification; and `LinkStatus` is either `evaluated`, `non_evaluated`, and `aborted` — again reflecting dynamic aspects of execution.

Structure— Each association between a start tag and an end tag is described by a tuple `link/6` of the kind:

```
structure(StartTask,EndTask)
```

expressing the 1-to-1 binary association between tags with identifiers `StartTask` and `EndTask`.

Flow— The overall structure of a flow is described by the tuple `flow/4`:

`flow( FlowName,FlowType,TaskList,FlowStatus )`

`FlowName` is an identifier of this particular flow in the overall BPEL specification; `FlowType` is either `normal`, `compensation`, or `fault`, depending on the origin of the flow in the specification; `TaskList` is a list of the identifiers of all the tasks of this flow, starting with the initial start tag; and `FlowStatus` is either `ready`, `executing`, or `completed`.

The initial status of the ReSpecT tuples representing the shipping service specification are as shown in Figure 7; note that these are precisely the tuples produced by the ***interpreter*** agent from the actual (XML-based) BPEL specification.

In particular, we note the following aspects:

- The `JoinCondition` of each task is given by the term `status(n)`, where `n` is the incoming link of the task — or is given by a disjunction (`or`) of such terms when many incoming links exist.
- Some tasks necessarily carry additional information, corresponding to all the data specified as XML tag properties in the BPEL specification — summarised in the figure by the term `info(...)`. For instance, the initial receiving task (`receive(1)`) actually provides the following term [14]:

```
info( partnerlink(customer), portType(sns:shippingServicePT),
      operation(shippingRequest), variable(shipRequest), yes,
      [correlationset([shipOrder], yes)]
)
```

- The `TransitionCondition` property of links is generally set to `true`, with the exception of flowsynch links (not present in this flow) and cycle links. For instance, the condition for link 11 is expressed as:

```
condition(not(
    bpws:getVariableData('itemsShipped') <
    bpws:getVariableProperty('shipRequest','props:itemsTotal')
))
```

representing the boolean condition for the main (`while`) cycle.

5.3 Main Execution Cycle

To correctly evolve this structure of tuples and properly stimulating the agents realising activities, the ***bpel_engine*** tuple centre is to be explicitly programmed — that is, to act as a WVM.

The main idea behind the execution cycle is that each time a task completes, the target tasks of all its outgoing links are candidates for execution, but should be actually scheduled only if *(i)* all their incoming links and *(ii)* their order and join conditions are positively evaluated. Figure 8 provides the reactions realising this main execution loop of the engine.

