

RESTful Petri Net Execution

Gero Decker, Alexander Lüders, Hagen Overdick,
Kai Schlichting, and Mathias Weske

Hasso-Plattner-Institute, University of Potsdam, Germany
(gero.decker,hagen.overdick,weske)@hpi.uni-potsdam.de
(alexander.lueders,kai.schlichting)@student.hpi.uni-potsdam.de

Abstract. Representational State Transfer (REST) has received a lot of attention recently as architectural style for distributed systems made up of loosely coupled resources. While most research in process enactment focuses on BPEL and SOAP, most internet applications are based on REST. To leverage this new architectural style also for process enactment, this paper introduces process enactment in REST environments. The approach is based on Service Nets, a specific class of Petri nets supporting value passing and link passing mobility. Implementation considerations of a prototype are presented. The approach is compared with the traditional BPEL/SOAP approach to process enactment.

1 Introduction

The service-oriented architecture (SOA) is an architectural style for building software systems based on services. Services are loosely coupled components that can be discovered and composed [6]. Such composition is often realized through process execution engines, interpreting business process models and invoking services accordingly. Using SOAP as communication protocol is a typical option for realizing web services [8]. Furthermore, the Business Process Execution Language (BPEL [10]) is a widely used standard for implementing business processes that are based on SOAP services.

SOAP services are a concrete implementation of a SOA, yet there are alternatives readily available. In [20], we characterized Representational State Transfer (REST [11]) as a restricted subset of SOA, hence RESTful usage of the Hyper Text Transfer Protocol (HTTP [12]) qualifies as SOA just as well. The most important restrictions imposed by REST are globally unique identification of each service instance (called resource) and identification of the interaction intention at the protocol level. HTTP supports *resource reflection* (GET), *at-least once delivery* (PUT/DELETE), and *at-most once delivery* (POST) directly, other intention can be represented by combining the former. In essence, SOAP-based services merely use HTTP as a transfer protocol, REST advocates HTTP as application protocol, enabling increased distributability, scalability and mashability of service-based systems. A resource-oriented approach as demanded by REST has proven strengths in environments of multiple autonomous peers [27], the World Wide Web being the most prominent example of such a system.

Most research in the area of process-oriented service implementations focuses on BPEL and SOAP-based services, where machine-to-machine communication is at the center of attention. On the other hand, most successful internet applications (e.g., flickr.com, amazon.com, XING.com) are based on the REST architecture style. This paper introduces process enactment in REST environments, i.e., RESTful process enactment. The approach is conceptually based on Service nets, a specific class of high level Petri nets that include value passing, i.e. colored tokens and guard conditions. Dynamically evolving structures realized through URI passing is a core aspect in the REST world. Therefore, this notion of link passing mobility will also be captured in the formal model.

The remainder of this paper is structured as follows. The next section will present a motivating example and explain central REST concepts. Section 3 introduces the formal model specifying RESTful execution of processes specified by Service nets, before section 4 introduces implementation concepts of a prototypical engine that we have implemented. Section 5 reports on related work, especially focusing on the relationship of the presented approach to the BPEL/-SOAP approach to process enactment in web environments. Section 6 concludes and points to future work.

2 Motivating Example and Approach

Figure 1 shows the example we will use for illustration throughout this paper. The typical notation for Petri nets is used, where circles denote places, rectangles denote transitions and arrows flow connections between places and transitions. Read arcs as special kind of flow connection are represented by lines without arrowheads. The dashed rectangles denote different nets. The dashed arrows between transitions of different nets denote that the same transition appears in different nets.

Several participants are involved in the sample scenario: While browsing an online store, a customer creates a shopping cart where she selects items she is interested in. Before submitting the order, she is also allowed to already define the address where the goods should be delivered to. Once she is sure what to buy, she submits the order, triggering subsequent payment handling and delivery, which in turn can be done concurrently. Payment is handled through an external payment service. The customer is automatically forwarded to the respective web site. There are two alternatives for delivery: standard delivery and express delivery. For each alternative there is a respective service.

All interaction between two participants are carried out through HTTP requests/response cycles, represented as communication transitions in Figure 1. We mentioned before, that the HTTP reflects the intention of an interaction directly at the protocol level. REST calls this feature a uniform interface and HTTP provides the following verbs to express intentions:

GET. Messages labeled as GET have an empty service request and are guaranteed to have no effect within the receiver of such request, i.e. they are *safe* to

