Institute of Architecture of Application Systems

State Propagation for Business Process Monitoring on Different Levels of Abstraction

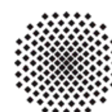
David Schumm, Gregor Latuske, Frank Leymann,
Ralph Mietzner, Thorsten Scheibler

Institute of Architecture of Application Systems,
University of Stuttgart, Germany
lastname@iaas.uni-stuttgart.de

BibTeX:

```
@inproceedings{Schumm11,
  author    = {David Schumm and Gregor Latuske and Frank Leymann and
               Ralph Mietzner and Thorsten Scheibler},
  title     = {State Propagation for Business Process Monitoring
               on Different Levels of Abstraction},
  booktitle = {Proceedings of the 19th European Conference on
               Information Systems (ECIS 2011)},
  year      = {2011},
}
```

© 2011 David Schumm, Gregor Latuske, Frank Leymann,
Ralph Mietzner, Thorsten Scheibler.



Universität Stuttgart
Germany

STATE PROPAGATION FOR BUSINESS PROCESS MONITORING ON DIFFERENT LEVELS OF ABSTRACTION

David Schumm, Institute of Architecture of Application Systems, University of Stuttgart, Universitätsstraße 38, 70569 Stuttgart, Germany.

Gregor Latuske, Institute of Architecture of Application Systems, University of Stuttgart, Universitätsstraße 38, 70569 Stuttgart, Germany.

Frank Leymann, Institute of Architecture of Application Systems, University of Stuttgart, Universitätsstraße 38, 70569 Stuttgart, Germany.

Ralph Mietzner, Institute of Architecture of Application Systems, University of Stuttgart, Universitätsstraße 38, 70569 Stuttgart, Germany.

Thorsten Scheibler, Institute of Architecture of Application Systems, University of Stuttgart, Universitätsstraße 38, 70569 Stuttgart, Germany.

Abstract

Modeling and execution of business processes is often performed on different levels of abstraction. For example, when a business process is modeled using a high-level notation near to business such as Event-driven Process Chains (EPC), a technical refinement step is required before the process can be executed. Also, model-driven process design allows modeling a process on high-level, while executing it in a more detailed and executable low-level representation such as processes defined in the Business Process Execution Language (BPEL) or as Java code. However, current approaches for graphical monitoring of business processes are limited to scenarios in which the process that is being executed and the process that is being monitored are either one and the same or on the same level of abstraction. In this paper, we present an approach to facilitate business-oriented process monitoring while considering process design on high-level. We propose process views for business process monitoring as projections of activities and execution states in order to support business process monitoring of running process instances on different levels of abstraction. In particular, we discuss state propagation patterns which can be applied to define advanced monitoring solutions for arbitrary graph-based process languages.

Keywords: Process Monitoring, Process View, State Abstraction.

Acknowledgement:
The author David Schumm would like to thank the German Research Foundation (DFG) for financial support of the project within the Cluster of Excellence in Simulation Technology (EXC 310/1) at the University of Stuttgart.

Introduction

Business Activity Monitoring (BAM) has been identified as an important field for *Business Process Management* (BPM) long ago (Gartner Research 2002). Nowadays, BAM approaches and products are in place and *Key Performance Indicators* (KPIs) can be monitored. KPIs represent formulas used to measure a particular aspect of the business as a whole, or to measure an aspect of a single process instance. For example, a KPI could measure the number of received orders per hour. In order to visualize their evaluation in business dashboards, all required execution events need to be processed, this is typically performed by *Complex Event Processing* (CEP) engines. However, for some tasks in BPM it is not sufficient to check the status of a set of process instances in a business dashboard. Support for process-based analysis of the status of a current instance can be seen as an additional source of information. In many process languages like the *Business Process Model and Notation* (BPMN) by OMG (2010) or *Event-driven Process Chains* (EPC) (Kindler 2004), a process model can be represented by a process graph. Such a graph is made up of nodes and edges. Nodes represent activities which are executed in a process and edges stand for control dependencies between activities. In order to visualize the current status of a process instance, the nodes and edges in this graph need to be augmented with runtime information about their status (like running, completed, faulted etc.). The components which actually execute the automated processes (so-called *process engines*) provide a database which contains the status of each process instance: the *audit trail*. The contents of this database can be mined and used for the augmentation of the nodes in the process graph. Tools for process modeling and analysis can present this information to the user, for example a decorator on nodes can be used to visualize the current status of an activity as illustrated in Figure 1 (lower right).