Initially, the ***boss*** agent executes an `out(start_scope(0))` operation, meaning that the main scope is to be started. This is intercepted by reaction

```
flow(no_name,no_variable,normal,[scope_start(0),sequence_start(1),...,scope_end(0)],ready)
flow(shippingError,error,fault,[fault_start(1),invoke(3),terminate(1),fault_end(1)],ready)

task( scope_start(0),    tag, ready, true,      true, [ ], [1], no, no_info )
task( sequence_start(1), tag, ready, status(1), true, [1], [2], no, no_info )
task( receive(1),   activity, ready, status(2), true, [2], [3], no, info(...))
task( assign(1),    activity, ready, status(3), true, [3], [4], no, info(...))
task( while_start(1),    tag, ready, status(4), true, [4,19], [5,6], no, no_info)
task( sequence_start(2), tag, ready, status(6), true, [6], [7], no, no_info )
task( invoke(1),    activity, ready, status(7), true, [7], [8], no, info(...))
task( switch_start(1),   tag, ready, status(8), true, [8], [9,15], no, no_info)
task( sequence_start(3), tag, ready, status(9), true, [9],[10], no, no_info )
task( assign(2),    activity, ready, status(10),true, [10],[11],no, info(...))
task( invoke(2),    activity, ready, status(11),true, [11],[12],no, info(...))
task( assign(3),    activity, ready, status(12),true, [12],[13],no, info(...))
task( sequence_end(3),   tag, ready, status(13),true, [13],[14],no, no_info )
task( throw(1),     activity, ready, status(15),true, [15],[16],no, no_info )
task( switch_end(1),tag,ready,or(status(14),status(16)),true,[14,16],[17],no,no_info)
task( sequence_end(2),   tag, ready, status(17),true, [17],[18,19], no, no_info )
task( while_end(1), tag,ready,or(status(5),status(18)),true,[5,18],[20],no,no_info )
task( sequence_end(1),   tag, ready, status(20),true, [20],[21], no, no_info )
task( scope_end(0),      tag, ready, status(21),true, [21],[ ],  no, no_info )
task(fault_start(1),     tag, ready, true,      true, [ ], [22], no,no_info)
task(sequence_start(4),  tag, ready, status(22),true, [22],[23], no,no_info)
task(invoke(3),     activity, ready, status(23),true, [23],[24], no,info())
task(terminate(1),  activity, ready, status(24),true, [24],[25], no,info()))
task(sequence_end(4),    tag, ready, status(25),true, [25],[26], no,no_info)
task(fault_end(1),       tag,ready,status(26),true,[26],[ ],no,no_info)

link( 1, order, scope_start(0),   sequence_start(1), true, non_evaluated)
link( 2, order, sequence_start(1),receive(1),        true, non_evaluated)
link( 3, order, receive(1),       assign(1),         true, non_evaluated)
link( 4, order, assign(1),        while_start(1),    true, non_evaluated)
link( 5, cycle2,while_start(1),   while_end(1),      condition(...), non_evaluated)
link( 6, order, while_start(1),   sequence_start(2), condition(...), non_evaluated)
link( 7, order, sequence_start(2),invoke(1),         true, non_evaluated)
link( 8, order, invoke(1),        switch_start(1),   true, non_evaluated)
link( 9 ,order, switch_start(1),  sequence_start(3), true, non_evaluated)
link( 10,order, sequence_start(3),assign(2),         true, non_evaluated)
link( 11,order, assign(2),        invoke(2),         true, non_evaluated)
link( 12,order, invoke(2),        assign(3),         true, non_evaluated)
link( 13,order, assign(3),        sequence_end(3),   true, non_evaluated)
link( 14,order, sequence_end(3),  switch_end(1),     true, non_evaluated)
link( 15,order, switch_start(1),  throw(1),          true, non_evaluated)
link( 16,order, throw(1),         switch_end(1),     true, non_evaluated)
link( 17,order, switch_end(1),    sequence_end(2),   true, non_evaluated)
link( 18,cycle2,sequence_end(2),  while_end(1),      false, non_evaluated)
link( 19,cycle1,sequence_end(2),  while_start(1),    true, non_evaluated)
link( 20,order, while_end(1),     sequence_end(1),   true, non_evaluated)
link( 21,order, sequence_end (1), scope_end(0),      true, non_evaluated)
link( 22,order, fault_start(1),   sequence_start(4), true, non_evaluated)
link( 23,order, sequence_start(4),invoke(3),         true, non_evaluated)
link( 24,order, invoke(3),        terminate(1),      true, non_evaluated)
link( 25,order, terminate(1),     sequence_end(4),   true, non_evaluated)
link( 26,order, sequence_end(4),  fault_end(1),      true, non_evaluated)

structure( scope_start(0),    scope_end(0) )
structure( sequence_start(1), sequence_end(1) )
structure( sequence_start(2), sequence_end(2) )
structure( sequence_start(3), sequence_end(3) )
structure( sequence_start(4), sequence_end(4) )
structure( switch_start(1),   switch_end(1) )
structure( while_start(1),    while_end(1) )
structure( fault_start(1),    fault_end(1) )
```