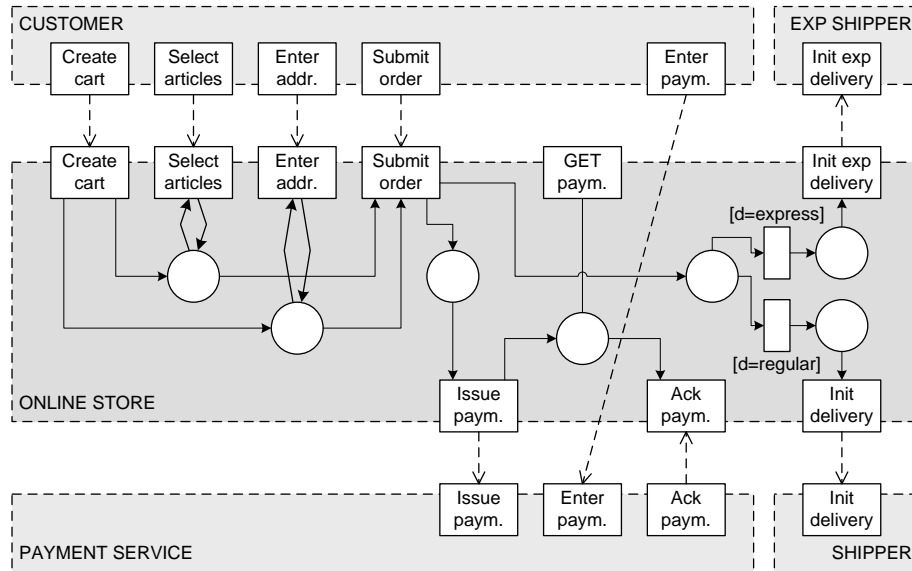


Fig. 1. Sample scenario

call. GET responses are expected to be a description of the current state of the targeted resource. Also, as GET does not alter the state of the targeted resource, the response can be cached. This has great benefits to a distributed architecture and both aspects can be seized without prior semantic knowledge of the targeted resources.

PUT. Messages labeled as PUT do cause an effect in the targeted resource, but do so in an *idempotent* fashion. An idempotent interaction is defined as replayable, i.e. the effect of N identical messages is the same as that of 1. In a distributed system, where transactions may not be readily available, this is a great help to recover from situations where messages might have got lost. Here, it does not harm to simply resend a message. Again, this assumption can be made without any prior semantic knowledge of the resource involved.

DELETE. Messages labeled as DELETE do cause an effect in the targeted resource, where that effect has a negative connotation. Just as PUT, DELETE is defined as *idempotent*. However, as with all messages, the interpretation is solely the responsibility of the receiver, i.e. a DELETE has to be regarded as “please terminate”.

POST. All other types of messages are labeled as POST, i.e. they cause an effect in the receiver and they are not safe to replay. This is a catch-all mechanism for all messages that can not be described by the prior verbs. Without a uniform interface, all messages would be treated like this, loosing context-free resource

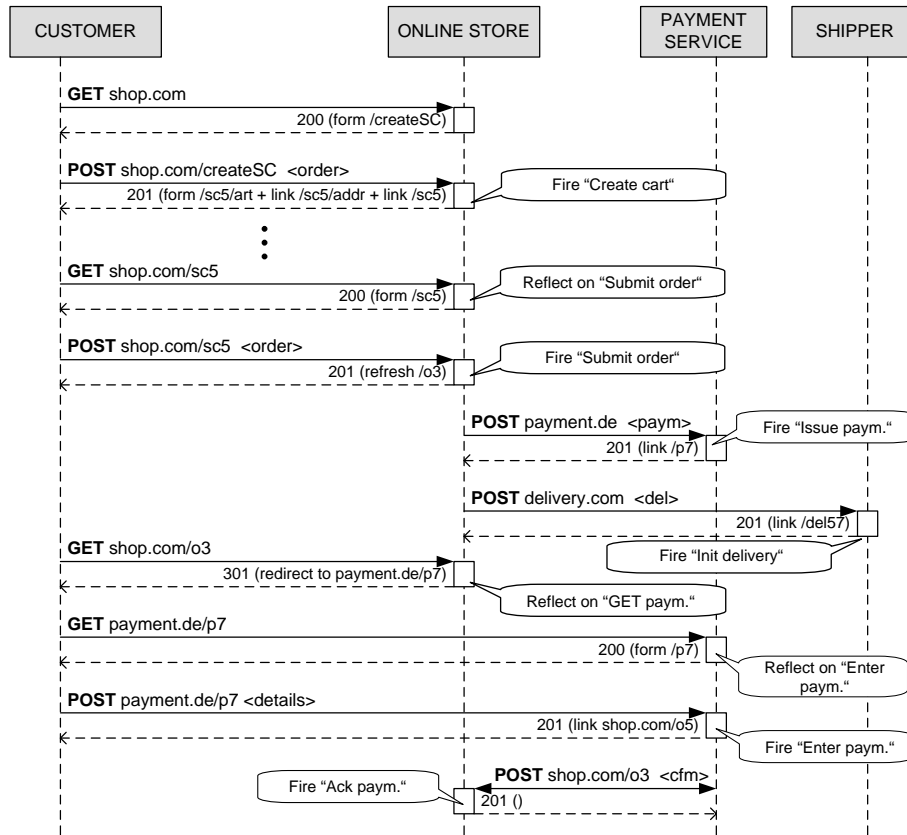


Fig. 2. Sample interaction sequence

reflection, caching and replayability.

The state of the online store service is represented by the marking of the Petri net. Most *GET* requests do not have any explicit representation in the net. The only exception in the example is the “GET paym.” transition. As this transition only has a read arc, firing it does not affect the marking. Therefore, this communication is safe.

