Steffen Preißler, Dirk Habich, Wolfgang Lehner

Standing Processes in Service-Oriented Environments

Erstveröffentlichung in / First published in:

Congress on Services - I. Los Angeles, 06.07-10.07.2009. IEEE, S. 115-122. ISBN 978-0-7695-3708-5

DOI: https://doi.org/10.1109/SERVICES-I.2009.102

Final edited form was published in "Congress on Services - I. Los Angeles 2009", S. 115-122, ISBN 978-0-7695-3708-5
https://doi.org/10.1109/SERVICES-I.2009.102

Standing Processes in Service-Oriented Environments

Steffen Preißler, Dirk Habich, Wolfgang Lehner
Technische Universität Dresden
Database Technology Group
01062 Dresden, Germany

***Abstract*—Current realization techniques for service-oriented architectures (SOA) and business process management (BPM) cannot be efficiently applied to any kind of application scenario. For example, an important requirement in the finance sector is the continuous evaluation of stock prices to automatically trigger business processes—e.g. the buying or selling of stocks—with regard to several strategies. In this paper, we address the continuous evaluation of message streams within BPM to establish a common environment for stream-based message processing and traditional business processes. In detail, we propose the notion of standing processes [1] as (i) a process-centric concept for the interpretation of message streams, and (ii) a trigger element for subsequent business processes. The demonstration system focuses on the execution of standing processes and the smooth interaction with the traditional business process environment.**

I. Introduction

The ever increasing networking in today's society clears the way for the provision of desired information to business operators or individual users in near real-time. In general, there is a constant propagation of information—also called data—from a variety of distributed sources, which are normally offered and encapsulated as services (e.g., *status of payment/delivery*, *sensor/monitoring data*, *stock market data*, etc.). Based on such continuous data, operators can stay informed about changes to their individual business and can also react to occurring events immediately with the execution of application-specific business processes (e.g., *buying or selling stocks*).

Basically, two different and independent technologies have emerged to deal with business processes and event / continuous data processing. *First*, the concept of service-oriented architectures (SOA) introduced an abstraction of functional components that can be used independently and in a loosely coupled way. Since SOA is just an abstract paradigm, Web services [1] have been established as the standard solution for implementing a SOA in industry and science. On top of SOA, Business Process Management (BPM) has received a lot of attention in the last several years. BPM in a SOA describes tools and techniques to orchestrate SOA components to so-called business processes. One de-facto standard is the Web Service Business Process Execution Language (WSBPEL or simply BPEL [2]). It provides a comprehensive syntax system for the description of workflow logic and offers a number of predefined activities to realize fully-fledged business processes.

[1]The project was funded by means of the German Federal Ministry of Economy and Technology under the promotional reference "'01MQ07012"'. The authors take responsibility for the contents.

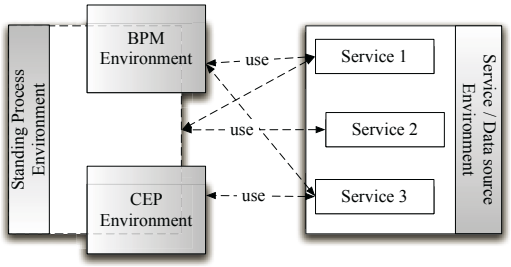

Figure 1. Overall Standing Process Environment.

Second, the area of event processing originated in data stream processing and event pattern detection within data streams is called Complex Event Processing (CEP). Approaches and tools for CEP are orthogonal to the BPM layer [3] since the concept of BPM is based on the strict request-response paradigm and a step-wise, control flow-oriented execution of workflow logic. This, e.g. implies, that in BPM, an input message triggers the explicit generation and execution of a new process instance. For n incoming messages, n process instances are created. Another reason for the architectural separation of BPM and CEP is that service invocations with continuous data are not supported inherently by the request-response paradigm in SOA. Instead, event/message processing, as it is the goal of CEP, requires a stream-based, data-driven handling of data.

To bridge the gap between BPM and CEP, effort has been put into a better interaction of both areas, but currently, these approaches still consider BPM and CEP as mostly independent [3]. However, from an architectural point of view, it would be beneficial to treat BPM and CEP with the same technologies and concepts. In this case, a corporate IT infrastructure could be used to build application scenarios of both areas with common tools. Furthermore, both kinds of technologies could be implemented using a common environment resulting in a seamless integration with no infrastructural hurdles.