Fig. 7. Shipping Service expressed as ReSpecT tuples

```
% (1): Initial Firing
reaction( out_r(start_scope(0)),(
    in_r(start_scope(0)),
    in_r(flow(FlowName,normal,[StartTask|List],ready)),
    out_r(flow(FlowName,normal,[StartTask|List],executing)),
    in_r(task(StartTask,tag,ready,true,true,...)),
    out_r(task(StartTask,tag,started,true,true,...))
)).
% (2): Tag trivial execution
reaction( out_r(task(TaskName,tag,started,true,true,InLinkList,OutLinkList,...)),(
    in_r( task(TaskName,tag,started,true,true,InLinkList,OutLinkList,...)) ,
    out_r( task(TaskName,tag,completed,true,true,InLinkList,OutLinkList,...)),
    out_r( update_outgoing_links(OutLinkList))
)).
% (3): Task execution scheduling
reaction( out_r(task(TaskName,activity,started,true,true,_,_,_,TaskInfo)),(
    out_r(task_to_do(TaskName,TaskInfo))
)).
% (4): Task execution acknowledge
reaction( out_r(task_success(TaskName,ResultInfo)),(
    in_r( task_success( TaskName, ResultInfo ) ),
    in_r( task( TaskName,activity,started,true,true,InLinkList,OutLinkList,...)),
    out_r( task( TaskName,activity,completed,true,true,InLinkList,OutLinkList,...)),
    out_r( update_ougoing_links(OutLinkList))
)).
% (5): Update of outgoing links
reaction( out_r(update_ougoing_links([Link|List])),(
    in_r( update_links([Link|List])),
    rd_r( link(Link,_,_,_,TransitionCondition,not_evaluated)),
    out_r( task_to_do(transition_evaluation,info(TransitionCondition,Link))),
    out_r( update_ougoing_links(List))
)).
% (6): Link evaluation success, with firing of (back) link-verification on target
reaction( out_r(task_success(transition_evaluation,info(TransRep,LinkName))),(
    in_r( link( LinkName,LinkType,Source,Target,_,not_evaluated) ),
    out_r( link( LinkName,LinkType,Source,Target,TransRep,evaluated)),
    rd_r( task(Target,_,ready,_,_,InLinkList,_,_,_,_)),
    out_r( verify_incoming_links(Target,InLinkList))
)).
% (7): Recursion, with check of positive evaluation
reaction( verify_incoming_links(Target,[Link|List]),(
    in_r( verify_incoming_links(Target,[Link|List])),
    out_r( verify_incoming_links(Target,List)),
    rd_r( link(Link,_,_,_,true,evaluated))
)).
% (8): Fixpoint, with scheduling of the task order_evaluation
reaction( verify_incoming_links(Target,[]),(
    in_r( verify_incoming_links(Target,[])),
    rd_r( task(Target,_,ready, OrderCondition,JoinCondition,_,_,_,_,_)),
    out_r( task_to_do( order_evaluation,info(Target,OrderCondition,JoinCondition)))
)).
% (9): Starting of a new task
reaction( out_r(task_success(order_evaluation,info(OrderRep,JoinRep,Target))),(
    in_r( task(Target,_,ready, _,_,_,_,_,_,_)),
    out_r( task(Target,_,started, OrderRep,JoinRep,_,_,_,_,_))
)).
% (10): Managing of a fault
reaction( out_r(task_failure(FaultName,FaultVariable)),(
    in_r(flow(FlowName,FlowVariable,normal,TaskList,executing)),
    out_r(flow(FlowName,FlowVariable,normal,TaskList,terminated)),
    out_r(freeze_status_of_current_flow),
    out_r(terminate(TaskList))
)).
```

Fig. 8. Reactions for the main execution cycle of the engine

(1) causing tuple `flow/4` to switch to the `executing` status (via an `in_r` / `out_r` sequence); the status of the initial task is then switched to `started`.

Each time the state of a task moves to `started`, it has to be scheduled for execution, hence either reaction (2) or (3) are fired, depending on whether the task is a tag or an activity. While in the first case the task automatically completes, in the second case a tuple `task_to_do(TaskName,TaskInfo)` is inserted in the space, representing the activity task to be executed. As shown in next section this event will eventually cause a tuple of the kind `task_success(TaskName,ResultInfo)` to be inserted in the space, meaning the task has been completed. Therefore, for both tags (reaction 2) and activities (reaction 4), when they complete their tuple state is moved to `completed` and tuple `update_outgoing_links(OutLinkList)` is inserted in the space, firing the whole process in charge of finding new tasks to be executed.

