Pluggable Framework for Enabling the Execution of Extended BPEL Behavior

Rania Khalaf[1], Dimka Karastoyanova[2], and Frank Leymann[2]

[1] IBM TJ Watson Research Center, 19 Skyline Drive, Hawthorne, NY 10532, USA
rkhalaf@us.ibm.com
[2] University of Stuttgart, Universitätsstr.38,70569 Stuttgart, Germany
{karastoyanova,leymann}@iaas.uni-stuttgart.de

Abstract. Adding runtime support for BPEL extensions typically requires (1) reacting to navigation events from a BPEL engine executing an extended process model and (2) affecting the engine's navigation behavior based on external triggers. This is usually achieved in a proprietary way for each engine and for each extension.

In this paper, we provide a systematic approach to controlling and reacting to process behavior as well as growing the set of supported control points, thus enabling support for multiple application domains in a composable manner in a BPEL engine. The framework presented in this paper (1) enables a BPEL engine to support extensions, even on existing BPEL processes, and (2) allows developers to create pluggable extension implementations that can be reused across multiple BPEL engines. An implementation of the approach is presented and used in three different projects that need widely differing extended BPEL capabilities.

Keywords: Web services, AOP, middleware reuse, business process, BPEL.

1 Introduction

Service Oriented Architectures are enabling the creation of, interaction between, and composition of applications as services on the network. Composition and reusability often go hand in hand: one composes existing services and may reuse not only the services but also the compositions themselves. Composition in SOA is also about composing extra-functional requirements related to these interactions: domain specific extensions, quality of service concerns, middleware capabilities. BPEL [24], the business process language in the Web services stack, has become the standard for composition in SOA. It focuses on specifying process behavior that can (and has been) implemented in a portable manner by several vendors, research groups, and open source consortia. The standard itself stays out of implementation specific issues; however, the BPEL language is extensible. Domain-specific extensions, which add functionality, may be seamlessly added by defining XML attributes and elements that tag onto BPEL constructs. Such extensions, whether directly syntactic or otherwise attached at runtime, complement the language's constructs without changing their existing fundamental behavioral semantics. Examples include adding BPEL-SPE's

D. Kaeli and K. Sachs (Eds.): SPEC Benchmark Workshop 2009, LNCS 5419, pp. 376–387, 2009.

sub-processes [14], BPELJ's Java capabilities [5], BPEL4Chor' choreography connections [11], parameterization in [16] and others. The problem we aim to solve is that of providing a framework for adding behavioral capabilities that (1) enables a BPEL engine to support extensions, possibly on existing BPEL processes, and (2) allows developers to create pluggable extension implementations that are not dependent on the particular (instrumented) engine in use. While it still requires modifying the code of the chosen engine(s), it provides a systematic approach to controlling and reacting to process behavior as well as growing the set of externally accessible control points.

Several projects that add extensions to BPEL do not extend the language directly, but add functionality in an external manner that affects the process at runtime. [28] adds transactional capabilities via policy attachment; [7] enables Aspect Oriented Programming for BPEL; [21] plugs in (chains of) services that can fulfill the need of the process using semantic information; a similar functionality extension is possible with the approach presented in [16, 26]. BPEL Version 1.1 itself described how the Business Activity protocol of WS-Coordination can be used to enact the relationship between parent and child BPEL scopes.

Nearly all such proposals share a common pattern: They need to (1) react to navigation events from the BPEL engine executing process models that make use of extensions and (2) affect navigation based on external triggers. Consider the following examples of the first case: the start of an invoke activity may lead to running a security policy [6], the end of a scope may lead a transaction to commit [28], activity lifecycle events are sent to a visual monitoring tool [16]. For the second case, consider a fault that needs to be thrown to another fragment of a split BPEL scope [19], and the adaptation of portType/operation at runtime by an external entity/tool in [16]. The usual course of action has been to go into the internals of different engines and add such mechanisms: custom events (to which external applications react or that trigger change in the process navigation), invokers, lifecycle monitors [13]. This was done separately and in a way that did not have generalization and reuse as one of its goals.