Figure 1 depicts a high-level architecture of our proposed approach. We introduce the notion of *standing processes*

Final edited form was published in "Congress on Services - I. Los Angeles 2009", S. 115-122, ISBN 978-0-7695-3708-5
https://doi.org/10.1109/SERVICES-I.2009.102

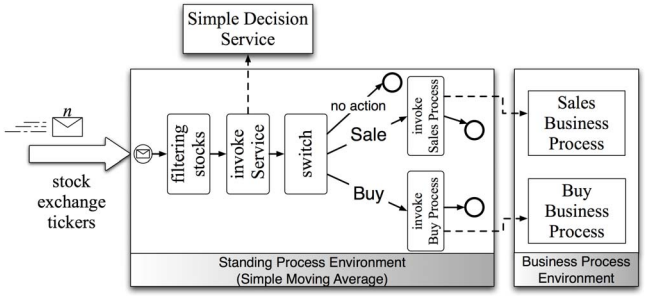

Figure 2. Scenario 1.

that integrates both areas in one common environment by adjusting the execution semantic and slightly extending the modeling semantic of BPM to empower business modelers to integrate stream-based processing of events/continuous data in a process centric way. As a building block, we enhance Web service invocation to handle continuous data allowing us to flexibly extend the functionality of stream-based message processing in standing processes.

Our Contribution and Outline: In this paper, we introduce the concept of stream-based processing of continuous data in service-oriented environments. We start our explanations with an introduction to a application scenario in Section II. This application scenario is the central guideline in this paper. Subsequently, we introduce our developed concept of stream-based Web service invocation in Section III. This stream-based Web service invocation is a one building block for our concept of standing processes, which is explained in Section IV afterwards. Moreover, we describe (1) details of our prototypical implementation and (2) our demonstration system (Section V). In general, the demonstration combines standing processes and traditional business processes. This combination is essential to demonstrate (1) the integrated approach, (2) the seamless interaction between our concept and the traditional environment, and (3) the possibilities of our developed concepts. Finally, we review briefly some related work (Section VI) and conclude our paper (Section VII).

II. Application Scenario

In this section, we provide a detailed introduction to our considered finance application scenario. In this domain, we monitor stock prices with the intention to (semi-) automatically trigger traditional business processes, like the buying or selling of stocks. The monitoring should be done using our concept of standing processes. Therefore, we focus on the description of two realistic examples of standing processes in this section.

Figure 2 represents a first simple standing process with a connection to the corresponding traditional business processes—*the buying and selling of stocks*. The starting point is a *stock exchange ticker* data source, which continuously propagates stock exchange messages to subscribers. In our case, the illustrated standing process is such a subscriber. The process receives many of these messages depending on the granularity of the subscription. Whenever a new message arrives that does not contain any stock of interest for further processing, the message is filtered. In our example, only messages from *XYZ Inc.* are interesting. Then, the selected stock messages (stock prices) are evaluated over a time window—usually 200 trading days—using a service called (*Simple Decision Service*) that computes *simple moving average* values internally. Based on these moving averages and the actual stock prices, the following simple strategies are used as common standard:

Buying Strategy:
Stocks are to be bought as soon as the stock's current price intersects the moving average from below.

Selling Strategy:
Stocks are to be sold as soon as the stock's current price intersects the moving average from above.

These strategies are illustrated with a small example in Figure 3, which depicts a stock price curve and the corresponding 200-day moving average curve. Furthermore, important time points regarding the previously introduced strategies are highlighted. For every incoming message, the *Simple Decision Service* returns one of the three possible values: *no action*, *buy* or *sell*. These response messages are evaluated by the subsequent *switch* operator in our illustrated standing process, and the corresponding actions are triggered.

As we can see in Figure 3, the strategies based on *simple moving averages* are not very sophisticated and many signals are wrong. Therefore, more sophisticated methods have been proposed in this area. For example, one technique is to use *crossing moving averages*. In this case, a long-term (e.g., 200 days) and a short-term (e.g., 20 days) moving average are considered and the strategies are as follows:

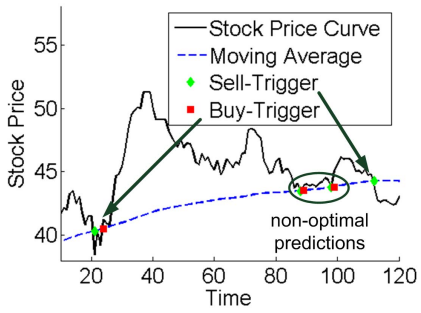

Figure 3. Example for Scenario 1.

Final edited form was published in "Congress on Services - I. Los Angeles 2009", S. 115-122, ISBN 978-0-7695-3708-5
https://doi.org/10.1109/SERVICES-I.2009.102

Buying Strategy:
Stocks are to be bought as soon as the short-term moving average curve intersects the long-term moving average from below.