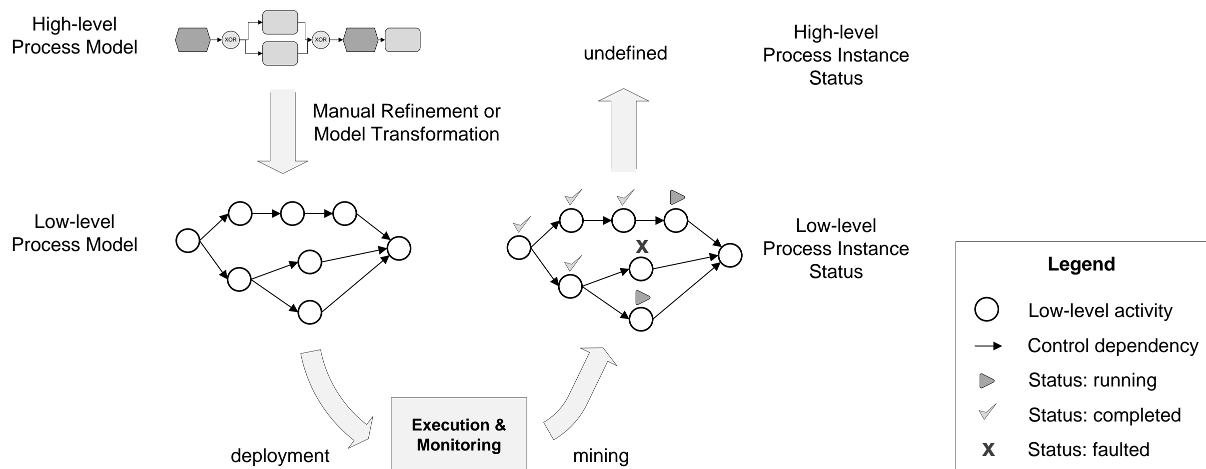


Figure 1. Current status of a process instance is undefined on high-level.

The procedure for the graphical monitoring of process instances described above is state of the art in current BPM solutions. However, this approach is limited to scenarios in which the process the user wants to monitor and the process that is actually being executed are identical. A common use case for this is the modeling, execution and monitoring of processes defined in the *Business Process Execution Language* (BPEL) by OASIS (2007). There are scenarios in which this is not the case though. For example, business processes are often modeled on a high-level notation near to business, like BPMN or EPC. This requires a refinement phase in which a process is modified and augmented by technical personnel. It may even require a translation to a different format in order to make the process executable. In addition, model-driven process design allows modeling a process on high-level, while actual execution code is generated from that high-level description. As depicted in Figure 1 (upper right), a problem arises in those scenarios in which manual refinement or model transformation is involved: The status of the process instance on the high-level is undefined.

In this paper we address this problem with the help of process view transformation concepts and the propagation of execution states from a low-level model to a higher level. A process view can be seen as a presentation of the result of specific transformations applied to a given process model. These transformations comprise structural changes like omission and aggregation of activities and also visualization techniques can be applied. The main contribution of this paper is a discussion of patterns and their application to define user-oriented monitoring solutions for arbitrary graph-based process languages. These *state propagation patterns* capture different ways on how to propagate the status of a low-level process instance to a high-level model.

The paper's further structure is the following: In Section 2 the background is explained and works related to our approach are discussed. Section 3 describes the application scenarios that motivate our approach and that provide requirements for the propagation of states. Section 4 describes the elementary patterns which are needed to manage process monitoring of the presented application scenarios. In Section 5 we discuss advanced aspects of state propagation. In Section 6 we present a generic architecture to support the approach of business process monitoring on different levels of abstraction. Opportunities and limitations of the approach are characterized in Section 7.