The extensions we focus on in this paper are behavioral changes that affect existing (or already extended) BPEL navigation, as opposed to core changes that impact data representations, or performance optimizations. The latter changes affect the engine in cross-cutting ways and are therefore excluded from this approach. We strongly believe they are better implemented natively within a process engine. For example, our approach enables adding support to BPEL engines for ACID semantics of BPEL scopes as introduced in BPELJ; however, it would not allow adding support for BPELJ's arbitrary Java expressions.

The approach in this paper is based on experiences from several different, and ongoing projects in our institute alone where we have found overlapping needs for the proposed interaction pattern with a BPEL engine. Some of these are: pluggable transaction models in BPEL scopes [22] using WS-Coordination; arbitrarily splitting BPEL processes into fragments outsourced to external partners while maintaining the overall process behavior [19]; adding aspect-oriented features to BPEL [17].

We propose pulling out the identified common behavior and providing it as a separate component that can interact with a BPEL engine and affect its behavior accordingly. The paper will describe the design of this component and its interaction with a BPEL engine to enable the modular addition of extension behavior to existing BPEL engines in a minimally obtrusive and maximally reusable manner.

The paper goes on to describe additional related work, followed by the event-based approach, a description of an implementation, and summary and future work.

2 Related Work

BPEL extensions have been addressed in different ways. One important modularization paradigm is aspect oriented programming. [7] applies the AOP approach to BPEL for ad-hoc process change support by defining aspects where the advices are BPEL code and the pointcuts are XPath expressions. Their implementation also requires extending the BPEL engine to support lifecycle event interception and process instance navigation control; however, the extensions have not been realized in terms of a generic infrastructure like the one presented in this paper. The AOP approach has been used in other projects to enable dynamic changes in BPEL processes either in terms of implementation or language constructs. For example, in [8] the authors define process aspects for modifying process models during execution, and engine aspects that are pieces of code for dynamically extending the engine with logging, security, tracing or even new language constructs. Since the approach does not provide a common event model it does not foster reusability of the domain-specific extensions among multiple engine implementations.

BPEL has been extended to support performance measurements and monitoring using the AOP approach in [3]. The extensions are aspects (defined as activities in BPEL) that perform logging of performance related parameters per process instance; the process engine has been instrumented to support the execution of these aspects, which are statically weaved on the process model level using a preprocessor.

In [6], the authors use WS-PolicyAttachment and XPath to non-intrusively attach message-level and process-level policies on a BPEL process. It is implemented on Colombo [9] in which policy handlers are triggered by server-internal events, and adds BPEL activity lifecycle handler triggering events. Adaptability of processes as a reaction to changing QoS requirements has also been addressed in [13], with an engine specific support of dynamic weaving of policies. Both approaches focus on particular engine implementations and do not rely on a generic extendable infrastructure or a reusable event model, hence the extensions for the concrete application domains are not reusable across other engine implementations.

3 An Event-Based Solution to Reuse Infrastructure

The approach being presented uses an event-based mechanism that requires instrumenting the BPEL engine one wants to enable for behavioral extensions or modifications. The architecture behind our approach is presented in Figure 1. Note the 'BC' stands for Blocking with Callback as will be detailed in the next section. It supports interaction with and reaction to 'controllers' that handle the particular extensions in the engine's processes. A Generic Controller is provided that is responsible for the events in the generic event model: the glue code ties between the event model in use and the native navigation model and capabilities of the engine., The event generation/consumption capabilities enable the Generic Controller to communicate with