Selling Strategy:
Stocks are to be sold as soon as the short-term moving average curve intersects the long-term moving average from above.

The corresponding enhanced standing process is shown in Figure 4. In this process description, several services are combined to reach the aim. The long-term and short-term averages are computed using two services that return a new message with a new adopted value for each arriving message. Then, the corresponding average values are sent to the *Decision Service*, which is responsible for the evaluation. Again, the response message takes one of the three values: *no action*, *buy* or *sell*. Based on the response, the corresponding actions are triggered.

The presented techniques are only based on the stock exchange data. However, more sophisticated methods include information on the US Dollar, Euro, oil prices, etc. All of these information sources also represent continuous data and have to be processed in the same fashion as the stock data. Furthermore, there are a lot of domains where continuous data have to be efficiently processed using a process-centric concept like the one we focus on.

III. Stream-Based Service Invocation

Generally, the foundation of SOA is the strict *request–response paradigm* [4]. Thereby, services are requested by a client and a response is generated. If a set D of n equally structured data items d_i with $D = (d_1, \ldots, d_i, \ldots, d_n)$ has to be processed by a service in one common context (e.g., aggregation), this set has to be transferred to the service within one request message. The client has to wait for the request to be fully processed by the service before it is responded. Alternatively, if the service implements some kind of session management, it can map different requests (i.e., invocations) containing different data items $d_i \in D$ to one context. On the one hand, these invocations can be single-item requests, where one request contains exactly one data item d_i. On the other hand, data items can be pooled into chunks and transferred to the service as chunk-based invocations. Both approaches have their drawbacks, either in performance (single-item invocation) [5] or in single-item response latency (chunk-based invocation) [6].

To efficiently realize our vision of standing processes in a service-oriented environment, an enhanced invocation model is needed; it should naturally provide a common context for all data items a client wishes to process without implementing any kind of session management on the service side (e.g., for services like the *Simple Decision Service* or *Long-term Average Service*). Furthermore, the invocation model has to provide low latency regarding single response items that allows the standing processes to use fine-grained information for their stream-based message processing. Therefore, we use this section to propose our developed invocation model that meets the stated requirements and that preserves the basic *request–response* paradigm. The semantics of request and response are simply adjusted to incorporate stream semantics into the service invocation and its execution. In general, three adjustments have to be made to the traditional invocation procedure and the service execution. An overview is depicted in Figure 5 and a detailed description is given below:

Stream Definition: First, we define a request message R that contains the data set D of size n as *input stream* S_I for the service. Furthermore, a response message R' returned by the service that contains the data set D' is defined as *output stream* S_O. Since the input stream is limited to the size of D, a stream is only established for the time during which D is transmitted. Hence, it represents a request from a traditional point of view. This can be modified to a more general concept, since $|D|$ is theoretically not limited in the number of incoming messages in our application scenario.

Every request R_j with one separate D_j creates a new input stream $S_{I,j}$ and, depending on the service's functionality, an output stream $S_{O,j}$ (see Figure 5). This implies that every pair of $\{S_{I,j}$ and $S_{O,j}\}$ belongs to exactly one client request R_j and one service instance W_j processing R_j. Now, we have two streams that preserve a common context for all data items in D.

The streams are structured according to the SOAP specification. Remember, a SOAP message consists of a header

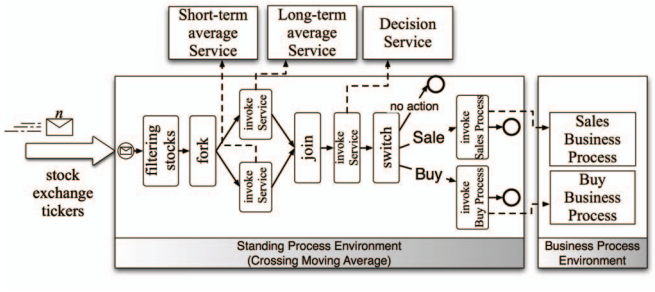

Figure 4. Scenario 2.

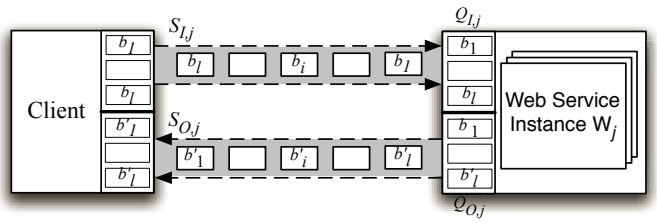

Figure 5. Stream-Based Invocation Model.