1 Background and Related Work

Typically, an arbitrary high-level process cannot be automatically transformed to a low-level execution format. A refinement phase in which technical experts semi-automatically map the high-level model (e.g., based on EPC) to an execution language (like BPEL) is often required. This refinement phase may include a change of the process structure leading to a process model that significantly differs from the high-level process model. However, when the high-level language and the low-level execution format are aligned, then the refinement phase might not be necessary and monitoring on high-level is more or less straightforward. Two languages can be aligned when they have a similar expressiveness and language constructs. One technique to align two process languages is language subsetting. This denotes that particular language constructs may not be used in a process when they are not supported by the other language. Some BPM vendors use this technique in order to allow users to model a process on high-level while executing it on lower level, as illustrated in Figure 2. As showed in Schumm et al. (2009), a certain subset of BPMN and BPEL can be aligned.

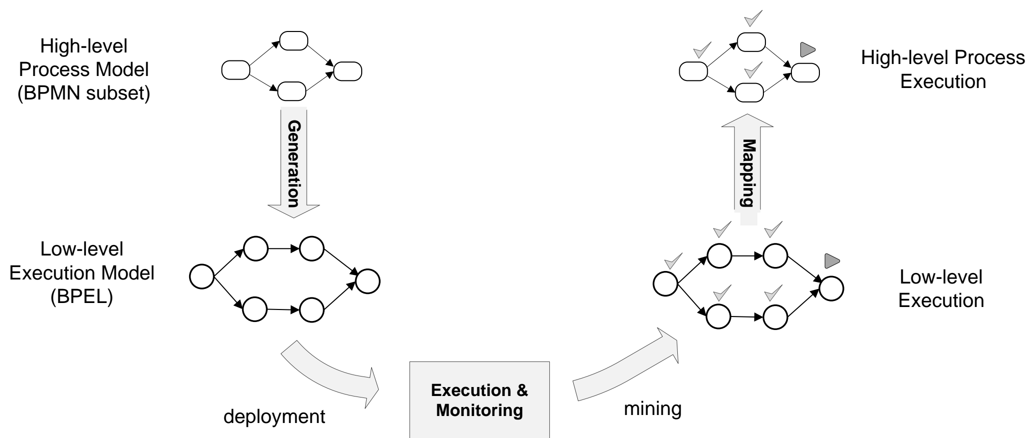


Figure 2. Monitoring a low-level process instance on high-level when the high-level language and the low-level language are aligned.

Process view transformation concepts are originating from approaches that address the increasing size of process models. Early works on process views mostly discussed the creation of sub-models of process graphs, i.e., graph abstraction. For example, in Avrilionis et al. (1996) a mechanism for creation and composition of sub-models of Petri Nets was proposed. Due to the increasing relevance of BPM on the one hand and the increasing complexity of business processes on the other hand, more

and more works on process views have been proposed. Most of them make use of model transformations such as aggregation or omission of structures, as shown in Figure 3a. These techniques can for example be used to generate a public view on a process that hides internal details (Eshuis and Grefen 2008). Nowadays, process views also employ visualization techniques to reduce complexity, separate concerns and abstract from undesired details. In summary, process views allow a reduction of the complexity of a process and they allow a user to focus on particular aspects.