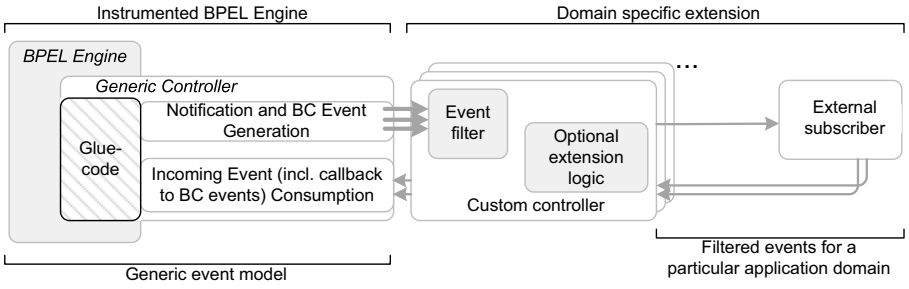

Fig. 1. Approach overview

pluggable Custom Controllers via these events. A Custom Controller does the work related to a specific extension: it filters the events produced by the Generic Controller to only those that are relevant for its specific application domain, and provides the necessary extension behavior. A custom controller optionally interacts with external subscribers involved in implementing the extension logic. The subscribers may not be aware of the event model and use their own transports and message formats to interact, such as SOAP over JMS, direct Java calls, etc. The custom controller then interacts with the subscriber(s) to complete the work over the chosen transport.. For example, an external subscriber used for AOP in [17] is an aspect weaver that interacts using WS-Notification compliant messages; in [19, 25] an external subscriber is a coordinator that interacts over SOAP using WS-Coordination (protocol) messages.

To support an extension, the BPEL engine must be able to (1) produce/consume events needed by extensions in its deployed processes and (2) enable a custom controller to subscribe to some or all of the events produced by the generic controller and send incoming events to it. The custom controller must be able to react to and produce its extension's events and implement the necessary extension behavior.

3.1 Event Types

The engine and a controller need to be able to send and react to events from each other. In order for the controller to react to events from the engine, the engine needs to be able to broadcast 'Notification Events' upon which the controller performs a custom action (produce another event, call a security policy, etc…). In order for an engine to be able to react to events while processing, it needs to be able to accept 'Incoming Events'. Finally, the engine needs to be able to send an event and wait for a callback ('Blocking with Callback' events, or BC events). Each BC event has a specific unblocking Incoming Event that the engine waits for before unblocking the path of the process on which the BC event was hit.

In Section 3.5, we will illustrate the usage of these event types by looking in depth at the <while> BPEL activity. In the next section, we discuss the use of events in the engine and their association with extensions and controllers.

3.2 Events, Extensions, and Controllers

The specific events that an engine needs to support depend on the specific extensions used in each process model. Recall that an extension can be a direct syntactic extension

in the BPEL process itself, or could also be provided externally as part of the deployment information or as an attachment. To enable the engine to recognize an extension, one must associate each extension with the set of its needed events. The association with the list of events enables the engine to determine, for each deployed process, the list of Notification and BC events that custom controllers can subscribe to and a list of incoming event types that it has to accept. Therefore, the system must provide a mechanism (exposed via a graphical tool, configuration file, etc) with which to make this mapping.

A common event model eases the ability support a generic infrastructure for our approach, enabling different engines to be more systematically instrumented. Ideally, the event model would finally be defined in the BPEL specification itself. Based on our experience with BPEL runtimes and applications as well as the requirements of the projects that inspired this work, we defined a BPEL event model [15] derived from the description in the BPEL specification of how a process behaves at runtime. Since all BPEL engines have their own (distinct) internal navigation models, the glue-code needs to map between the internal navigation and the generic event model.

We expect that new kinds of events will be added as more application domains extend the base language behavior. For an instrumented BPEL engine to be reusable, detailed documentation of supported event types and their intended effects is critical. Our event model provides such documentation, creating a common base to which others may add. Adding new events more importantly involves modifying the glue-code in the Generic Controller, which may involve delving deep into the intricacies of the engine's navigation or simply reacting to its visible APIs depending on the complexity of the needed intrusion into the navigation.

3.3 Events in the BPEL Engine

In this section, we first consider the events that the engine must send. These are either Notification Events or BC Events. Notification events are simply sent from the engine to any registered listeners by means of message oriented middleware. After sending the event, the engine continues its regular work. An example is a lifecycle event for a monitoring tool.

On the other hand, BC events block the path in the process of the construct that produced them, causing the engine to wait for a signal from a controller before it can continue. No other paths of the process are affected. BC events are used in scenarios where the engine needs additional knowledge to decide the next navigation steps. A well-known example is the request made to a service bus to interact with a Web service when a BPEL engine hits an <invoke> activity: Navigation cannot continue from that activity until the response message has been passed to the process engine. Another example is the completion of an iteration of a fragment of a loop that has been split across multiple process fragments: The loop fragment cannot iterate again or continue until all other fragments belonging to that loop (in other process fragments) have also completed their iteration. Therefore, the engine blocks that path until the controllers gives it the go-ahead. The logic behind deciding when and how to continue depends on the use case at hand and should be pluggable.