Final edited form was published in "Congress on Services - I. Los Angeles 2009", S. 115-122, ISBN 978-0-7695-3708-5
https://doi.org/10.1109/SERVICES-I.2009.102

containing meta data and a body containing the user data. So, in our approach, the header comprises meta data about the stream itself, and the body contains the data set D, which forms the actual stream. To establish the input stream S_I, the SOAP header is sent to the service. This establishes a low-level channel using HTTP and TCP/IP, which remains open for the time during which D is transmitted. After S_I has been opened, the starting SOAP `body` tag is transfered.

Now, the client can send all d_i in D with the help of l *processing buckets* b, where one processing bucket describes the granularity the service can process in one step (see *Bucket Definition*). These buckets are actually pure XML snippets with a predefined structure that embraces a defined number of data items of data set D. Note that the buckets are transmitted on a lower XML hierarchy level than the already transferred `body` tag. When the client has finished sending D, it transmits the closing SOAP `body` tag (which resides on the same hierarchical level as the starting `body` tag) and the closing SOAP `envelope` tag. This triggers the disconnection of S_I. The advantage of sending the corresponding `body` and `envelope` tags as stream delimiters is that the whole stream itself forms one large SOAP message. Therefore, it can be read by a traditional service that buffers the whole XML stream and processes the full DOM tree at once. (see Figure 6).

The output stream S_O from the service to the client is established when the first response bucket b'_1 has to be sent. The establishment of output stream S_O works in the same fashion as that of input stream S_I. Since one processing bucket b_i can be processed by the service in one step, the response bucket b'_i can be streamed back immediately after the successful processing of b_i, even if further buckets are still to be received. S_O is closed when the last bucket b'_l, the closing `body` tag, and the closing `envelope` tag are sent.

Bucket Definition: Remember, data set D represents an array of equally structured data items d_i. To process this set in an efficient and stream-based fashion, the service has to define the granularity of D that it can process within one step. So, D is logically structured into streamable items or *processing buckets*. We divide data set D into l processing buckets b_i with $B = (b_1, b_2, \ldots, b_l)$. Each bucket itself represents an XML element that embraces the data items that have to be processed within one step, and it implies that these data items are child nodes of this bucket

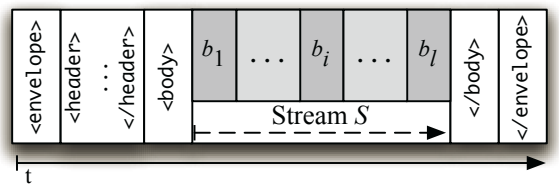

Figure 6. Stream-Based SOAP Message.

element. These buckets are used transparently by the client framework, since they do not contribute to the application logic directly. A process bucket may contain additional meta data expressed as attributes of the bucket element. Currently, every bucket is augmented with an attribute that holds a unique *bucket identifier*, which enables the client to correlate request and response buckets independent from the user data the bucket embraces. Obviously, such meta data can be used further to improve reliability and failure tolerance.

As with traditional service invocation models, the number of data items *in* each *processing bucket* can differ between a request bucket b_i and a response bucket b'_i. Furthermore, the number of response buckets b' can differ for one request bucket b_i. The most common assumption is that every request bucket b_i generates exactly one response bucket b'_i. This describes a traditional $1:1$ relationship. If the service generates a number of response buckets for every request bucket, a $1:N$ relationship is described. An example is a service that returns all invoices of a given customer ID as single response buckets. Both relationships allow to stream back response items even though not all request items have been processed completely.

A different case is the $N:1$ relationship, where many buckets are consumed to generate one response bucket. This scenario can be found in services that implement aggregation functions that, for example, compute the average value of all data items. Since the exact aggregation value is computed upon the arrival of the last bucket b_l (or the last data item d_n, respectively), the client has to wait until D has been processed completely. Thus, no response buckets can be streamed back in between. But in certain scenarios, preliminary values may be used already for further processing or for the definition of stop criteria. To circumvent the mentioned drawback, the application logic on the service side can be extended to put preliminary values into the output stream S_O. Thereby, the type of a value (i.e., preliminary or final) can be flagged with the help of an additional attribute, which have to be added to the response item's XML structure within the service description.

The description of processing buckets, and thus, the issue of how to structure the application data into stream items, is predetermined by the service. Since the service provides the implementation to process the buckets, its description must reside within the service's WSDL document, so that the client can transmit the given data set accordingly. In addition, the WSDL must indicate the capability of the stream-based service. This is briefly described in Section V.