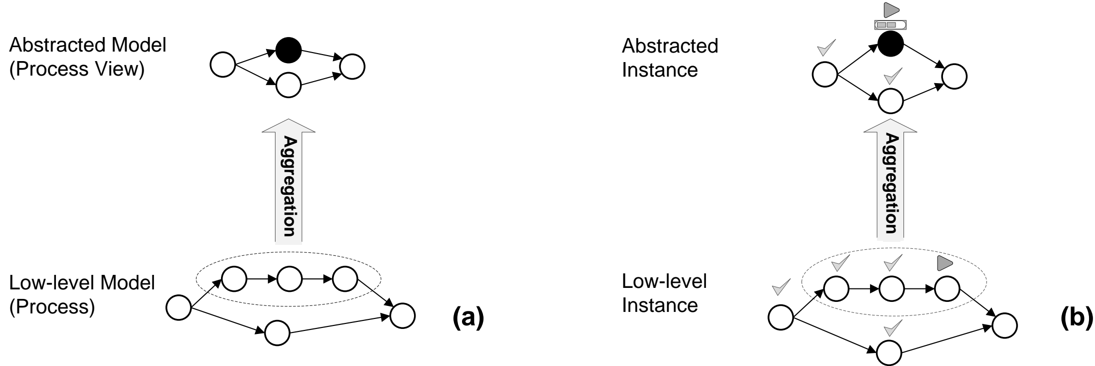


Figure 3. A view on a process model (a) and a view on a process instance (b).

Similar to public views, Naef et al. (2001) introduced the concept "shadow process", which provides an outline of a process while hiding confidential details. For monitoring support, the execution state of activities of the process is being mapped to the activities which are visible in the shadow process. We also consider this concept in the *Direct State Propagation Pattern*. Zur Mühlen (2004) described a "business state" as an abstracted state model of an underlying process state model. This approach contributes to the *State Combination Pattern*. Although recent works on utilizing abstraction and view transformations for process instance monitoring are promising (Rinderle et al. 2006, Reichert et al. 2010), research on views for monitoring is still in an early stage - the ways to propagate a low-level state to a higher level are directly bound to the transformation operation as the example for aggregation in Figure3b shows. Some frameworks already provide support for graphical instance monitoring of abstracted process models. Most notably the custom monitoring tool in Java CAPS (Oracle 2010) allows the usage of a different process model for monitoring than for execution. However, it does not provide state propagation rules or automation to project the status of a low-level process instance to a high-level model. To the best of our knowledge there currently exists no approach to facilitate monitoring at different levels of abstraction in the generic way we propose.

The approach described in Momm et al. (2009) shows how model-driven development can be leveraged for monitoring Web service compositions. According to this work, process performance indicators can be defined for a high-level process model and subsequently transformed into monitoring instructions for a low-level process model. The approach in Barbon et al. (2006) discusses monitoring of performance indicators in Web service compositions with the concept of instance indicators and class indicators. An instance indicator applies to a single process instance, while a class indicator aggregates the runtime information of all instances of a process model. We reflect this concept in the *Instance State Aggregation Pattern*. In summary, existing works on BAM are much more advanced in supporting different levels of abstraction in process monitoring, but they cannot be directly adapted to monitor the concrete or abstracted status of a process instance based on a process graph.

2 Application Scenarios

In this section, we describe two scenarios in which the process that has been modeled (high-level) differs from the one that is actually being executed (low-level). We discuss for each scenario how state propagation and monitoring views can be utilized to enable business process monitoring on high-level process models. We describe multiple scenarios to show the general applicability of our approach.

2.1 Abstract Process Modeling and Custom Monitoring

Decidr (http://www.decidr.eu) is a "BPM as a service" platform that allows users to model, execute and monitor human-centric workflows. Users sign up for the service in the Decidr Web portal and then can model their own high-level process models described in the *Decidr Workflow Definition Language* (DWDL) using pre-defined activity types in a Web-based editor. It is feasible to use a high-level notation that abstracts from technical details, as this allows a user to focus on the process structure. Once a user has modeled such a process and would like to execute it, the high-level DWDL process model is transformed into a low-level BPEL process model using pre-defined mapping rules. The resulting model is then executed on a standard BPEL engine. For example the "send e-mail" activity in DWDL is converted into a sequence of *assign*, *invoke* and *assign* activities in BPEL: the parameters which have been set on the high-level *send e-mail* activity are split and transformed into one activity which prepares the input data for the e-mail service invocation (*assign*), an activity that calls a Web service which actually sends the e-mail (*invoke*), and an activity to make the response of the e-mail service available for further processing (*assign*).

However, as users have modeled DWDL, they do not want to see the instance status of the generated BPEL process model, but they want to see the status of the process they have modeled in DWDL. Thus, a projection from the low-level status to high-level is desirable as shown in Figure 4 ("State Projection"). Furthermore, in addition to the high-level instance status, various custom monitoring views can be defined which are tailored to the individual needs of different stakeholders. For each stakeholder a monitoring view could be generated which transforms the low-level process model into a custom representation on a higher level of abstraction, see Figure 4 (upper right, "Transformation").