It is important to note that a BC event will only be thrown if a custom controller has subscribed to it, forming a contract to send the callback event allowing the process to eventually unblock the path.

On the other hand, the engine also needs to accept incoming events, such as 'throw fault in scope x', 'compensate scope y' or 'evaluate link z'. These incoming events from the controller always affect navigation. The difference between an incoming event and the callback of a BC event is that the latter is associated with a blocked path. Instead, incoming events can occur while the engine is navigating as normal.

3.4 Interacting with the Engine through the Controller

The interaction with the engine happens through the Generic Controller. However, there may be several controllers subscribing to the same or different events. As noted earlier, custom controller behavior will be different depending on the application domain. The system must provide a custom controller subscription mechanism.

A custom controller subscribes to a set of event types from the engine based on its published list of supported event types. If it subscribes to a BC event it must also provide the corresponding callback event. The controller should also be able to send the engine the incoming events corresponding to the extension. When a process model has been successfully deployed, all its BC events must have a subscribed controller; otherwise, an 'extension not supported' error is thrown.

At most one controller can subscribe to the same BC event from the same process instance. This restriction is placed because, although it can be well-defined, it can cause undesired behavior in the controllers. This includes inconsistent state between the controller and the process because the controller thinks the process is (waiting) but the process has already been unblocked by another, race conditions, etc.

The domain specific behavior of a custom controller is incorporated by plugging it into the Generic Controller. It is very well possible that the controller responsible for returning the callback combines logic from several others. This would result in an n-staged hierarchy between controllers, where n is greater than 1, and the root controller is the one responsible for interacting with the engine for that BC event. In this paper, we focus on adding runtime support for new BPEL extensions with minimal disruption and maximal reuse. Future work aims to investigate handling collaborations between controllers resulting from conflicting extensions on the same process instance.

3.5 Illustrating the Usage of Events Using <while> Loops

We present the while loop event model, shown in Figure 2, to illustrate our approach. First, we discuss the events and their usage. Then, we show some examples of how different custom controllers can use these events. A loop starts like any activity in the 'inactive' state. Once all its incoming links have been evaluated, its join condition is evaluated. If the condition is false, it enters the 'Dead-Path' state and the 'Activity_Dead_Path' notification event is published; otherwise, the 'Activity_Ready' event is published. If there is a controller registered for the 'Activity_Ready_Blocking' Blocking with Callback Event, then that event is published instead and the path is blocked until the 'Start_Activity' incoming event arrives. Once the activity starts, the 'Activity_Executing' notification event is published and the loop condition is ready to be checked.

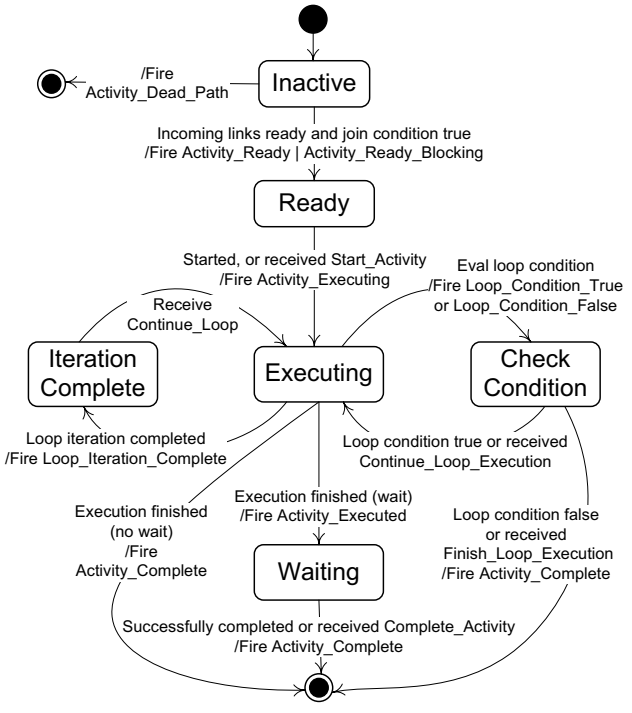

Fig. 2. Loop Event Model, based on a combination of figures 2.1.1 and 2.4.1 in [15]

If a controller is registered for the 'Loop_Condition_True' or the 'Loop_Condition_False' Blocking with Callback Events, the corresponding event is published. The controller then decides whether the execution may continue by sending the Incoming Event 'Continue_Loop_Execution' or 'Activity_Complete'. In the latter case, the activity enters the end state, while in the former case, the body of the loop will be executed and the nested activities enter the state 'Executing'.