XML Processing Model: The last modification to advance services with stream semantics is to adjust the execution environment for the application logic. Traditionally, one service instance W_j, which is instantiated by one service request R_j, can randomly access the whole data set D_j of its request. W_j builds D'_j as the content for R'_j until D_j has been processed completely. To incorporate stream-based data

Final edited form was published in "Congress on Services - I. Los Angeles 2009", S. 115-122, ISBN 978-0-7695-3708-5
https://doi.org/10.1109/SERVICES-I.2009.102

processing into the service execution, the service instance must be aware of an input queue $Q_{I,j}$, where items $d_{i,j}$ arrive, and an output queue $Q_{O,j}$, where response items $d'_{i,j}$ have to be delivered to. Both queues work concurrently, i.e., the instance can already deliver response items while still receiving request items.

IV. Process Model

The previously presented stream-based invocation model is only one step in the right direction to realize our vision. Next, we introduce our concept of standing processes that takes the paradigm of BPM and the core activities of the traditional workflow approach (BPEL) and advances its execution semantics to handle stream of incoming messages from different sources in a stream-based fashion.

State-of-the-Art Process Execution: Traditional workflow engines execute processes in an *instance-based execution model*, i.e., one incoming message is executed in one process instance in a step-wise fashion [7]. In this control-flow-oriented model, one instance is typically executed in a single-threaded approach. This implies that a stream of n incoming messages executes n process instances. This methodology fits very well in traditional business process scenarios but fails when applied to our scenario, where a standing process instance has to handle n incoming messages in one common context.

Stream-Based Process Model: Compared to the traditional *instance-based execution model*, the fundamental difference of our approach is that n messages generate only *one* process instance $P_{standing}$. This conceptually provides one context for all incoming messages on the process side. Of course, the execution semantics of all activities in $P_{standing}$ have to be modified to allow them to process all messages consecutively. This is done by replacing the *instance-based execution model* with the *pipes-and-filters execution model*, where every modeled activity represents a separate operator that is conceptually executed in a single thread, and each flow edge between two consecutive activities contains a message queue. Thereby, this execution model also replaces the control-flow-oriented execution with a data-flow-oriented execution.

Generally, in our extended execution environment, each activity A_i references an *input queue* Q_I and an *output queue* Q_O. The output queue Q_O of the activity A_i corresponds to the input queue Q_I of the consecutive activity A_{i+1}. When a message enters $P_{standing}$, a message context M is created and the message payload, along with other meta data (such as a *message context id*, a time stamp, and meta data from the SOAP header, e.g., *replyTo*), is placed inside M. In a next step, M is placed in the input queue Q_I of the first activity A_1.

Each activity in $P_{standing}$ consumes a message context if at least one is available in the input queue. Furthermore, it processes the message context M corresponding to the activity's type and outputs the modified message context M' to the referenced output queue. If no M is available in the input queue, the activity blocks until a new message context is available. If the last message context is consumed by an activity A_i, and if the input queue has been closed by a preceding activity A_{i-1}, activity A_i closes its output queue accordingly and terminates.

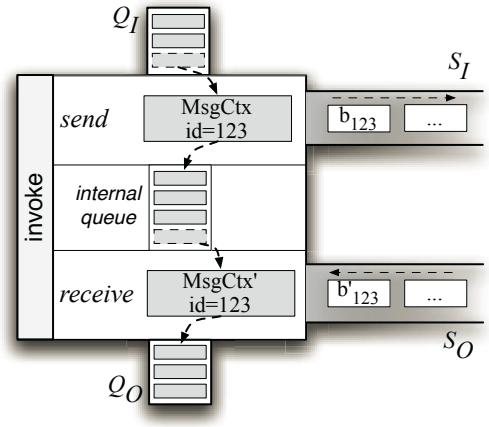

Figure 7. Modified *Invoke* Operator.

While most activities can be transferred to stream-processing activities without further effort, the straightforward modification of the *invoke* activity would lead to single service calls for every message context that is processed. This traditional *invoke* activity can be used to realize the seamless integration of our standing process environment with services and processes in the traditional business process environment (e.g., *Buy Business Process*, Figure 2). We rename this invocation activity to *business invoke* (short: *binvoke*) in order to denote its semantics explicitly. Nevertheless, the drawbacks stated in Section III still occur.

