Runtime Process Adaptation for BPEL Process Execution Engines

Simon Tragatschnig and Uwe Zdun
Software Architecture Research Group,
University of Vienna,
Austria
{*simon.tragatschnig, uwe.zdun*}@*univie.ac.at*

***Abstract*—Requirements for business processes can change over time. Adapting a process to meet the changed requirements is not always possible, especially for long running processes, where stopping the execution of process instances might be necessary and/or instance migration or compensation scenarios must be implemented. Adaptations for processes can be described in a generic way using adaptation patterns. Interpreting these adaptation patterns will enable adaptation support at runtime, independently from a specific process execution engine. This paper presents a framework which enables adaptation support for process execution engines. It explains how runtime information of process instances can be monitored by using aspect-oriented programming. A model for adaptation patterns is presented as well as an adaptation engine which interprets instances of the adaptation pattern model and applies the adaptations to running BPEL processes and their instances. The presented adaptation framework is not tied to a specific process execution engine, so any process execution engine can be extended to provide adaptation support.**

***Keywords*-Runtime Process Adaptation, Adaptation Model, Process Execution Engine, BPEL**

I. Introduction

Requirements on business processes usually change over time, requiring executable business process models to be changed accordingly. As many business processes are long running entities, usually instances are still running, when a business process model must be changed. Different strategies exist for integrating process model changes into running process instances, such as manually tailoring the process instance or its results to comply to the new process model, or writing a script to automatically apply these changes. However, these strategies and similar strategies are usually difficult, error-prone, and time consuming.

A number of approaches exist that address this problem. A process-aware information system supporting modifications during runtime is presented by Reichert et al. [1]. Even though this approach solves the problem, it does not handle process adaptation in a generic way, as it does not allow to extend existing process execution engines with adaptation support, but is dependent on a proprietary process engine. Some other approaches [2]–[5] support adaptation of BPEL processes by monitoring and selecting or replacing the services triggered by process activities of a running BPEL process (e.g., based on quality of service (QOS) definitions). These approaches use different strategies for adaptation, such as general service proxies performing service adaptation [5], or aspect-oriented engine adaptation plus indirection by an interception and adaptation layer [2]–[4]. Even though these approaches allow for adapting existing process engines, they do not support modifying a process' structure at runtime.

This paper presents a framework and a prototype implementation to support structural modification of processes and their instances at runtime and enables extending existing BPEL process execution engines with adaptation support. This is achieved by providing an abstract process model, an adaptation patterns model, and an adaptation engine – to support process adaptation in a generic way. The adaptation patterns model describes adaptation patterns, based on the abstract process model, and supports specifying adaptations based on established adaptation structures. Instances of the adaptation patterns model are interpreted by the adaptation engine, which applies the adaptations to instances of the abstract process model as well as to any process execution engine on which the processes are executed. To allow for monitoring and later adapting processes in an existing process engine, without having to modify is implementation, aspect-oriented programming (AOP) is used. In this paper, the Apache ODE [6] BPEL process engine is exemplarily extended to provide adaptation support.

Figure 1 gives a simplified overview of our adaptation framework and the extension of Apache ODE. To integrate the adaptation framework with Apache ODE (or any other existing BPEL engine), only a monitor and a set of basic adaptation operations (like inserting or deleting an activity) must be implemented. Aspect-oriented programming is only required, if the engine does not already implement sufficient monitoring capabilities. For instance, in the case of Apache ODE aspect-oriented programming was necessary as Apache ODE does not emit all required events. The monitor observes Apache ODE to synchronize deployed processes and its instances with instances of the abstract process model. Basic adaptation operations for Apache ODE are used to execute the instances of the adaptation patterns model. Analogously, to extend another process execution engine, such as JBoss jBPM, only implementations of the monitor and the basic adaptation operations are needed.