By reaction (5), the first link of the list is considered, the evaluation of its `TransCond` is scheduled as if it were a true task — by means of the insertion of a proper `task_to_do` tuple — and finally the reaction is fired again on the remaining links in the list. As a condition is positively evaluated, by reaction (6) the corresponding link is moved to the `evaluated` state, so that its target task becomes a potential candidate for scheduling. This is actually the case only if *all* its incoming links have been evaluated, therefore a verification is fired by inserting a tuple `verify_incoming_links`. Reaction (7) handles the recursive phase of such verification: if the first link in the list has been already positively evaluated, the verification process proceeds on the list tail by firing reaction (7) again. Reaction (8) handles the fixpoint phase: if the list is empty the task can be executed, but before doing so its `OrderCondition` and `JoinCondition` are to be evaluated. Again this is realised by a `task_to_do` and a `task_success` tuple, by which in reaction (9) the task is marked as started. The execution cycle is therefore reactivated at reactions (2) or (3). Finally, if any task fails reporting a fault name and an error (`task_failure`), reaction (10) reacts to this by terminating current flow and executing the flow associated to the fault.

This specification is independent from the entity executing the single task, we simply supposed that as a `task_to_do` tuple is inserted, the correspondingly tuple `task_success` or `task_failure` would eventually appear. This transparently supports the possibility of having external agents executing tasks, as discussed in Section 4. As a first case, this is necessary for those tasks involving interactions of the orchestration Web service with other Web services, namely activities `invoke`, `receive`, and `reply`. For them, external agents are in charge of either listening for incoming messages and properly notifying the tuple centre when it is ready to receive, or waiting for the tuple centre to

request some new message to be sent. As a second case, this is also necessary for those tasks involving interaction with another tuple centre, namely for the management of handlers, for invoking other scopes, or for reading/updating the content of a variable. As a third case, the burden of some activities may be moved to external agents just because it is about more algorithmic computation aspects, including evaluating computations of expressions and of boolean conditions.

In the current version of our implementation: *(i)* only simple activities `empty`, `terminate`, and `throw` are internally executed; *(ii)* activities `scope` and `compensate` are resolved by interacting with other tuple centres; *(iii)* activities `invoke`, `receive`, `reply`, `wait` and `assign`, along with condition evaluations, are delegated to one external agents each. Each such agent, then, repeatedly waits for removing a `task_to_do` tuple, executes the corresponding task, and provides the tuple `task_success` as result.

5.4 Other aspects

The reactions presented in Figure 8 are a necessarily simplified version of the actual implementation, abstracting away from a number of details for the sake of space and ease of presentation. Other aspects of interest are as follows:

While— Handling loops of a `while` introduces a complication in the net evolution, for we may need to move the state of all activities in the construct body from `completed` back to `ready`. This event is intercepted as a cycle2 link is evaluated, and is managed by a bunch of reactions properly propagating this state-change.

Pick— The `pick` construct specifies a finite set of `receive` tasks to be executed in mutual exclusion: as one completes the other should be immediately aborted. This is realised by letting each such task carry in its `Suppress` field the list of tasks to abort, and moving their state from `started` to `aborted` with proper reactions.

Compensate— As the *compensate* activity is to be executed, a tuple `compensate(scope(Y))` is inserted in the proper tuple centre, by which its flow starts. The main difference between compensation flows and normal flows is that the former manage variables locally — that is, in the same tuple centre —, so it needs to install them at the beginning and uninstall them at the end of the flow.

6 Conclusions and Future Works

Designing and developing orchestration and workflow engines is a hard task because of the complexity of managing interaction and concurrency in a general way. ReSpecT tuple centres allow such a complexity to be faced at a proper abstraction level, exploiting constructs and mechanisms explicitly thought for manipulating interactions and shaping the coordination flow.