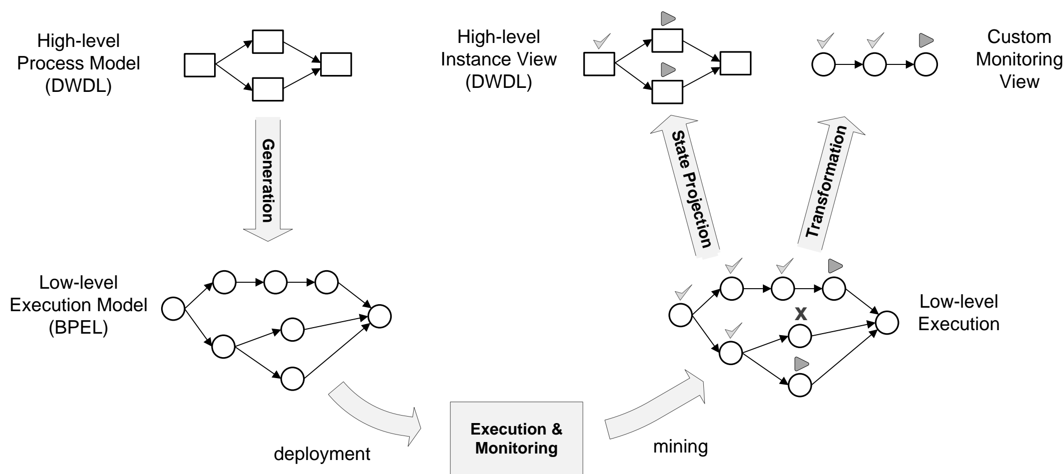


Figure 4. Approach for abstract process modeling and custom monitoring.

2.2 Modeling and Monitoring in Integration Scenarios

Enterprise Application Integration Patterns (Hohpe and Woolf 2003) are a toolbox to decompose complex integration tasks into small and manageable recurring problems. EAI patterns suggest guidelines and solutions on how to solve these smaller recurring problems. They are structured into atomic patterns and additionally describe how to combine these atomic patterns into composite patterns that can be used to solve complex integration problems. *Parameterizable EAI patterns* (PEP) by Scheibler et al. (2009) is an extension of those patterns. PEPs allow IT architects dealing with integration problems to reuse the advice captured in EAI patterns and to configure the patterns so that they fit to a specific integration problem. By utilizing the *Model-driven Development* (MDD) approach (Völter and Stahl 2006), PEPs can be transformed into executable code. *Generating Enterprise Integration Executable Scenarios* (GENIUS) by Scheibler and Leymann (2009), for example, provides

PEP as modeling artifacts within a graphical user interface and transforms this model into various executable target platforms, for example into BPEL and WS-* integration infrastructures. Thus, a PEP model can be transformed into executable code like BPEL processes. Monitoring of such an integration scenario is possible on BPEL level. However, users also would like to analyze and monitor on the same level they used for modeling. Therefore, the execution information, i.e., the *status* of the low-level execution, has to be propagated to the PEP level as illustrated in Figure 5 (upper right). This can be done because the executable code is annotated with the information which BPEL elements represent which elements in the PEP.