The paper is structured as follows: The next section intro-

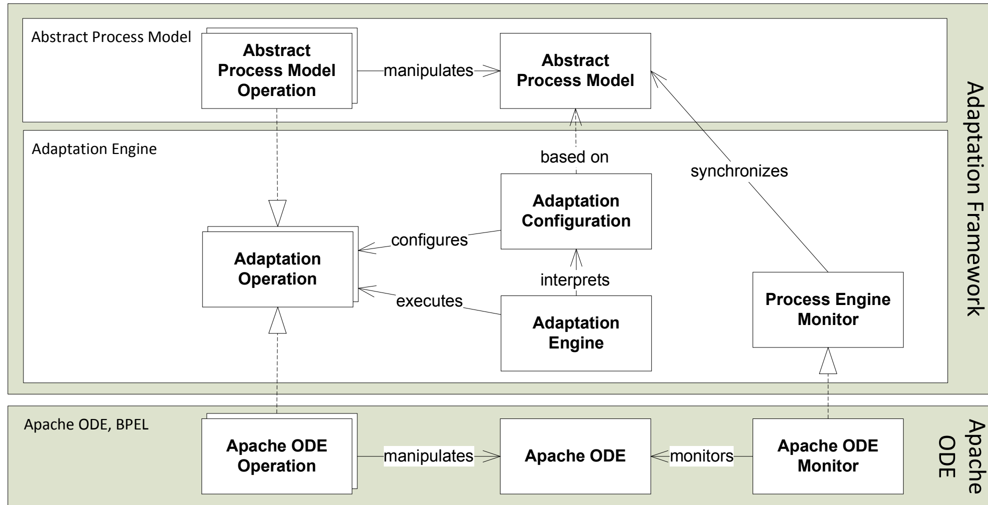

Figure 1. Overview of the adaptation framework extending Apache ODE

duces a motivating example to illustrate the problem further. The monitoring of Apache ODE is presented in Section III (using AOP). Section IV introduces the abstract process model that describes detailed information about processes and instances during runtime and how monitoring is used to synchronize runtime information of the process engine with the adaptation framework. The adaptation patterns model for describing adaptation patterns is presented in Section V and the adaptation engine, which applies adaptations to a process, is introduced in Section VI. This section will also explain how concrete adaptation descriptions are interpreted to apply adaptation to processes and instances within the adaptation framework and Apache ODE. A comparison to the related work is presented in Section VII, and in Section VIII we conclude.

II. Motivating Example

The following example illustrates the problem of changing a process at runtime. The left-hand side of Figure 2 shows a process for registration to a PhD program at a university. An application for admission has to be submitted to the university where it must be examined. The examination will take about 6 weeks. If the examination leads to granting the admission, the student is allowed to register for the PhD program. Consider that 5 weeks after starting the examination of the admission, the process designer realizes that for registering the PhD program a passport photo of the student is needed as well. So a new activity *get_passport_photo* has to be inserted before the activity *register_for_phd_program*. However, at this stage there might already be many running process instances, which are for example in the 6 weeks examination phase. These running process instances neither contain the novel activity nor the process variable needed to store the reference to the passport photo.

Possible approaches for applying the change to running process instances are:

- Stopping all running process instances, changing the process and starting new instances of the changed process. This would cause the university to examine the admission twice and the applying student would have to wait another 6 weeks before being allowed to register for the PhD course, which might lead to problems regarding application deadlines.
- Ignoring the change, which may cause inconsistent data (missing picture) and maybe additional administrative effort to obtain the student's picture. Only new instances will take the new activity into account.
- Instance migration (manually or using scripts) by adding a dummy picture to avoid inconsistent data, but maybe cause additional administrative effort to obtain the student's picture later on.

These options will cause data inconsistencies and/or additional effort for instance migration. Support for applying changes on processes as well as instance migration during runtime is required to solve the problem without manual efforts or inconsistencies. As explained in Section VII, so far no approach exists that supports structural changes to processes and their instances at runtime for existing process engines, such as popular BPEL engines. Our approach achieves this goal by providing a general adaptation framework, based on an abstract process model, an adaptation pattern model, and an adaptation engine, that can be combined with existing engines through monitoring interfaces (possibly realized using AOP) and the implementation of a set of basic adaptation operations.

The outcome is illustrated in Figure 2 from the perspective of the user of our adaptation framework: The user must specify an adaptation pattern instance. First, the adaptation