On the one side, the data-orientation of the model has been fundamental for enhancing the uncoupling among the concurrent activities and for representing BPEL data, communications and the structure and state of BPEL orchestration activities as inspectable and changeable first-class objects (tuples). On the other side, the programmability of the medium and the computing power of ReSpecT have been the key for setting up general-purpose engines, powerful enough to encode and execute full- fledged BPEL specifications.

This approach promotes the integration of multi-agent systems and coordination infrastructures with standard, mainstream Web technologies for the engineering of complex distributed application. Accordingly, future works account for testing the approach with real-world applications (such as logistics), setting up a full orchestration management system based on agents — working behind Web services — using ReSpecT tuple centres as BPEL orchestration engines, on top of TuCSoN coordination infrastructure and related technology.

References

[1] Bergstra, J. A., A. Ponse and S. A. Smolka, editors, "Handbook of Process Algebra," North-Holland, 2001.

[2] Bocchi, L., C. Laneve and G. Zavattaro, *A calculus for long-running transactions*, in: *Formal Methods for Open Object-Based Distributed Systems (FMOODS 2003)*, Lecture Notes in Computer Science **2884** (2003), pp. 124–138.

[3] Butler, M. and C. Ferreira, *A process compensation language*, in: *Integrated Formal Methods IFM2000*, Lecture Notes in Computer Science **1945** (2000), pp. 61–76.

[4] Christensen, E., F. Curbera, G. Meredith and S. Weerawarana, *Web Services Description Language (wsdl) 1.1* (2001), http://www.w3.org/TR/wsdl.

[5] Denti, E., A. Natali and A. Omicini, *On the expressive power of a language for programming coordination media*, in: *Proc. of the 1998 ACM Symposium on Applied Computing (SAC'98)* (1998), pp. 169–177.

[6] Gelernter, D., *Generative communication in Linda*, ACM Transactions on Programming Languages and Systems **7** (1985), pp. 80–112.

[7] Koshkina, M. and F. van Breugel, *Modelling and verifying web service orchestration by means of the Concurrency Workbench*, SIGSOFT Softw. Eng. Notes **29** (2004), pp. 1–10.

[8] Mazzara, M. and R. Lucchi, *Framework for generic error handling in business processes*, in: *Workshop on Formal Models for Web Services (WS-FM2004)*, Electronic Notes in Theoretical Computer Science **105**, Elsevier Science B. V., 2004 pp. 133–145.

[9] Omicini, A. and E. Denti, *Formal* ReSpecT, in: A. Dovier, M. C. Meo and A. Omicini, editors, *Declarative Programming – Selected Papers from AGP'00*, Electronic Notes in Theoretical Computer Science **48**, Elsevier Science B. V., 2001 pp. 179–196.

[10] Omicini, A. and E. Denti, *From tuple spaces to tuple centres*, Science of Computer Programming **41** (2001), pp. 277–294.

[11] Omicini, A. and F. Zambonelli, *Coordination for Internet application development*, Autonomous Agents and Multi-Agent Systems **2** (1999), pp. 251–269, special Issue: Coordination Mechanisms for Web Agents.

[12] Peltz, C., *Web Services orchestration and choreography*, IEEE Computer **36** (2003), pp. 46–52.

[13] Ricci, A., A. Omicini and E. Denti, *Virtual enterprises and workflow management as agent coordination issues*, International Journal of Cooperative Information Systems **11** (2002), pp. 355–379.

[14] Tony Andrews et al. , *Specification: Business Process Execution Language for Web Services version 1.1* (2003), http://www-106.ibm.com/developerworks/webservices/library/ws-bpel/.

[15] Viroli, M., *Towards a formal foundational to orchestration languages*, in: *Workshop on Formal Models for Web Services (WS-FM2004)*, Electronic Notes in Theoretical Computer Science **105**, Elsevier Science B. V., 2004 pp. 51–71.