Unsafe communication corresponds to firing of the other transitions, in the context of the paper we resort to using *POST*. For instance selecting articles removes the token from the input place and produces a (possibly) different token to the same place. The internals of the payment and delivery services are not shown in Figure 1.

Figure 2 shows a sample sequence of message exchanges. Here, *GET* requests are also included. The first interaction happens between the customer’s web browser and the online store. A *GET* request is issued for `http://shop.com`. As response, the HTTP code 200 (OK) is returned with an XHTML page as

representation for `http://shop.com`. The intention of this interaction is to receive a representation of the targeted resource. This representation contains the reference to the shopping cart creation resource, namely `http://shop.com/createSC`. Invoking this service results in the creation of a new resource, identified by `http://shop.com/sc5`. Here, we already see how the topology dynamically evolves and navigation from one resource to another happens through URI passing.

Here comes in another vital feature of REST: hypermedia as the engine of application state. In a Petri net, application state is the position of all tokens in a net, the marking, at a given time. Calling a service is mapped to a transition with a certain set of input tokens in the underlying net. As we just learned, a new URI representing a transition (with input tokens), we can fire tokens in an at-most once fashion, therefore mapped to *POST*ing to the order service. The contained XML document is used as input to the service, the result is the creation of a new resource and returning a *201 created* response including the link to the newly created resource, here `http://shop.com/sc5` and a representation of the resource including references to the services `http://shop.com/sc5/art` and `http://shop.com/sc5/addr`. These services in turn return XHTML pages providing forms for selecting items and a delivery address respectively.

The remaining interactions correspond to submitting the order, triggering the payment service and triggering the delivery service. Upon *GET* request by the customer, the online store redirects her to the payment service.

GET requests are a resource reflection mechanism in the REST world. In our scenario, the returned representation of the identified resource describes how to interact with the resource and what data is being expected. In our scenario, all representations are optimized for rendering a human-readable web page in a browser. However, this information can also be used by a machine. An alternative representation could be a WSDL file also defining the data structure expected in a request for SOAP-legacy integration or more advanced techniques such as microformats [15] and RDFa [1]. If different representations are available, content negotiation realizes the selection of a desired representation.

Figure 1 contains several sample Universal Resource Identifiers (URI [3]). The concept of web-wide unique identification of resources is at the center of REST. We can distinguish between at least two interesting types of resources to be identified:

- *Static ports* are entry points into process instances. *POST*ing data to such resources leads to the creation of activity instances or the data sent is routed to existing process instances. Static means that the URI is independent of any particular process instance. In our example, `http://shop.com/createSC` or `http://payment.de` identify static ports.
- *Dynamic ports* are also entry points into process instances, but here a dynamic port corresponds to exactly one activity instance. In our example, `http://shop.com/o3` or `http://payment.de/p7` identify dynamic ports.

The notion of dynamic ports or activity instances is not present in SOAP-based systems, where only static ports are available. Here, application-specific

parameters are used for relating requests to process instances. This hampers the possibility of “bookmarking” activity instances, one of the driving features of the World Wide Web.

In the REST context it is crucial to avoid “URI guessing”, i.e. all URIs that are actually addressed in a request must have been obtained somehow before. This implies that it should never be demanded that requesters know how to construct particular URIs, e.g. constructing the URI `http://shop.com/o3` from the store’s URI and the store’s internal Id of the shopping cart. This URI must have been passed to the customer previously. Again, the concept of *link passing mobility* [19] is of central importance for RESTful systems and taken even beyond by treating link passing mobility as the driver of the application flow, where the application is completely located within the client, the server side is simply providing services.

As all interactions with such services have explicit intention, exploiting edge conditions such as caching *GET* interaction possible without application knowledge on either side of the communication. The message itself is enough for any intermediary to optimize its own behavior and in turn optimize the operating cloud in total.

3 Formal Model

All message exchanges between the online store and its environment happen via HTTP request/response interactions. As already illustrated in Figure 1 such synchronous communication is modeled in the Petri net using communication transitions. This section will introduce *service nets* specifying the behavior of systems that implement processes in a RESTful manner.

The online store receives XML documents from the customer’s browser and the payment service and sends XML documents to the payment and the delivery services. The tokens flowing within the online store also carry XML data. Branching decisions are based on such XML-tokens.

3.1 Basic Definitions

In the following definitions we will denote the (infinite) set of all XML documents as *XML* and the (infinite) set of all URIs as *URI*.

Definition 1 (Service Net). A service net is a tuple $S = (P, T, F, F_{read}, T_S, T_R, init, g, uri)$ where

- P and T are disjoint sets of places and transitions,
- $F \subseteq (P \times T) \cup (T \times P)$ is the flow relation,
- $F_{read} \subseteq F \cap (P \times T)$ is a set of read arcs,
- $T_S, T_R \subseteq T$ are disjoint sets of send and receive transitions, collectively called communication transitions,
- $init : P \rightarrow MS(XML)$ is the initial marking, a function assigning multi-sets of tokens to places,

- g is a function assigning guard conditions to transitions, where a condition $g(t) \subseteq (\bullet t \rightarrow XML)$ specifies combinations of input documents and
- uri is a function assigning URIs to tuples of communication transitions and combinations of input documents, i.e. $uri(t) : (\bullet t \rightarrow XML) \rightarrow URI$.