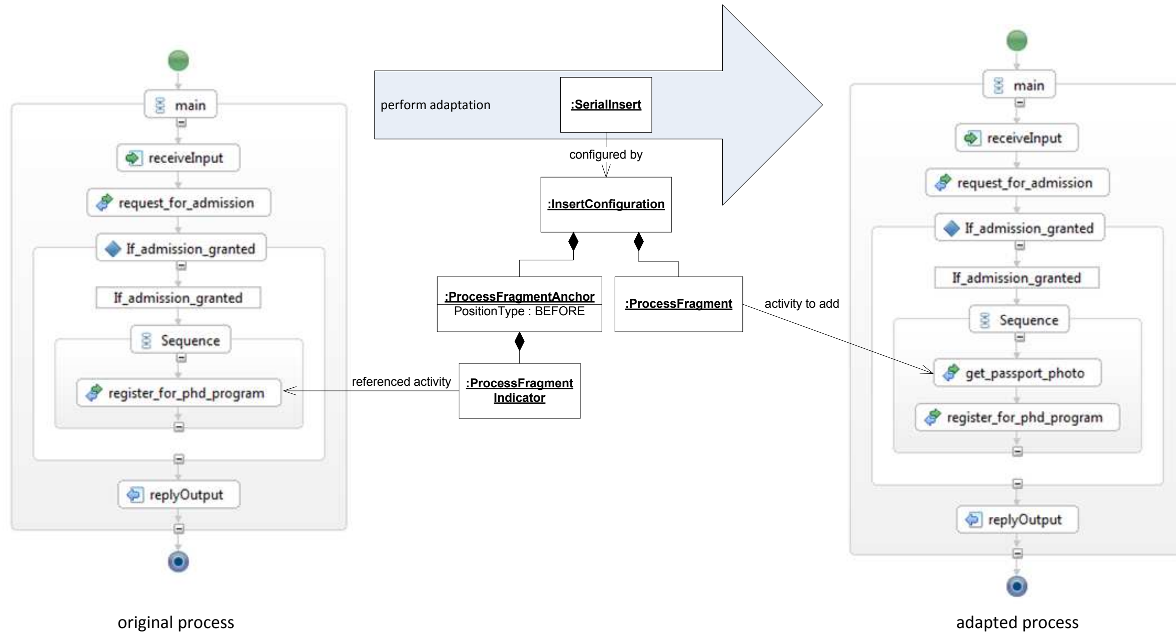

Figure 2. Example: Registration to a PhD program

type is selected (in this case *SerialInsert*). Next, the insert operation is configured by selecting the place in the original process, where the insert should take place, and by defining a process fragment to insert. Upon deployment of this adaptation, our prototype will update the BPEL process description deployed in the engine, as well as all process instances of the BPEL process.

III. Monitoring Apache ODE

The process engine Apache ODE only provides rudimentary monitoring support, e.g., basic information like the name and state of deployed processes and their instances. For supporting adaptation of processes during runtime much more information is needed like the currently executing activities, already processed and future activities, and even information about the process engine's internal execution of an activity.

For the adaptation engine presented in this paper, the monitored information about a process' structure and runtime information of a process' instances is needed. This runtime information includes information about which activities were already processed, which one is actually executed and which one still have to be executed. Detailed information about the internal processing of an activity is extracted, as well.

The goal of our prototype implementation was that the existing source code of Apache ODE should not be affected by the additional monitoring code. Hence, AOP with runtime weaving is used for adding the monitoring functionality. Another advantage of using AOP for monitoring compared to event-based monitoring (as for example used in the Business Process Illustrator [7], [8]) is the direct access to objects within the process execution engine. We can use these monitored objects directly to apply the adaptations (i.e., modifications of a process' structure) on them, as explained in Section VI.

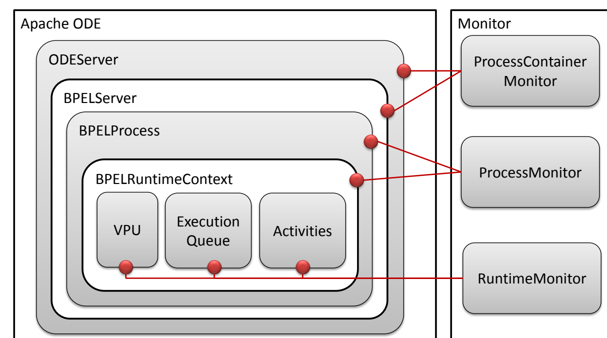

Figure 3. Overview: Monitoring Apache ODE

Figure 3 shows a simplified overview of Apache ODE's structure and which parts have to be monitored to access the information needed. The *ProcessContainerMonitor* monitors the state of the process engine (start, stop, un-/deploy process), which is represented by the implementation of *ODEServer* and *BPELServer*. The *ProcessMonitor* monitors the deployed processes and their instances. The required information is provided by implementations of *BPELProcess* and *BPELRuntimeContext*. The *RuntimeMonitor* monitors the instances of a process and gives information about the state (e.g., active, completed, failure, ...) and which activi-

ties were/are/will be executed. The required information is provided by implementations of *VPU* (Apache JACOB's [9] virtual processing unit, dealing with persistence of execution state and consistency), the execution queue where all activities to be executed are stored, and the representation of (executed) activities.

By monitoring of Apache ODE the process' runtime information can be made explicit, which is explained in the next section.

IV. Using an Abstract Process Model to Collect Process Runtime Information

As described before, Apache ODE does not provide detailed information about processes, instances, or the runtime. By monitoring the process engine this information can be made explicit using a model for the representation of a process and its instances at runtime. Inspired by Reichert et al. [1] a simplified process model was designed. For simplification the model only supports a sequence of process fragments. To keep the example simple, joins, forks, and parallel fragments are not considered at the moment. The presented process model is called *abstract process model* in this paper.

In Figure 4 an overview of the abstract process model is given. A *ProcessContainer* contains *Processes*. A *Process* contains an ordered sequence of *ProcessFragments*, which contains an ordered sequence of the *ProcessFragmentSteps*. A *ProcessFragment* is a specialization of *BaseFragment* which allows to describe nested fragments.

Each instance of a *Process* is represented by *Instance*. An *Instance* contains a queue *ProcessFragmentQueue* of *ConfiguredFragments*. A *ConfiguredFragment* is related to a corresponding *ProcessFragment* including runtime information. The realization of *Suspendable* allows suspending and resuming a *Process*, an *Instance* and a *ProcessFragmentQueue* while applying adaptations to them.