Once a loop iteration completes, the 'Loop_Iteration_Complete' Blocking with Callback Event is sent (as always, only if there is a registered controller). The loop iteration gets unblocked by receiving the 'Continue_Loop' event, at which point the loop condition is evaluated again and we continue as before from 'Executing'.

If no controller is registered for the 'Loop_Iteration_Complete' BC event or the 'Loop_Condition_True/False' BC event, then the engine performs the loop's work in the 'Executing' state. Once the loop completes, then if no one is registered for the 'Activity_Executed' BC event, the activity simply fires the 'Activity_Completed' Notification Event and ends. If there is a registered controller for the 'Activity_Executed' event, the engine would wait for its unblocking 'Activity_Complete' Incoming Event to end <while>'s lifecycle.

Having explained the events of a loop, their use in different custom controllers is described next by sketching some potential extensions.

3.5.1 Examples of Loop Extensions and their Custom Controllers

Consider a monitoring extension. It would register for only the Notification Events of the loop, i.e. there is no blocking at all. The controller would get notified of the lifecycle changes in the loop, without affecting the process's runtime behavior.

Assume an extension that will trigger an action every time the loop is about to begin iterating. Then, one would create a custom controller that filters out all events except for the 'Loop_Condition_True' BC event and that knows how to trigger the action. The controller will perform the action every time it receives this event. Once the action completes, the controller will send the 'Continue_Loop' event to unblock the path and allow the iteration to take place.

Now consider a more elaborate example: a business process containing a loop is split among multiple partners, and a loop ends up being fragmented: several of its activities are in partner A and several others are in partner B. The controller needs to know about the lifecycle events, and needs to affect the navigation. It needs to work with a coordinator that ensures that all fragments start, iterate, and end together. To enable this, the fragmented loop custom controller is registered for the following loop lifecycle events: 'Activity_Ready', 'Activity_Dead_Path', 'Loop_Condition_True', 'Loop_Condition_False', and 'Activity_Executed'. The controller sends the coordinator a message when it gets to any of the 'Loop_Condition_True' or 'Loop_Condition_False' or 'Activity_Dead_Path' Events. This means that the particular fragment of the loop has been reached in the navigation in that process.

The other fragments in the loop are in other processes and their controllers also send a message to the coordinator once the loop is reached in the navigation of their processes. The coordinator now has the information it needs to notify the controllers of all the fragments of whether the fragments should go ahead in executing or not. The coordinator sends a message to the controller that the latter maps to the appropriate unblocking event: either the 'Continue_Loop_Execution' or 'Finish_Loop_Execution'. If the loop should iterate, the controller waits for the next 'Loop_Condition_True' or Loop_Condition_False' event and sends a corresponding message to the coordinator to signal the end of the iteration. The coordinator waits to hear from all fragments that they have reached the end of the iteration. It then sends each controller the appropriate unblocking message. The fragments will either iterate or end together as a result. More details on coordinating fragmented loops are in [19]. The details of the implementation's usage of this approach to go from the WS-Coordination protocols to the Event Model and control the engine are in [25].

4 Implementation

An implementation of this approach has been prototyped based on the open source BPEL engine ActiveBPEL [1], the monitoring extensions added in [29], and propagation of process instance lifecycle events to a monitor over JMS [23, 29].