The auxiliary function $\bullet t$ denotes all input places for a transition t , i.e. $\bullet t = \{p \in P \mid (p, t) \in F\}$, in analogy to this $t\bullet$ denotes all output places for a transition.

The definition of service nets shows how the distinction between static ports and dynamic ports is formally reflected: any receive transition t without input places is a static port. Here exists a URI id , such that $uri(t, \emptyset) = id$. Dynamic ports are characterized by a tuple (t, f_{in}) where $f_{in} : \bullet t \rightarrow XML$, i.e. by a receive transition with a set of input documents. Such a dynamic port's URI is given by $uri(t, f_{in})$.

The definition of function g allows for the same expressiveness as using boolean expressions that evaluate to true or false for given input documents. Imagine a transition t with one input place p . A sample guard condition could be $g(t) = \{\{(p, xml_p)\} \mid \langle shippingType \rangle express \langle /shippingType \rangle \text{ is part of } xml_p\}$.

As seen in the motivating example, firing receive transitions might or might not result in state changes. In this context read arcs are a central feature. Firing transitions without outgoing arcs and only with read arcs as incoming arcs will not change the system's state and therefore is *safe*. Such transitions are solely used for resource reflection. However, this reflection is restricted to certain states of the system – defined by the read arcs.

Definition 2 (Transition Modes, Enablement and Firing). *Let $(P, T, F, F_{read}, T_S, T_R, init, g, uri)$ be a service net. A transition mode is a tuple $(\sigma_{in}, t, \sigma_{out})$ where $\sigma_{in} : \bullet t \rightarrow XML$ assigns documents to the input places of $t \in T$ and $\sigma_{out} : t\bullet \rightarrow XML$ documents to output places.*

A transition mode $tm = (\sigma_{in}, t, \sigma_{out})$ is enabled in marking m iff $\sigma_{in} \in g(t)$ and $\forall p \in \bullet t [\sigma_{in}(p) \in m(p)]$. The reached marking after firing of tm is m' , where $m'(p) := m(p) - \{\sigma_{in} \mid q \in P \mid (q, t) \notin F_{read}(p)\} + \{\sigma_{out}(p)\}$.

The firing semantics of service nets is similar to that of classical place / transition nets in the sense that a transition is enabled only if there is at least one token on each input place. Firing of a transition will lead to consuming one token from each input place (except in the case of read arcs) and producing one token onto each output place. Guard conditions further restrict the enablement of transitions. As tokens carry values, we speak of *transition modes*, i.e. bindings of values to input and output places of a transition.

We see that T_S , T_R and uri have no influence on the firing semantics of an individual service net. They are essential for the communication behavior, which is manifested in the composition of service nets.

3.2 Composition of Service Nets

The interaction behavior between multiple service nets is specified by the following definition of service net composition. We distinguish between *closed world*

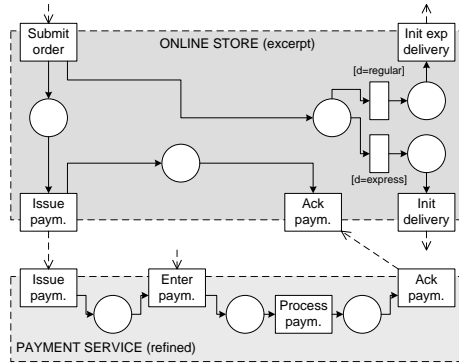


Fig. 3. Excerpt from the example

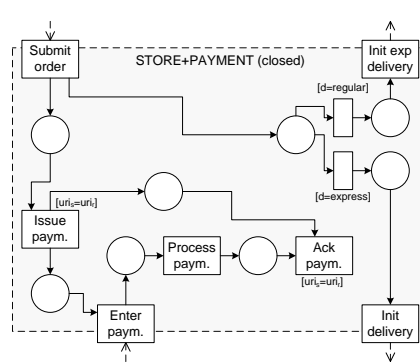


Fig. 4. Closed world composition

composition and *open world composition*. In a closed world, a communication transition is not available for communication any longer, once it is used in the composition. The definition of the open world is the more realistic one, where the same port can be used by different other services.

Definition 3 (Closed World Composition). Let S_1 and S_2 be two service nets, where $P_1 \cap P_2 = \emptyset$ and $T_1 \cap T_2 \subseteq ((T_{S_1} \cap T_{R_2}) \cup (T_{S_2} \cap T_{R_1}))$. The closed world composition $S_1 \oplus_c S_2$ is the service net $(P', T', F', F'_{read}, T'_S, T'_R, init', g', uri')$ where

- $P' = P_1 \cup P_2$, $T' = T_1 \cup T_2$, $F' = F_1 \cup F_2$, $F'_{read} = F_{read1} \cup F_{read2}$,
- $T'_S = (T_{S_1} \cup T_{S_2}) \setminus (T_1 \cap T_2)$,
- $T'_R = (T_{R_1} \cup T_{R_2}) \setminus (T_1 \cap T_2)$,
- $init' = init_1 \cup init_2$,
- $g'(t) = (g_1 \cup g_2)(t)$ for all $t \in (T_1 \cup T_2) \setminus (T_1 \cap T_2)$ and else $g'(t) = \{f_1 \cup f_2 \mid f_1 \in g_1(t) \wedge f_2 \in g_2(t) \wedge uri_1(t, f_1) = uri_2(t, f_2)\}$ and
- $uri' = (uri_1 \cup uri_2)|_{T'_S \cup T'_R}$.