In the case of Apache ODE a *ProcessContainer* is the BPELServer on which the the BPEL processes are deployed. A *Process* is a BPEL process and the *ProcessFragment*s its activities. Each single step a BPEL engine has to perform for each activity is represented by a *ProcessFragmentStep*. The instance of a BPEL process including it's runtime context (if available) is represented by *Instance*. A *ConfiguredFragment* represents a BPEL activity during runtime.

The abstract process model provides the information at runtime needed for applying adaptations. A mechanism for applying adaptations and their synchronization between the process engine (Apache ODE) and the process model instances is introduced in the next section. Using this model will decouple the adaptations, described in the next section, from a specific process definition languages, such as BPEL and BPMN, as well as their process execution engines.

V. Adaptation Patterns Model

Based on the abstract process model presented in Section IV, an adaptation patterns model is presented in this section. An adaptation pattern describes a structural change of a process. To execute an adaptation, an instance of adaptation patterns model is configured with specific data about the process to be modified.

In [10]–[12] patterns of changes in process-aware information systems were identified. E.g., the insert adaptation pattern is defined as follows (from [11]):

> **Name**: Insert Process Fragment
> **Description**: A process fragment is added to a process schema.
> **Example**: For a particular patient an allergy test has to be added due to a drug incompatibility.
> **Problem**: In a real world process a task has to be accomplished which has not been modeled in the process schema so far.
> **Design Choices**: D. How is the additional process fragment X embedded in the process schema?
> (1.) X is inserted between 2 directly succeeding activities (serial insert)
> (2.) X is inserted between 2 activity sets (insert between node sets) a) Without additional condition (parallel insert) b) With additional condition (conditional insert)
> **Implementation**: The insert adaptation pattern can be realized by transforming the high level insertion operation into a sequence of low level change primitives (e.g., add node, add control dependency).

The adaptation patterns model presented in this paper allows one to specify these change patterns for a later adaptation of a running process. An outline of the adaptation patterns model is shown in Figure 5.

An *Adaptation* represents a set of *Operations*, which are applied to a process. An *Atomic Operation* is the most low-level operation like adding or deleting a process fragment (e.g., a BPEL activity). More complex operations can be assembled by these atomic operations (e.g., moving a process fragment can use the delete and insert operation) but, if needed, can be defined as an atomic operation as well for semantic distinction (e.g., moving a process fragment is semantically different from deleting and inserting it). These operations are completely independent of a particular process' structure. That is, in particular, they can be used for adapting an instance of the abstract process model introduced in Section IV as well as for adapting a BPEL process. Decoupling the operations from a specific process structure enables decoupling of the adaptation engine, too, which interprets instances of the adaptation patterns model presented in Section VI. To enable adaptation support for a specific process execution engine only these atomic opera-