The event model governs the behavior that needs to be common regardless of which BPEL engine is used. The part of the system that has to be created upon enabling a new BPEL engine with this approach is the Generic Controller, including modifications to it and to the engine to enable creating and reacting to the new types of events. The engine must also be modified to support event notification. The

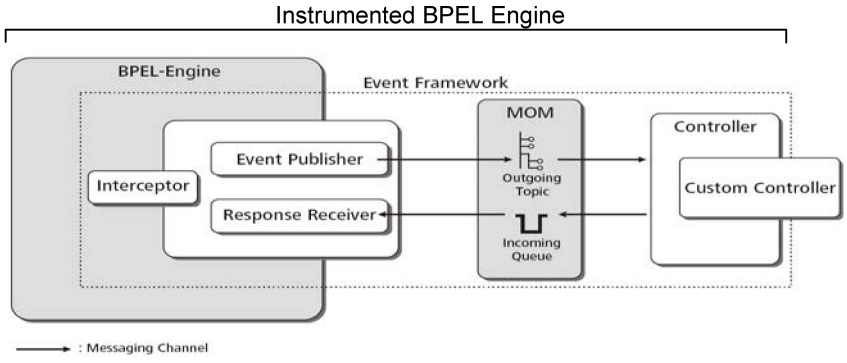

Fig. 3. Implementation in [27]

Generic Controller is the glue code between the particular BPEL engine and our common event model, mapping from calls native to the former into known event types in the latter. The pluggable custom controllers work off of the common event model and are thus able to be independent of the particular BPEL engine.

Our implementation uses ActiveBPEL as BPEL engine. The first step in enabling it for our approach was to instrumented it to support the event model in [15]. For example, it has to be made able to stop where needed when a BC event is created, to be able to override a loop condition due to an event from the controller, and so on.

The system architecture is presented in Figure 3. The event framework contains an interceptor, an event publisher, a response receiver, a message broker and a generic controller. The interceptor detects lifecycle changes from the processor/navigator, performs any necessary mapping to the generic event model, and publishes them on a JMS topic; a pub/sub implementation has been chosen to enable subscription of other components, like monitoring tools and audit logs, to the life cycle events. The JMS implementation used is ActiveMQ [2]. The controller subscribes to the events published on the JMS topic. A custom controller implements filtering capabilities for a particular application domain and sends the filtered set of event types to subscribers either directly or via a pub/sub system (not shown in Figure 3, compare to Figure 1). The events that influence the behavior of the running process instances (Incoming Events and Call Back events) are sent by the controller to the engine via the queue in the event framework. The response receiver in the engine consumes the event notifications and implements additional logic that steers the reaction to these events in the engine according to the event type. The messages that notify the events contain correlation information like the process instance ID, process model ID, the process language construct that caused the event to be thrown, the event type, the engine ID, the custom controller instance ID, and any data specific to the event type (fault name, link condition, etc).

The approach has been used successfully and independently in the three projects described above to test the concepts and their reusability. Detailed results are documented in [22, 25, 27].

The approach itself does not dictate a particular technology for communicating with the custom controllers. In our projects, JMS was an appropriate implementation choice for. several reasons. First, the monitoring project required supporting possibly multiple monitors (subscribed only to notification events) that may be on different machines. Second, the business processes run over a long period of time and thus some quality of service guarantees are desirable (such as unhandled event persistence). Third, some systems may require support for several BPEL extensions handled by different custom controllers that require the same events, in which case a pub/sub system provides much of the necessary machinery to manage that. However, if one anticipates that their controllers will always be co-located with the engine thereby making a more tightly coupled system acceptable, then performance can be improved by using a different interaction mechanism.

The complexity of a system using this approach depends in large part on how complex the glue-code is, and that in turn depends on the particular engine chosen: how easy/hard it is to control and how far its native navigation model is from our event model. The approach itself has similar complexity to other approaches for injecting and controlling BPEL behavior: one must somehow attach the extension, register it, provide logic to perform it, and be able to manipulate the engine. However, implementations custom-built for a particular BPEL engine and/or extension might be able to do better as they can optimize for a particular engine's navigation logic and even provide the extension behavior directly as part of the native engine code-base.

5 Conclusion and Future Work

The framework presented here enables plugging domain specific extensions to existing BPEL engines in a systematic manner. The domain specific extensions can be used even within already deployed BPEL process models, without the need to change the process models. The extensions are based on a generic event model, which can be mapped on life cycle event models of existing systems and realized in terms of custom controllers that use the proposed framework. This renders the approach flexible and modular, while standard-compliance is maintained. These advantages are due to the generic architecture that supports reaction to lifecycle events in a BPEL engine and enables steering the execution of processes though external triggers.