The basic idea is to merge corresponding send and receive transitions when composing two service nets. As a transition might correspond to a number of ports, it is crucial to ensure that the URI addressed by the sender matches the URI offered by the receiver. This is manifested in the definition of $g'(t)$, where this matching of URIs is added as additional guard condition to the merged transitions. This URI matching realizes link passing mobility in service nets.

Figure 4 shows an example where parts of the online store’s service net is composed with a service net describing the payment service. Here, the transitions “issue payment” and “ack. paym.” are not communication transitions any longer.

Definition 4 (Open World Composition). Let S_1 and S_2 be two service nets and $S_1 \oplus_c S_2 = (P, T, F, F_{read}, T_S, T_R, init, g, uri)$. Then the open world composition $S_1 \oplus_o S_2$ is the service net $(P, T', F', F'_{read}, T'_S, T'_R, init', g', uri')$, where

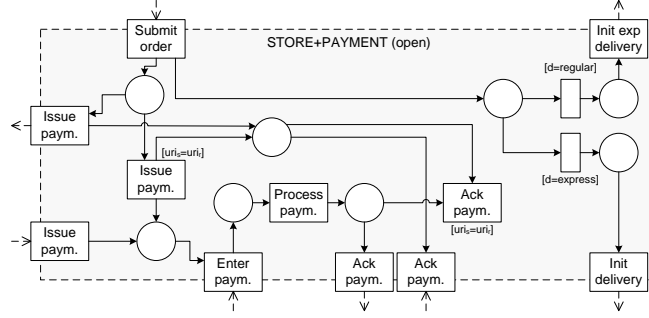


Fig. 5. Open world composition

- $T' = T \cup T_{new1} \cup T_{new2}$, where T_{new1} and T_{new2} are sets of new transitions where for each $x \in (T_1 \cap T_2)$ there is a transition t_{x1} in T_{new1} and a transition t_{x2} in T_{new2} ,
- $F' = F \cup \{(p, t_{x1}) \mid \exists p, x ((p, x) \in F_1)\} \cup \{(t_{x1}, p) \mid \exists p, x ((x, p) \in F_1)\} \cup \{(p, t_{x2}) \mid \exists p, x ((p, x) \in F_2)\} \cup \{(t_{x2}, p) \mid \exists p, x ((x, p) \in F_2)\}$,
- $F'_{read} = F_{read} \cup \{(p, t_{x1}) \mid \exists p, x ((p, x) \in F_{read1})\} \cup \{(p, t_{x2}) \mid \exists p, x ((p, x) \in F_{read2})\}$,
- $T'_S = (T_S \cup T_{new1} \cup T_{new2}) \cap (T_{S1} \cup T_{S2})$,
- $T'_R = (T_R \cup T_{new1} \cup T_{new2}) \cap (T_{R1} \cup T_{R2})$,
- $g'(t) = g(t)$ for all $t \in T$ and else $g'(t) = g_1(t)$ if $t \in T_{new1}$ and $g'(t) = g_2(t)$ if $t \in T_{new2}$ and
- $uri'(t) = uri(t)$ for all $t \in (T_S \cup T_R)$, $uri'(t) = uri_1(t)$ for all $t \in T_{new1}$ and $uri'(t) = uri_2(t)$ for all $t \in T_{new2}$.

Figure 5 illustrates the outcome of an open world composition for the same example. Here, payment might be issued to another service and other services might still issue payment. The same applies to the payment acknowledgment.

Regarding enablement and firing of service nets we assume that exactly one token is removed from every input place and exactly one token is placed onto every output place. That way, service nets can be simulated by corresponding place/transition nets. The only exception are read arcs, where corresponding tokens must be present on the place for a transition to be enabled. However, the token will not be consumed upon firing. For simulating this behavior in place/transition nets, read arcs could be seen as bi-flows. This works as long as the respective places that are read from are not output places at the same time.

We assume that there is no functional dependency between input token values and output token values, i.e. firing the same transition with the same input token values twice might yield different output token values.