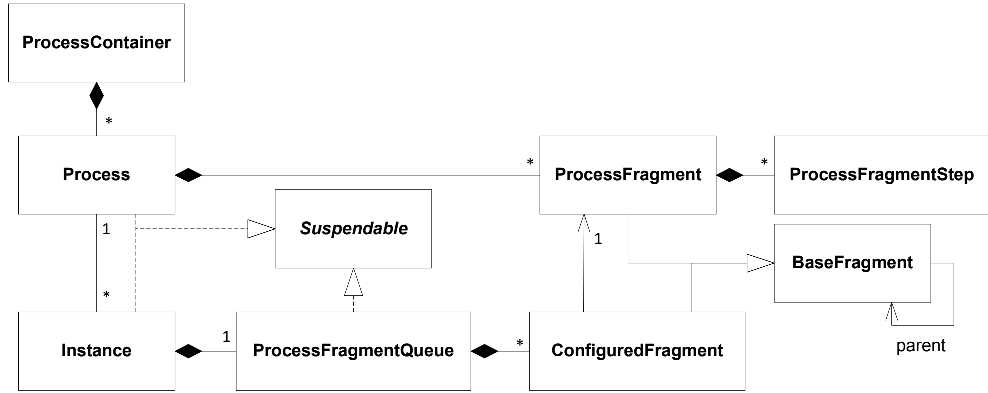

Figure 4. Overview of the abstract process model

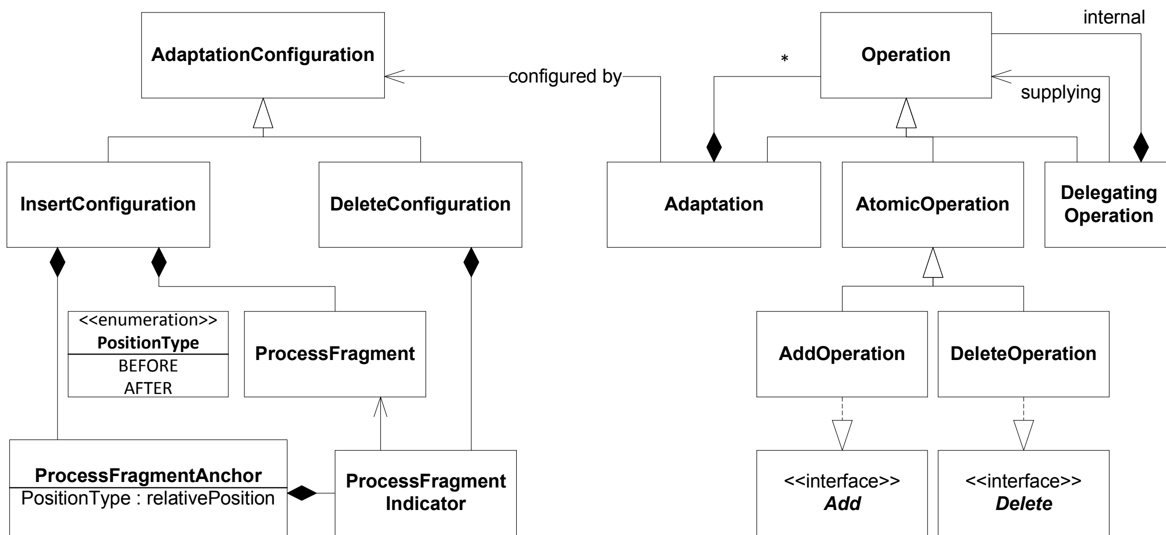

Figure 5. Outline of the adaptation patterns model

tions have to be implemented.

When the adaptation engine, presented in Section VI, interprets instances of the adaptation patterns model, it holds an instance of the abstract process model, presented in Section IV, which represents the process to be adapted of the process execution engine. If the structure of the process is changed within the process execution engine, the adaptation engine needs to know about these changes. It is assumed, that the process execution engine does not provide support for monitoring changes of a process' structure. A mechanism to apply the modifications on the processes of the process execution engine as well as on its abstract process model representation is needed. This is realized by a *Delegating Operation*, which is defined by two operations of the same type (e.g., adding a process fragment), called internal and supplying operation. First, the supplying operation is applied on the process of the process execution engine. If this operation was able to proceed successfully, the internal operation will be applied on the abstract process model instance. This will cause both process representations having the same structure after applying a *Delegating Operation*. Please note that there is a need for a transaction handling (which is not realized right now in our prototype, but will be implemented in future versions).

Figure 6 shows the relationship between the serial insert adaptation pattern, using the delegating add operation. More details about binding an adaptation patterns to instances of the corresponding operation are presented in Section VI. The code snippet in Figure 7 shows how the configuration for a serial insert adaptation pattern can be created. Though, this code should never be typed since the information can be extracted from existing process definitions, e.g. by comparing the structure of the running process instance with its modified BPEL process definition.

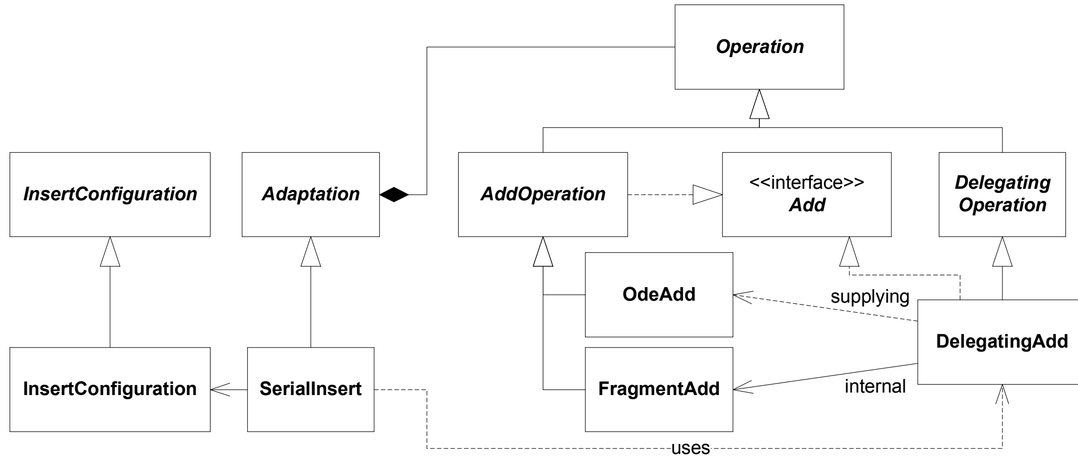

Figure 6. Example for describing a serial insert adaptation pattern for Apache ODE

```
// the abstract process representation of
// the executed process of the process execution engine
Process process;

// fetch the fragment from process, which
// represents the activity register_for_phd_program
ProcessFragment referenceFragment =
   process.findActivity("register_for_phd_program");

// create the new bpel activity
// get_passport_photo to be added
OProcess owner = (OProcess) process.getBase();
OActivity parent =
   (OActivity) referenceFragment.getParent().getBase();
OInvoke activity_get_passport_photo =
   new OInvoke(owner, parent);
// set properties for activity_get_passport_photo
[…]

// create the according process fragment
// for the BPEL activity
ProcessFragment newFragment = new ProcessFragmentImpl(
   activity_get_passport_photo,
   referenceFragment.getParent(),
   referenceFragment.getOwningProcess());


AbstractInsertConfiguration insertConfig =
   new SerialInsertConfiguration();
insertConfig.setNewFragment(newFragment);
insertConfig.setFragmentAnchor(
   new ProcessFragmentAnchor(
      referenceFragment,
      ProcessFragmentPositionType.AFTER
   )
);
```

Figure 7. Example code snippet for serial insert configuration

So far, an adaptation knows which operations have to be performed, but it does not know on to which process fragments it should be applied during runtime. A configuration solves this problem by providing information about the specific process structure. For example, the adaptation for deleting a process fragment needs to be configured with the specific process fragment to be deleted. The adaptation for inserting a new process fragment needs to be configured with the new process fragment and a relative position to an existing process fragment, as shown in Figure 2.