Since we want to efficiently extend the stream-based functionality of our standing processes, we modify the traditional *invoke* activity to use services that are implemented with our stream-based service invocation approach. Figure 7 depicts this new activity in detail. Internally, the modified *invoke* operation is divided into two components that are executed concurrently. The *send* component takes a message context M out of the input queue and places the corresponding application data into one *processing bucket* b (see Section III). It sets the *bucket id* to its *message context id*. After the bucket has been sent, M is placed in an internal queue that the *receive* component consumes. Since the *bucket id* is not touched by the service implementation, it remains the same for the response bucket b', and it can be correlated to the corresponding *message context*. This correlation is needed if the cardinalities of request buckets and response buckets are different (see Section III). Note that our *invoke* activity

Final edited form was published in "Congress on Services - I. Los Angeles 2009", S. 115-122, ISBN 978-0-7695-3708-5
https://doi.org/10.1109/SERVICES-I.2009.102

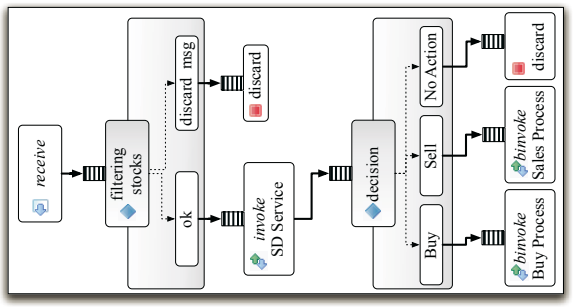

Figure 8. Standing Process of Scenario 1 in BPEL-like Notation.

opens one request stream S_I when the first message context is processed. Accordingly, the response stream S_O from the service is registered. Both streams stay open until the last message has been consumed and the input queue is closed. This provides one execution context on the *service* side for all messages that are processed by $P_{standing}$. For instance, this could be used to implement sliding-window semantics for the time during which the streams are open.

If we apply our modified execution model to Scenario 1 (with a simple moving-average calculation), Figure 8 depicts a modeled standing process with a BPEL-like notation. Thereby, each activity owns an input queue and references an output queue. The process executes as follows: If a message enters the process instance using the *receive* activity, it is first evaluated against the filter criterion. This filter can be traditionally modeled using a switch activity with one decision path (not filtered, *ok*) leading to the consecutive process activities shown in Figure 2 and a second decision path (filtered, *discard msg*) leading to a *terminate* activity, which is modified to discard message contexts instead of terminating the process instance. If the message is not filtered (*ok* path), the payload is prepared and transferred to the *Simple Decision Service* using our stream-based service invocation model and its processing buckets.

Remember, this service provides a common context for all consecutive messages that have passed the invoke activity by maintaining separate time windows for all distinct stock symbols that enter the service. For every bucket the service receives, it updates its moving average and returns a response bucket with the current time window information for the stock and a decision to buy, to sell, or not to act. This information is stored in the message context and passed to the second *switch* activity (*decision*). There, the decision is evaluated and a corresponding decision path is chosen: either nothing is done (*No Action*—the message is discarded in the *terminate* activity) or the invocation of one business process is executed (*Buy* or *Sell*), which triggers stock purchase or stock disposal. These traditional business processes are invoked using the *business invoke* activity.

Within the traditional business environment, processes are exposed as single services and they are executed upon explicit requests. A standing process instance can be provided with message streams in two ways. First, it can subscribe to producing data sources like RSS feeds or other event-emitting information sources (e.g., using WS-Notification). Second, the standing process instance can be explicitly called by a stream-based invocation-enabled client that attempts to process a given data set in a stream-based fashion. In fact, a standing process can expose itself as an extended service, as we described in Section III.

If the process becomes more complex, as depicted in Figure 4, the currently available activities derived from BPEL are no more sufficient, since these activities build on the instance-based execution model and its control-flow-centric execution. Therefore, additional activities must be introduced that cover the data flow aspect more explicitly. To realize Scenario 2, we have to introduce a *copy* activity and a *merge* activity. The *copy* activity creates copies of incoming message contexts and places them into every outgoing queue. Copied message contexts retain their *message context ids*. To join message contexts, the *merge* activity merges message contexts with the same *message context id* based on a data structure mapping to be specified by the process modeler.

V. Implementation and Demo Details

In this section, we briefly describe implementation details of our proposed invocation model and the engine to execute standing processes. Moreover, we give an introduction to our demonstration system, where the two illustrated scenarios from Section II are available.

Stream-Based Web Services Implementation: The implementation of our stream-based Web service invocation model presented in Section III is realized using the AXIS2 SOAP engine (http://ws.apache.org/axis2/—Java environment). This implementation allows to run both traditional services and our stream-based services in one single service engine. The core architecture of this AXIS2 extension, with its basic components, is shown in Figure 9, where the modified components are highlighted in gray.