4 Implementation Considerations

This section presents the service net execution engine we implemented. It behaves as specified in the previous section. Figure 6 provides an overview of the

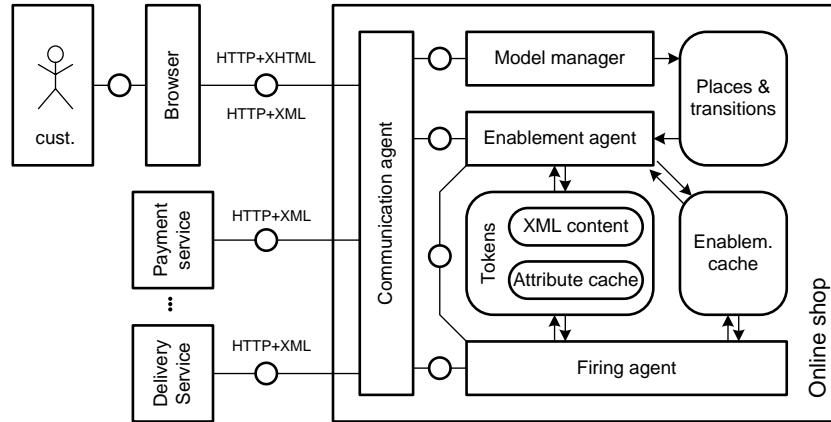


Fig. 6. Architecture of the service net execution engine

engine's overall architecture using the FMC block diagram notation [16]. Four main components can be distinguished within the engine:

- The *communication agent* handles incoming HTTP requests from the customer's web browser and the payment service and forwards them to the other agents. Furthermore, it issues HTTP requests to the payment service and the delivery services.
- The *model manager* deploys new Petri nets within the engine. The Petri Net Markup Language (PNML [4]) is used with engine-specific extensions. Internal representations of places and transitions are created.
- The *enablement agent* computes which transitions are currently enabled for what combinations of input tokens. This agent also evaluates guard conditions. If transitions are enabled and do not rely on an incoming HTTP request to be fired, the enablement agent triggers the firing agent.
- The *firing agent* is responsible for the firing of transitions. Firing leads to the deletion of tokens and the creation of new ones.

4.1 Concurrency

The engine runs within a web container and takes advantage of the multi-threading capabilities offered by the container. Parallel incoming HTTP requests are handled by different threads. The conflict between firing two transitions with the same input token is resolved on the database transaction level.

In the case of receive transitions, first the input tokens are consumed, then a response is returned to the requester, before output tokens are produced. This in turn immediately triggers the evaluation for enablement of subsequent transitions, which happens within the same thread. If such a subsequent transition is actually enabled, firing will occur immediately. Therefore, a certain sequentialization regarding internal transitions and send transitions applies. In case a send

The screenshot shows a web browser window with the title "Add article - Mozilla Firefox". The address bar displays "http://wwwserver:3000/transitions/75/case/49". The main content area is titled "Select articles" and contains a table with the following data:

Quantity	Article No	Price	Total
5	B0013IJ2LQ	3.00	\$ 15
2	B0015L4WBQ	6.50	\$ 13
1	3832180575	12.99	\$ 12.99

Below the table, there are three empty input fields for quantity, article number, and price. At the bottom left, it says "Total: \$ 40.99" and there is a "Send" button.

Fig. 7. Screenshot for “Select articles”, realized using XForms

transition is enabled and the server handling requests for the corresponding URI does not respond or returns an error message, another outgoing HTTP request will be issued again later. A particular worker thread is assigned to realize such requests.

4.2 XForms Representations

In our case the web resources addressed are static and dynamic ports. Forms are the classical way for describing the data expected by a web resource. XForms [5] are a way for not only defining the syntax of expected XML documents but also prescribe how to render this XML information in an interactive web form. XForms is suited not only for interpretation by humans through web browser but also by machines, as the specification of the expected XML document can be given e.g. using an XML schema.

Figure 7 shows a screenshot of the form for the “select articles” transition from section 2. Upon submission of the form, the browser assembles an XML document as specified in the XForms model and sends it to the given URI as POST message.

4.3 Interchange Format

The Petri Net Markup Language (PNML [4]) is used as input format for the engine. While the concepts of places, transitions and arcs are already present in PNML, we added engine-specific extensions. Listing 1 shows a PNML code snippet for the example from section 2.

The listing shows two transitions and one place definition. The engine distinguishes four types of transitions: firing *receive* transitions is triggered through

Listing 1 PNML code snippet for the example

```

<transition type="receive" id="select_articles">...
  <toolspecific tool="Petri Net Engine" version="1.0">
    <output>
      <bindings href="http://wwwserver/select_articles/bindings.xml"/>
      <form href="http://wwwserver/select_articles/form.xml"/>
    </output>
  </toolspecific>
</transition>...
<transition type="automatic" id="forward_express_delivery">...
  <toolspecific tool="Petri Net Engine" version="1.0">
    <guard><expr>deliveries.shippingType=='express'</expr></guard>
  </toolspecific>
</transition>...
<place id="deliveries">...
  <toolspecific tool="Petri Net Engine" version="1.0">
    <locator>
      <name>shippingType</name><type>xsd:string</type>
      <expr>//shippingType/text()</expr>
    </locator>
  </toolspecific>
</place>...

```

incoming HTTP requests, firing *send* transitions results in outgoing HTTP requests, *automatic* transitions are internal transitions and *referer* transitions are used for GET messages only, referring the requester to another URI.