We showed that to support this framework, an engine has to be instrumented to support a set of events needed by its extensions. We refer to a base event model for common ones. A Generic Controller provides the glue between the internal navigation of the engine and event consumption and production. Custom controllers plug into the Generic Controller to provide the logic related to the extension. The system provides a mechanism to map extensions and custom controllers to a set of needed events.

We are investigating how custom controllers can be combined to enable composability of extensions, and how they then affect each other. This is a non-trivial area, related to policy handling and middleware composability, not yet supported in the prototype. The complexity is both in composing the (possibly contradicting) needs of the extensions, their interaction with the engine, and the work of the controller.

Provided that our framework enables dynamic plugging of advanced features to BPEL engines, audit functionality must be supported that captures the execution of the extended functionality as well.

We intend to build further domain specific extensions that use the presented event-based framework and to improve the ones mentioned here. For example, the approach extending BPEL processes with aspects, based on WS-Policy, uses deployment data to state which activities may be blocked to help choose the blocking or non-blocking implementation depending on the existence of a subscription; this feature will be improved in future. We are also investigating tooling for mapping extensions, events and controllers with any instrumented BPEL engine.

Acknowledgement. Ralf Schroth and Michael Paluszek for working on the implementation of this system. Stefan Pottinger for collaboration in the early stages.

References[1,2]

1. ActiveBPEL Engine, `http://www.activebpel.org/docs/architecture.html`
2. Apache Software Foundation. ActiveMQ, `http://activemq.apache.org/`
3. Baresi, L., Guinea, S.: Towards Dynamic Monitoring of WS-BPEL Processes. In: Benatallah, B., Casati, F., Traverso, P. (eds.) ICSOC 2005. LNCS, vol. 3826, pp. 269–282. Springer, Heidelberg (2005)
4. Benatallah, B., Dumas, M., Sheng, Q.Z.: Facilitating the rapid development and scalable orchestration of composite web services, Distributed and Parallel Databases (January 2005)
5. Blow, M., Goland, Y., Kloppmann, M., Leymann, F., Pfau, G., Roller, D., Rowley, M.: BPELJ: BPEL for Java Technology, BEA Systems and IBM Corporation (2004), `http://www.ibm.com/developerworks/library/specification/ws-bpelj/`
6. Charfi, A., Khalaf, R., Mukhi, N.: QoS-aware web service compositions using non-intrusive policy attachment to BPEL. In: Krämer, B.J., Lin, K.-J., Narasimhan, P. (eds.) ICSOC 2007. LNCS, vol. 4749, pp. 582–593. Springer, Heidelberg (2007)
7. Charfi, A., Mezini, M.: Aspect-oriented web service composition with AO4BPEL. In: Zhang, L.-J., Jeckle, M. (eds.) ECOWS 2004. LNCS, vol. 3250, pp. 168–182. Springer, Heidelberg (2004)
8. Courbis, C., Finkelstein, A.: Towards an Aspect Weaving BPEL Engine. In: Proceedings of AOSD Workshop ACP4IS, Lancaster, UK (March 2004)
9. Curbera, F., Duftler, M.J., Khalaf, R., Nagy, W.A., Mukhi, N., Weerawarana, S.: Colombo: Lightweight middleware for service-oriented computing. IBM Systems Journal 44(4), 799–820 (2005)
10. Curbera, F., Duftler, M., Khalaf, R., Mukhi, N., Nagy, W., Weerawarana, S.: BPWS4J. IBM, `http://www.alphaworks.ibm.com/tech/bpws4j`
11. Decker, G., Kopp, O., Leymann, F., Weske, M.: BPEL4Chor: Extending BPEL for Modeling Choreographies. In: Proc. of the IEEE International Conference on Web Services (ICWS), Salt Lake City, Utah, USA, July 2007. IEEE Computer Society, Los Alamitos (2007)

[1] All URLs in references were checked by the authors on July 19, 2007.

[2] University of Stuttgart Diploma theses are available online at http://www.informatik.uni-stuttgart.de/cgi-bin/NCSTRL/NCSTRL_view.pl?id=DIP-XXXX where XXXX is replaced by the thesis number, i.e.: 2444 for [22].