As depicted, the *transport layer* is modified to receive the message header and to instantly push it to the message receiver without waiting for the whole message to arrive. The *message receiver*, as the central component for message processing, instantiates an input queue and triggers the application logic. If the application has to return data, an output queue is created accordingly. and the application logic has references to the input queue instance and the output queue instance. The application logic uses a cursor concept to access all processing buckets in the input queue within a loop. If the input queue is empty but the input stream is still open, the queue blocks until a new processing bucket arrives. Within the loop, the request buckets with its data items are processed and response buckets are created and placed in the output queue. When the first response bucket is inserted into

Final edited form was published in "Congress on Services - I. Los Angeles 2009", S. 115-122, ISBN 978-0-7695-3708-5
https://doi.org/10.1109/SERVICES-I.2009.102

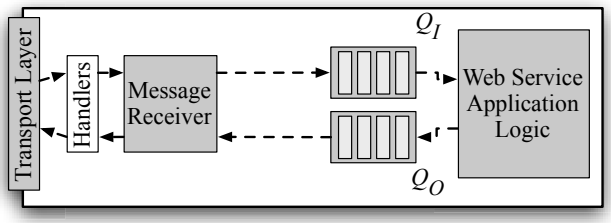

Figure 9. Extended AXIS2 Architecture.

the output queue, the *message receiver* triggers the output stream establishment, and the response message header is sent to its destination. Moreover, the *transport layer* is modified to ensure that the output connection remains open until the last bucket has been sent.

To access stream-based services successfully, the structure and size of processing buckets have to be visible in the service description. Since data set D is an array of equally structured data items d_i, the XML Schema definition describes an unbounded set of complex elements that contain the structure of every data item. To define processing buckets, we add an attribute `processingBucket` to the element that forms the envelope for one data item. Additionally, an attribute `maxsize` prescribes the maximum number of data items that can be bundled as one processing bucket. Figure 10 depicts the WSDL extension for a data structure for request items, where processing buckets are defined. The processing bucket definition for response items is not shown but it is labeled in the same fashion as the request items. Furthermore, operations that support stream-based data processing are flagged by a new attribute `streaming`.

Standing Process Implementation: The implementation of our process engine consists of two parts. The first part is the modeling component that enables a standing process designer to graphically model processes in a standardized fashion. The editor is based on the Eclipse GMF technology (http://www.eclipse.org/modeling/gmf). Furthermore, this graphical representation can be used at runtime to

```
<xs:element name="op1">                          data sets XML Schema structure
  <xs:complexType>
    <xs:sequence>
      <xs:element maxOccurs="unbounded" ...>
        <xs:complexType name="..."
          nstud:processingBucket="true" nstud:maxsize="2">
processing
 bucket
            <!--structure of all data items-->
        </xs:complexType>
      </xs:element>
    </xs:sequence>
  </xs:complexType>
</xs:element>
```

Figure 10. WSDL Extension within the Data Set's XML Schema Structure.

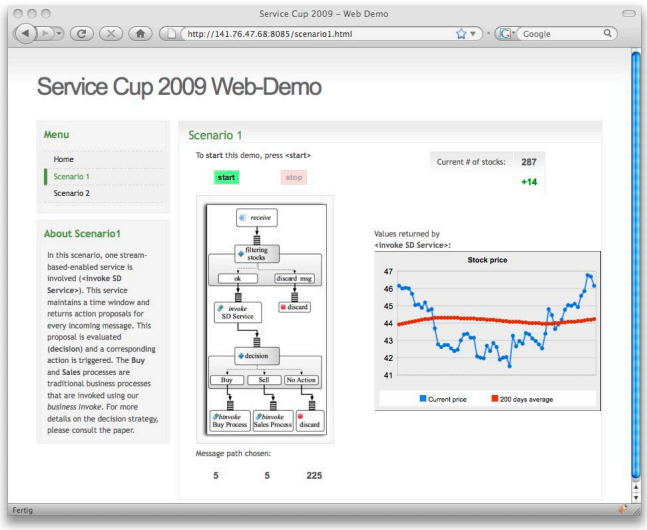

Figure 11. Demo Website Screenshot.

show statistical information about the queue utilization and the operator throughput.

The second part is a standing-process engine that is realized in Java. The operators, as the central execution logic, are executed in separate threads, which automatically leads to performance benefits on multi-core systems. Moreover, each operator provides a monitoring API that allows different applications to gather runtime information and to process them accordingly.

A. Demo Details

Since our standing-process modeling component is based on Eclipse GMF, it is currently not possible to present this component as a Web-based application. Therefore, we implemented a demo front-end that allows the user to execute both scenarios presented in Section II as standing processes. The **-Average* and **-Decision* services are implemented using our modified AXIS2 framework, which resides in a Tomcat server. Our standing processes are deployed in our standing-process engine and can be executed by the Web-based demo front-end. Additionally, all invoke activities are monitored using a PHP Website that fetches the actual values. All traditional processes that we call from our standing processes are modeled in the open-source Eclipse BPEL designer and executed in the open-source BPEL engine Apache ODE.