The definition of `select_articles` includes a reference to a XForms document. The *bindings* define how input token values are used as in the form. Transition `forward_express_delivery` includes the definition of a guard condition. `shippingType` is a so called *locator* for place `deliveries`. What part of the XML document is actually referenced by this locator is defined in the definition of place `deliveries`. This indirection mechanism allows caching of individual attributes that are relevant for guard conditions.

5 Related Work

This paper has used Petri nets as formal foundation for describing RESTful process execution. Petri nets are described in detail in [24] and colored Petri nets in [14]. The introduction of XML technology into Petri nets has already been done in [17] in the form of *XML nets*. Here, tokens carry XML documents that are consumed in and produced by transitions.

Petri nets have extensively been used for representing systems with interfaces to an outside world. In the case of *open workflow nets*, places serve as message channels that connect different systems. These nets can be used for deciding whether there are partners with which the system could interact successfully [25]

and how such partners need to look like [18]. Using communication transitions for representing synchronous communication was already introduced in [28].

π -calculus is a process algebra that could be used as alternative to the service nets presented in this paper [19]. π -calculus directly supports link passing mobility. The distinction between static and dynamic ports made in this paper corresponds to free and bound names in π -calculus. The motivation for choosing Petri nets instead was driven by the need for using the Business Process Modeling Notation (BPMN) as high-level modeling language, and generating executable definitions out of it. Here, we could resort to existing implementations¹ doing BPMN to Petri net transformations, which are based on [9].

A first comparison between SOAP and REST as alternative technical grounding for service choreographies can be found in [29]. Although REST raises major interest among practitioners, it remains rather undiscussed in academia. Among the few academic papers concerning REST are [27, 22].

RESTful process execution can be seen as alternative to service composition as proposed in Business Process Execution Language (BPEL [10]). A main difference is that BPEL only offers static ports. Relating messages to process instances is done by application-specific attributes, grouped into so called correlation sets. This hampers caching on the protocol level and does not allow for bookmarking of activity instances or process instances. Reflection is realized by event handlers in BPEL that do not alter the values of variables, resulting again in POST messages. Therefore, the communication intentions inherent in HTTP are largely ignored in BPEL. We have proposed an extension called RBPEL in [21], introducing dynamic ports through URI templates.

Bite [7] is a language for orchestrating REST services, using some of the constructs known from BPEL. With its scripting approach it does not require typing of variables nor the explicit definition of variables. However, the concept of dynamic ports as proposed in this paper is not present in this language. It still relies on correlation mechanisms similar to BPEL's.

Several process engines directly executing Petri nets have already been proposed [23, 26]. Further engines use (colored) Petri nets as process definition language but translate the definition into an internal representation [13, 2]. Our approach is different as it concentrates on RESTful communication with the environment, therefore allowing seamless integration into the World Wide Web.

6 Conclusion

This paper has discussed RESTful process execution on the basis of a special class of Petri nets. The main concepts of REST were introduced and related to the formal model. These include considering intentions on the protocol level and unique identification of resources. RESTful process execution as presented in this paper can be integrated with SOAP-based services. Before invoking such a service XML tokens would be wrapped into SOAP envelopes. In order to allow

¹ See <http://oryx-editor.org> for a running installation

SOAP-based invocations by service requesters, a static port would be offered and the XML payload extracted from the SOAP message.

We have implemented a process engine that executes service nets. The engine is available under MIT license and a running installation including the example from section 2 can be accessed from the engine's homepage <http://code.google.com/p/pnengine/>.

Future work includes the introduction of further process execution aspects into service nets. As a major point, authorization needs to be considered, where static and dynamic ports are only accessible for certain users, e.g. only for those that were involved in the previous process steps. This requires extending guard conditions by the capability to refer to the requesting user. Other work centers around efficient execution of Petri nets. As BPMN serves as primary modeling language, the introduction of certain high-level Petri net constructs such as reset arcs and inhibitor arcs promises simplification of the nets and increased execution performance.