VI. Adaptation Engine

The adaptation engine provides functionality for applying adaptations to a process. A main objective of the adaptation engine is to easily adapt it to any process engines, which can be mapped to our abstract process model, and their respective process definition languages (in this paper we focus on BPEL engines, but in principle our abstract process model can also be applied to other process definition languages).

The adaptation engine's task is to interpret instances of the adaptation patterns model and to apply these adaptations to processes and its instances within the process execution engine. The adaptation patterns model is presented in Section V, Figure 5 gives an outline of the model.

To apply an *Adaptation* to a process, a description of a specific adaptation is needed, which is represented by an instance of a adaptation patterns model. For example, as shown in Figure 6, for inserting a process element, information about the new process element and its position relative to an already existing process element is needed. This description is defined by an *AdaptationConfiguration*. A *ProcessFragmentIndicator* is used for identifying a specific process fragment within a process. To define a relative position in relation to a process fragment (e.g., before, after), a *ProcessFragmentAnchor* is used. The binding between a description and the corresponding operation happens at runtime. As shown in Figure 8, the adaptation engine is configured by a specific implementation of an *Adaptation Engine Configuration*, which provides a factory for instancing operations and knows about the mapping between adaptations and operations. When the execution of

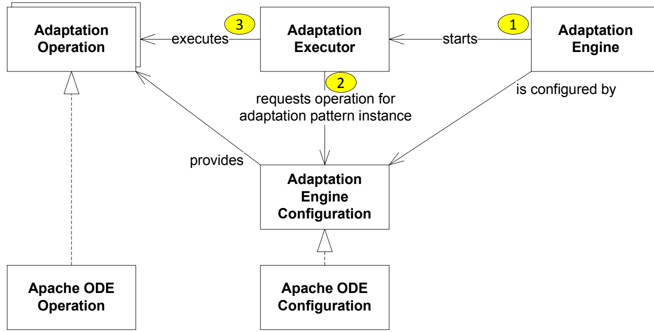

Figure 8. Operation binding at runtime using Adaptation Engine Configuration

an adaptation starts (1), the *Adaptation Executor* requests the corresponding operation for the adaptation to be executed (2), which then gets executed (3).

Using this binding at runtime and by just implementing the set of atomic operations for a specific process execution engine, an *Adaptation* can easily be adjusted to work on any process models and platforms, (e.g., not just on Apache ODE with BPEL, but jBPM with jPDL or BPMN as well).

Figure 1 gives an schematic overview of the relations between the Adaptation Engine, Adaptation Configurations, Adaptation Operations and the process environment to be adapted.

Figure 6 shows an outline of a more specific example: serially inserting a new process fragment. The *InsertConfiguration* defines the new process fragment to be added as well as it's new position. It configures the adaptation *Serial Insert*, which uses a *Delegating Add* to add the new fragment to both process representations, the abstract process model and BPEL. The *Delegating Add* will only apply the implementation of an *Add* to the abstract process model (via *Fragment Add*) if the add operation was successful on the Apache ODE engine. Which implementation of *Add* is used to adapt the Apache ODE engine is elicited during runtime using mappings for each process engine and process model. Using AOP for monitoring Apache ODE, as described in Section III, enables connecting process fragments of the abstract process model with the corresponding objects processed within Apache ODE. Operations can directly work on these objects, without querying the process engine for them during runtime.

When an adaptation is applied to a running instance of a process within a process engine, the instance must be paused. Because Apache ODE does not support this feature, the implementation of an instance's execution queue was replaced by a suspendable implementation using AOP . There is still a need for controlling processes and instances during runtime, but is not realized right now.

VII. RELATED WORK

A workflow system supporting adaptations for long running processes is presented by Minor et al. [13]. It provides a partial suspension mechanism, so elements which will not be affected by the modifications can continue to be executed during adaptation. Reasoning over previous adaptations are used for supporting the user to apply future ones.

Another process-aware information system supporting modifications during runtime is ADEPT, presented by Reichert et al. [1]. It supports modification of process level and process instance level as well as instance migration for evolving process schemes.