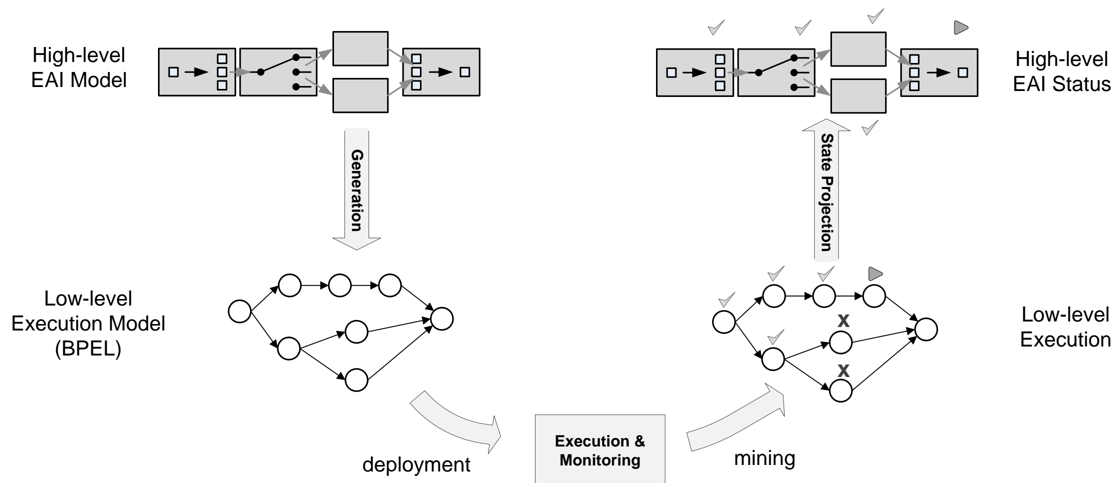


Figure 5. Approach for modeling and monitoring in integration scenarios.

In this application scenario some patterns are transformed into complex BPEL snippets. Moreover, expected normal behavior of some EAI patterns is even an error state at BPEL level. For example, the executable model of an *Aggregator* (Hohpe and Woolf 2003) handles a timeout exception as completion condition with the help of an *onAlarm* handler in BPEL. In terms of the EAI patterns, this is a normal and expected behavior and has to be represented accordingly at the PEP level. This means that the EAI pattern has to be marked as successfully completed although it is exceptional behavior in BPEL. In the following, generic patterns are described that can be utilized to map low-level states to high-level states and thus enable monitoring on the same level as modeling in model-driven scenarios.

3 State Propagation Patterns

In this section, we discuss the state propagation patterns which we identified during the work with the application scenarios and related approaches. To derive these patterns, we investigated the transformation from high-level to low-level model constructs. For each high-level construct we specified a solution how its state can be derived from the low-level elements which are generated out of it. The different forms of state propagation we observed are the basis for the patterns we propose. Both application scenarios contribute a different set of patterns, and for each pattern they provide meaningful examples and motivation. Study of related work (see Section 2) contributed to this further.

We defined all patterns in terms of the *rationale* of their usage and provide a *figure* that illustrates the pattern. We also give a small *example* of a situation when to use the pattern, based on the application scenarios. For space reasons we refrain from a complex pattern description as it is done for example in Hohpe and Woolf (2003). For the application scenarios, the patterns we present are sufficient and can be applied to other scenarios as well to solve common problems in process instance monitoring. However, we make no claim to provide a complete list of patterns for any monitoring scenario.

3.1 Direct State Propagation Pattern

Rationale: The most straightforward pattern for propagating a state from low-level to high-level is to define a direct projection of the status of an activity on low-level to an activity on high-level. This pattern is applicable to all cases in which a single activity from a high-level model is represented by a single activity in the low-level model. Figure 6 illustrates this pattern.

Examples: A visual representation of a BPMN task as modeled in the high-level modeling tool is presented to the user instead of a BPEL invoke activity as it is executed in the low-level execution engine. The visual representation includes the appropriate execution state.

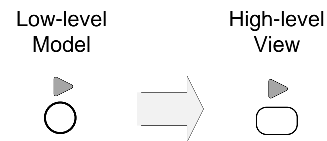


Figure 6. Direct state propagation.

3.2 State Alteration Pattern

Rationale: A state of an activity in the low-level is interpreted differently in the corresponding activity at high-level. This pattern is applicable in cases where the state space (i.e., the set of state types) of the low-level model is different to that of the high-level. Another use case is the situation when a state in a low-level model has a different semantic meaning on high-level. Figure 7 illustrates this pattern.

Examples: A caught and handled fault in the low-level model shall not be shown as a fault in the high-level view, because the fault needs to be thrown and caught due to a limitation of the modeling possibilities of the low-level model. Another example is the simplification of the high-level view, where only three different states are shown for simplicity: *Running*, *Completed*, *Faulted* whereas the underlying low-level model states are mapped to these three basic states.