Figure 11 depicts the demo website at http://141.76.47.68:8085/ (no login required). On the left, there is a menu where the interested user can switch between the two scenarios discussed in this paper. Basically, for every scenario, an image of the standing process with a BPEL-like notation is depicted. Above the process picture, a *start* and a *stop* button are placed, which start and stop the standing process,

Final edited form was published in "Congress on Services - I. Los Angeles 2009", S. 115-122, ISBN 978-0-7695-3708-5
https://doi.org/10.1109/SERVICES-I.2009.102

respectively. In the right top corner, the amount of stocks is displayed that the visitor virtually owns. On the right hand side, the values of the moving-average stock prices are displayed, which are retrieved from the running invoke activities. Below the process image, the *numbers of processed stock ticker messages are displayed,* separated by the decision path they took in the last switch activity. For simplicity, no stock messages enter the process that have to be filtered. If the *start* button is pushed, the process is executed and a stock message enters the process every 0.5 seconds. Since the triggered *Buy* and *Sell* processes typically involve human interaction to enter the number of stocks to be bought and sold, respectively, we define a random number between 10 and 20 every time such a process is triggered for our demo purpose.

VI. Related Work

In the area of *workflow-based message processing*, [7] presented an approach to enhance the standard workflow with pipelined process execution. While this approach increases the throughput and can be used to handle requests more efficiently, our own approach additionally aims to provide capabilities for message-stream analysis. Another recent system is presented in [8]. There, incoming messages are queued and then transformed according to specified rules before they are then re-queued in other (internal or external) queues. In the area of middleware systems and enterprise service buses, many commercial systems have emerged that aim to integrate messages between heterogeneous systems efficiently and that provide modeling capabilities to define control flows [9]. However, these systems use the *instance-based execution model* and do therefore not provide any process-driven approach for efficient message processing that includes stream-based message analysis. Furthermore, *Data Stream Management Systems* (DSMS) provide capabilities to query data streams and to provide data stream analysis [10], [11]. However, these systems are based on relational data models that map incoming messages to flat-structured tuples and process these data items accordingly. Furthermore, these descriptive queries do not allow the modeling of analysis streams with process-centric semantics.

VII. Summary & Outlook

In this paper, we addressed the realization of data stream semantics in SOA by i) introducing our concept of stream-based Web service invocation and ii) presenting the notion of standing processes for process-centric, stream-based message processing in BPM. In combination, both approaches enable the flexible creation of message-based stream-analysis processes with different services used as stream operators ”in the small.” Next, we want to extend the notion of standing processes, particularly its activities to support more data-centric functionalities, such as merge, join or sort, and thus, to support more than one input stream in a consistent way. Furthermore, we want to improve the modeling editor to support these advanced activities in a convenient way. Moreover, the modeling editor has to be integrated more tightly with the process engine.

References

[1] T. Erl, *Service-Oriented Architecture (SOA): Concepts, Technology, and Design*. Prentice Hall PTR, 2005.

[2] OASIS, “Web services business process execution language 2.0 (ws-bpel),” 2007, http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=wsbpel. [Online]. Available: http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=wsbpel

[3] C. Zang and Y. Fan, “Complex event processing in enterprise information systems based on RFID,” *Enterp. Inf. Syst.*, 2007.

[4] OASIS, “Soa reference model,” 2006, http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=soa-rm.

[5] A. Gounaris, C. Yfoulis, R. Sakellariou, and M. D. Dikaiakos, “Robust runtime optimization of data transfer in queries over web services,” in *ICDE*, 2008.

[6] U. Srivastava, K. Munagala, J. Widom, and R. Motwani, “Query optimization over web services,” in *VLDB*, 2006.

[7] B. Bioernstad, “A workflow approach to stream processing,” PhD Thesis, ETH Zurich, 2008. [Online]. Available: http://e-collection.ethbib.ethz.ch/view/eth:30739

[8] A. Boehm, E. Marth, and C.-C. Kanne, “The demaq system: declarative development of distributed applications,” in *SIGMOD*, 2008.

[9] IBM, “Ibm websphere message broker,” 2008, http://www-01.ibm.com/software/integration/wbimessagebroker/.

[10] J. Krämer and B. Seeger, “Pipes - a public infrastructure for processing and exploring streams.” in *SIGMOD*, 2004.

[11] S. Schmidt, T. Legler, S. Schär, and W. Lehner, “Robust real-time query processing with qstream.” in *VLDB*, 2005.