Even though these two approaches solve the problem, it does not handle process adaptation in a generic way, as it does not allow to extend existing process execution engines with adaptation support, but is dependent on a proprietary process engine. The approach presented in this paper provides an adaptation engine which can be used to easily extend existing process execution engines. The only effort to connect the adaptation engine with a process execution engine is to implement process execution engine specific monitoring and a very limited number of atomic operations for applying modifications to processes and its instances.

Other approaches enable monitoring and adaptation support by transforming the original process and enrich it with additional information and structural behavior using middleware. Because the original process differs from the executed process, it will be difficult to close the gap between the user performing an adaptation to the original process and the process execution engine executing a process which differs to the original one the user wants to adapt. If the information about the transformation as well as the process structure is issue of change, their co-evolution must be manageable for an adaptation engine, too.

Baresi and Guinea presented the framework Dynamo [14], [15] allowing to create monitoring rules, which can be weaved with a process during deployment time. During runtime, the monitoring can be parametrized.

Ezenwoye and Sadjadji present the TRAP/BPEL framework [5] that enables service adaptation by using service proxies. An adaptation ready version of a BPEL process can be generated, where web services are replaced by either a static or dynamic service proxy, where a static service proxy offers alternative, equivalent compositions of web services and a dynamic service proxy uses UDDI lookup for dynamically identifying equivalent web services.

VieDAME presented by Moser et al. [3], [4] allows dynamic adaptation of web service composition using aspect-oriented engine adaptation plus indirection by an interception and adaptation layer. Defined quality of service (QoS) properties are used to select an appropriate, equivalent service. Equality of services have to be defined manually and

stored in a repository. A service invocation is intercepted by the presented framework using aspect oriented programing.

Another, similar approach presented by Agarwal and Jalote [2] is calculating equality of services, using semantical annotated service descriptions. This allows to automatically use new available services without maintaining a repository.

The approaches for dynamic web service composition of Ezenwoye et al., Moser et al. and Agarwal et al. allow extending existing process engines with adaptation support, where adaptation support means exchanging web services. However, they do not support modifying a process' structure during runtime. Weber et al. [12] identify requirements on process-aware information systems to support these changes as well as patterns for changes. An evaluation of selected approaches and systems focusing change support is presented, as well.

MASC (Manageable and Adaptive Service Compositions) [16], presented by Erradi et al., provides a middleware for adaptive composite web services. A transformation of the process to be instantiated wraps each activity with a special activity, which acts as listener for and executor of adaptations, as well as executing the wrapped activity. In contrast to MASC the approach presented in this paper aims to offer a solution which can easily be plugged onto an existing process engine, without setting up a middleware.

Leitner et al. [17] use an approach, where sequences of a graph based workflow can be supplemented by composition fragments, which are linked to the original workflow. Using an aspect based description of fragment substitution, the fragments can be weaved into the origin workflow during runtime. The proposed generic fragments can be compared to the adaptation pattern model presented in our approach, where we focus on enabling the extension of process engines with model based adaptation support.

An approach without transforming a process is AO4BPEL [18]. This approach addresses expressing crosscutting concerns of processes and uses the aspect oriented paradigm for changing web service composition during runtime. Although AO4BPEL enables modifying a process' structural behavior, it can be hard to predict or reconstruct the sequential flow of a process adapted with aspects, especially with a growing amount of aspects.

VIII. Conclusion and Discussion

This paper presents an approach for adaptation of processes during runtime using the example of BPEL. The adaptation engine is able to interpret and execute instances of the presented model for adaptations and to apply the adaptations to deployed processes and its instances to any process execution engine. To show the feasibility of our approach, adaptation support has been implemented in a prototype for Apache ODE.

The adaptation framework presented in this paper shows that it is feasible to perform structural adaptation to BPEL processes running in existing process engines. In particular, it is possible to extract runtime information about instances from existing process engines and add adaptation support without touching the engine's code.

However, our adaptation framework still has some limitations: Still a lot of adaptation patterns exist which are not (yet) realized by the model. Currently the model is implemented in Java. However, an implementation in established modeling frameworks such as EMF will be used in future version (to be able to use EMF's tool support). As mentioned above, transaction support when updating the abstract process model and the engine model would be desirable.

Our approach enables a number of further future works: Modeling the adaptation patterns enables analyses of a concrete set of adaptations with established technologies, e.g., comparison of different sets of adaptations, monitoring concurrent adaptations, detection of dependencies between adaptations (e.g., conflicts, causal dependencies, mutual exclusion, ...) which can support optimization of adaptations as well as concurrency handling. By comparing two process versions, instances of configured adaptation pattern can be generated. Conversely, configured instances of adaptation patterns can be used for realizing versioning, using adaptations as deltas between versions. E.g., Langer et al. presented an approach for model versioning using operations [19], [20] for conflict detection, where operations are similar to configured adaptations presented in this paper.

References

[1] M. Reichert and P. Dadam, "Enabling Adaptive Process-aware Information Systems with ADEPT2." in *Handbook of Research on Business Process Modeling*. Information Science Reference, 2009, pp. 173–203.