12. Dumas, M., Fjellheim, T., Milliner, S., Vayssiere, J.: Event-based Coordination of Process-oriented Composite Applications. In: Proc. of the 3rd Int'l Conf. on Business Process Management (BPM), Nancy, France, September 2005. Springer, Heidelberg (2005)
13. Erradi, A., Maheshwari, P., Padmanabhuni, S.: Towards a Policy-Driven Framework for Adaptive Web Service Compositions. In: Proc. of NWeSP 2005, Los Alamitos, CA, USA (2005)
14. IBM, SAP AG, WS-BPEL Extension for Sub-processes – BPEL-SPE (September 2005), http://www-128.ibm.com/developerworks/library/specification/ws-bpelsubproc/
15. Karastoyanova, D., Khalaf, R., Schroth, R., Paluszek, M., Leymann, F.: BPEL Event Model. University of Stuttgart, Technical Report Nr. 2006/10 (November 2006), http://www.informatik.uni-stuttgart.de/cgi-bin/NCSTRL/NCSTRL_view.pl?id=TR-2006-10
16. Karastoyanova, D., Leymann, F., Nitzsche, J., Wetzstein, B., Wutke, D.: Parameterized BPEL processes: Concepts and implementation. In: Dustdar, S., Fiadeiro, J.L., Sheth, A.P. (eds.) BPM 2006. LNCS, vol. 4102, pp. 471–476. Springer, Heidelberg (2006)
17. Karastoyanova, D., Leymann, F., Schroth, R., Ortiz, G.: BPEL'n'Aspects: Chiming Into Orchestrations (in submission)
18. Khalaf, R., Leymann, F.: Role Based Decomposition of Business Processes Using BPEL. In: Proc. of the IEEE Int'l Conf. on Web Services (ICWS 2006), Chicago, Il (September 2006)
19. Khalaf, R., Leymann, F.: Coordination Protocols for Split BPEL Loops and Scopes. University of Stuttgart, Technical Report No. 2007/01 (March 2007), http://www.informatik.uni-stuttgart.de/cgi-bin/NCSTRL/NCSTRL_view.pl?id=TR-2007-01
20. Lehmann, T.J., McLaughry, S.W., Wyckoff, P.: T Spaces: The next wave. In: Proc. Hawaii Int'l Conf. on System Sciences HICSS32, Maui, Hawaii (January 1999)
21. Mandell, D., McIlraith, S.: Adapting BPEL4WS for the Semantic Web: The Bottom-Up Approach to Web Service Interoperation. In: Fensel, D., Sycara, K.P., Mylopoulos, J. (eds.) ISWC 2003. LNCS, vol. 2870, pp. 227–241. Springer, Heidelberg (2003)
22. Mietzner, R.: Extraction of WS-Business Activity from BPEL 1.1. Diploma Thesis 2444, University of Stuttgart (2006)
23. Nitzsche, J.: Entwicklung eines Monitoring-Tools zur Unterstützung von parametrisierten Web Service Flows. Diploma Thesis 2388 (2006)
24. OASIS: Web Services Business Process Execution Language Version 2.0 (April 11, 2007), http://docs.oasis-open.org/wsbpel/2.0/wsbpel-v2.0.html
25. Paluszek, M.: Coordinating Distributed Loops and Fault Handling, Transactional Scopes using WS- Coordination protocols layered on WS-BPEL services. Diploma Thesis 2586, University of Stuttgart (2007)
26. Rajasekaran, P., Miller, J., Verma, K., Sheth, A.: Enhancing Web Services Description and Discovery to Facilitate Composition. In: Cardoso, J., Sheth, A.P. (eds.) SWSWPC 2004. LNCS, vol. 3387, pp. 55–68. Springer, Heidelberg (2005)
27. Schroth, R.: Specification, Design and Implementation of a BPEL Engine with AOP Support and an Aspect Weaver for BPEL Processes. Diploma Thesis 2523. University of Stuttgart (2006)
28. Tai, S., Khalaf, R., Mikalsen, T.: Composition of Coordinated Web Services. In: Jacobsen, H.-A. (ed.) Middleware 2004. LNCS, vol. 3231, pp. 294–310. Springer, Heidelberg (2004)
29. Wutke, D.: Erweiterung einer Workflow-Engine zur Unterstützung von parametrisierten Web Service Flows. Diploma Thesis 2401, University of Stuttgart (2006)