References

1. B. Adida and M. Birbeck. RDFa Primer 1.0. Technical report, W3C, 2006. <http://www.w3.org/TR/xhtml1-rdfa-primer/>.
2. L. Aversano, A. Cimitile, P. Gallucci, and M. L. Villani. FlowManager: A Workflow Management System Based on Petri Nets. In *COMPSAC '02: Proceedings of the 26th International Computer Software and Applications Conference on Prolonging Software Life: Development and Redevelopment*, pages 1054–1059, Washington, DC, USA, 2002. IEEE Computer Society.
3. T. Berners-Lee, R. Fielding, and L. Masinter. Uniform Resource Identifiers (URI): Generic Syntax. Technical report, The Internet Engineering Task Force, 1998. <http://www.ietf.org/rfc/rfc2396.txt>.
4. J. Billington, S. Christensen, K. M. van Hee, E. Kindler, O. Kummer, L. Petrucci, R. Post, C. Stehno, and M. Weber. The Petri Net Markup Language: Concepts, Technology, and Tools. In *24th International Conference on the Applications and Theory of Petri Nets (ICATPN)*, LNCS, pages 483–505, Eindhoven, The Netherlands, June 2003.
5. J. M. Boyer. XForms 1.1. Technical report, W3C, November 2007. <http://www.w3.org/TR/xforms11/>.
6. S. Burbeck. The tao of e-business services: The evolution of web applications into service-oriented components with web services. Online document, October 2000. www.ibm.com/developerworks/library/ws-tao/.
7. F. Curbera, M. J. Duftler, R. Khalaf, and D. Lovell. Bite: Workflow composition for the web. In *Proceedings 5th International Conference on Service-oriented Computing*, volume 4749 of *LNCS*, pages 94–106, Vienna, Austria, Sept 2007. Springer.
8. F. Curbera, F. Leymann, T. Storey, D. Ferguson, and S. Weerawarana. *Web Services Platform Architecture: SOAP, WSDL, WS-Policy, WS-Addressing, WS-BPEL, WS-Reliable Messaging and More*. Prentice Hall PTR, 2005.
9. R. M. Dijkman, M. Dumas, and C. Ouyang. Semantics and Analysis of Business Process Models in BPMN. *Information and Software Technology (IST)*, 2008.
10. D. C. Fallside and P. Walmsley. Web Services Business Process Execution Language Version 2.0. Technical report, Oct 2005. <http://www.oasis-open.org/apps/org/workgroup/wsbpel/>.

11. R. Fielding. *Architectural Styles and the Design of Network-based Software Architectures*. PhD thesis, University of California, Irvine, 2000.
12. R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. Hypertext Transfer Protocol – HTTP/1.1. Technical report, The Internet Engineering Task Force, 1999. <http://www.ietf.org/rfc/rfc2616>.
13. Z. Guan, F. Hernandez, P. Bangalore, J. Gray, A. Skjellum, V. Velusamy, and Y. Liu. Grid-Flow: a Grid-enabled scientific workflow system with a Petri-net-based interface. *Concurr. Comput. : Pract. Exper.*, 18(10):1115–1140, 2006.
14. K. Jensen. *Coloured Petri Nets: Basic Concepts, Analysis Methods and Practical Use. Volume 1*. Springer, 1996.
15. R. Khare and T. Çelik. Microformats: a Pragmatic Path to the Semantic Web. In *Proceedings of the 15th International World Wide Web Conference*, 2006.
16. A. Knopfel, B. Grone, and P. Tabeling. *Fundamental Modeling Concepts: Effective Communication of IT Systems*. Wiley, May 2006.
17. K. Lenz and A. Oberweis. Interorganizational Business Process Management with XML Nets. In *Petri Net Technology for Communication-Based Systems, Advances in Petri Nets*, volume 2472 of *LNCS*, pages 243–263. Springer-Verlag, 2003.
18. P. Massuthe, W. Reisig, and K. Schmidt. An Operating Guideline Approach to the SOA. *Annals of Mathematics, Computing & Teleinformatics*, 1(3):35–43, 2005.
19. R. Milner, J. Parrow, and D. Walker. A Calculus of Mobile Processes. *Information and Computation*, 100:1–40, 1992.
20. H. Overdick. The resource-oriented architecture. In *2007 IEEE Congress on Services (Services 2007)*, pages 340–347, 2007.
21. H. Overdick. Towards Resource-Oriented BPEL. In *Proceedings of 2nd Workshop on Emerging Web Services Technology in Halle (Saale), German*, 2007.
22. P. Prescod. Roots of the REST/SOAP Debate. In *Proceedings of the Extreme Markup Languages 2002 Conference*, Montréal, Quebec, Canada, August 2002.
23. M. Purvis and S. Lemalu. An adaptive distributed workflow system framework. In *APSEC '00: Proceedings of the Seventh Asia-Pacific Software Engineering Conference*, page 311, Washington, DC, USA, 2000. IEEE Computer Society.
24. W. Reisig. *Petri nets*. Springer Verlag, 1985.
25. K. Schmidt. Controllability of Open Workflow Nets. In *Enterprise Modelling and Information Systems Architectures*, volume P-75 of *Lecture Notes in Informatics (LNI)*, pages 236–249, Bonn, 2005.
26. H. M. W. E. Verbeek, A. Hirnschall, and W. M. P. van der Aalst. XRL/Flower: Supporting Inter-organizational Workflows Using XML/Petri-Net Technology. In *Proceedings Web Services, E-Business, and the Semantic Web (WES 2002)*, volume 2512 of *LNCS*, pages 93–108, Toronto, Canada, May 2002. Springer.
27. E. Wilde. What are you talking about? In *2007 IEEE International Conference on Services Computing (SCC 2007)*, Salt Lake City, Utah, USA, July 2007.
28. M. Wolf. Synchronone und asynchrone Kommunikation in offenen Workflownetzen. Studienarbeit, Humboldt-Universität zu Berlin, May 2007.
29. M. zur Muehlen, J. V. Nickerson, and K. D. Swenson. Developing web services choreography standards: the case of REST vs. SOAP. *Decis. Support Syst.*, 40(1):9–29, 2005.