[2] V. Agarwal and P. Jalote, "From Specification to Adaptation: An Integrated QoS-driven Approach for Dynamic Adaptation of Web Service Compositions," *IEEE International Conference on Web Services*, pp. 275–282, Jul. 2010.

[3] O. Moser, F. Rosenberg, and S. Dustdar, "VieDAME - flexible and robust BPEL processes through monitoring and adaptation," in *Companion of the 30th International Conference on Software Engineering*, ser. ICSE Companion '08. New York, NY, USA: ACM, 2008, pp. 917–918.

[4] ——, "Non-intrusive monitoring and service adaptation for WS-BPEL," in *Proceeding of the 17th international conference on World Wide Web*, ser. WWW '08. New York, NY, USA: ACM, 2008, pp. 815–824. [Online]. Available: http://doi.acm.org/10.1145/1367497.1367607

[5] O. Ezenwoye and S. M. Sadjadi, "TRAP/BPEL: A Framework for Dynamic Adaptation of Composite Services," in *Proceedings of the International Conference on Web Information Systems and Technologies (WEBIST 2007)*, ser. WEBIST 2007, 2007.

[6] The Apache Software Foundation, “Apache ODE,” http://ode.apache.org/, Sep. 2010.

[7] G. Latuske, “Business Process Illustrator,” http://sourceforge.net/projects/bpi/, Sep. 2010.

[8] ——, “Sichten auf Geschäftsprozesse als Werkzeug zur Darstellung laufender Prozessinstanzen,” Diplomarbeit, Universität Stuttgart, Fakultät Informatik, Elektrotechnik und Informationstechnik, Germany, Oktober 2010.

[9] The Apache Software Foundation, “Apache ODE Jacob,” http://ode.apache.org/jacob.html, Sep. 2010.

[10] B. Weber, S. Rinderle, and M. Reichert, “Change Patterns and Change Support Features in Process-Aware Information Systems,” in *Proceedings of the 19th International Conference on Advanced Information Systems Engineering*. Springer-Verlag, 2007, pp. 574–588.

[11] B. Weber, S. Rinderle-Ma, and M. Reichert, “Identifying and Evaluating Change Patterns and Change Support Features in Process-Aware Information Systems.” Centre for Telematics and Information Technolog, Technical Report, March 2007.

[12] B. Weber, M. Reichert, and S. Rinderle-Ma, “Change patterns and change support features - enhancing flexibility in process-aware information systems,” *Data & Knowledge Engineering*, vol. 66, no. 3, pp. 438 – 466, 2008.

[13] M. Minor, D. Schmalen, A. Koldehoff, and R. Bergmann, “Structural Adaptation of Workflows Supported by a Suspension Mechanism stand by Case-Based Reasoning,” *16th IEEE International Workshops on Enabling Technologies: Infrastructure for Collaborative Enterprises (WETICE 2007)*, pp. 370–375, 2007.

[14] L. Baresi and S. Guinea, “Towards Dynamic Monitoring of WS-BPEL Processes,” in *Proceedings of the 3rd International Conference of Service-Oriented Computing (ICSOC 2005)*, ser. Lecture Notes in Computer Science, vol. 3826. Springer, 2005, pp. 269–282.

[15] ——, “Dynamo: Dynamic Monitoring of WS-BPEL Processes,” in *Proceedings of the International Conference on Service-Oriented Computing (ICSOC05)*, ser. Lecture Notes in Computer Science, B. Benatallah, F. Casati, and P. Traverso, Eds. Springer, 2005, vol. 3826, pp. 478–483.

[16] A. Erradi, V. Tosic, and P. Maheshwari, “MASC - .NET-Based Middleware for Adaptive Composite Web Services,” in *IEEE International Conference on Web Services (ICWS 2007)*. IEEE, 2007, pp. 727–734.

[17] P. Leitner, B. Wetzstein, D. Karastoyanova, W. Hummer, S. Dustdar, and F. Leymann, “Preventing SLA Violations in Service Compositions Using Aspect-Based Fragment Substitution,” in *Service-Oriented Computing*, ser. Lecture Notes in Computer Science. Springer Berlin / Heidelberg, 2010, vol. 6470, pp. 365–380.

[18] A. Charfi and M. Mezini, “Aspect-Oriented Web Service Composition with AO4BPEL,” in *Web Services*, ser. Lecture Notes in Computer Science. Springer Berlin / Heidelberg, 2004, vol. 3250, pp. 168–182.

[19] P. Langer, K. Wieland, and P. Brosch, “Specification, Execution, and Detection of Refactorings for Software Models,” in *Proceedings of the Work-in-Progress Session at the 8th International Conference on the Principles and Practice of Programming in Java (PPPJ 2010)*. CEUR-WS.org, 2010.

[20] P. Brosch, P. Langer, M. Seidl, and M. Wimmer, “Towards End-User Adaptable Model Versioning: The By-Example Operation Recorder,” in *Proceedings of the 2009 ICSE Workshop on Comparison and Versioning of Software Models*. IEEE, 2009, pp. 55–60.