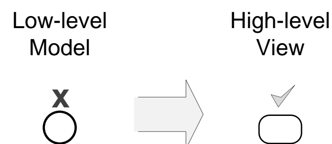


Figure 7. State alteration.

3.3 State Combination Pattern

Rationale: Several low-level activities ($a_1 \ldots a_n$) implement one single activity *hl* in the high-level model. The state *s(hl)* of the high-level activity is derived as a function *f* over the states of multiple, but not necessarily connected low-level activities, i.e., $s(hl) = f(s(a_1),\ldots,s(a_n))$. Arbitrary functions can be specified for the state combination. Figure 8 illustrates a straightforward form of state combination.

Examples: Several BPEL activities in a low-level execution model such as an *assign* activity, an *invoke* activity and a second *assign* activity are represented as one activity (task) in a high-level BPMN model. The state combination function is informally defined as follows: "if one of the low-level activities is in state running, the high-level activity is also in running, if all low-level activities are completed, the high-level activity is completed".

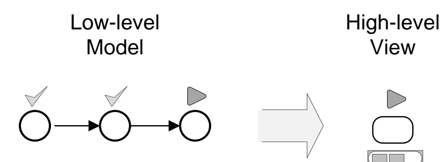


Figure 8. State combination.

3.4 State Deduction Pattern

Rationale: Some activities modeled in a high-level model are not directly reflected in the low-level model. Thus, their states have to be deduced from other states. In cases where a high-level activity that is not present in the low-level model has to be shown, this state deduction pattern is used. The state of

these activities can be deduced from the overall status of the process instance, from external events, or from other activities, for example from the preceding or following activities in the control flow.

Examples: A monitoring view for an administrator shows several activities that run parallel to the regular activities of the process. These activities represent non-functional properties of the process. For example, one activity represents the security status of the process engine that executes the process. The status of this activity, which has no representation in the low-level model, could either be deduced from events emitted by the corresponding security components, or be statically set to the status *Running* etc.

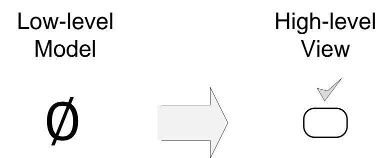


Figure 9. State deduction.

3.5 Transition State Pattern

Rationale: An activity present in a low-level model is not present in a high-level model. However, since it takes some time to process this activity the shown state of the high-level view can be inconsistent during the processing of the hidden activity. To indicate that processing is under way, transitions between the predecessors and successors of the hidden activity are annotated with a status.

Examples: A high-level BPMN diagram contains a task, *retrieve customers*, which is mapped to an *invoke* activity as well as a *for-each* assigning the customers to a special list in a low-level BPEL model. In the high-level view it should be shown that the customer retrieval has been completed, which can be done once the *invoke* activity completed. However, the loop is still executing and the subsequent activity is thus not running yet. Therefore, the control connector from the *retrieve customers* activity to the subsequent activity is annotated with a transition state indicating that something not specified in detail still needs to be completed before the next activity can be executed. Figure 10 illustrates this pattern.

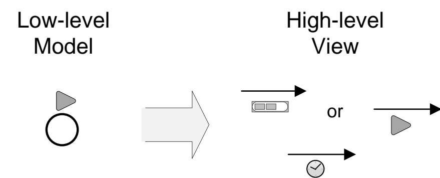


Figure 10. Transition state.

3.6 Instance State Aggregation Pattern

Rationale: In a high-level view, state information for several instances of a low-level model should be presented at once. In this case, each activity of the high-level view is annotated with the states of multiple instances of the corresponding low-level activity. Additionally, the set of low-level instances to include is specified by a query.

Examples: In a high-level view, all instances of a low-level model that faulted in the last 5 days should be shown to determine where faults occur often. In this case at first all instances that faulted in the last 5 days must be determined from the audit trail and then for each high-level activity the states of the corresponding low-level activities have to be aggregated as depicted in Figure 11.

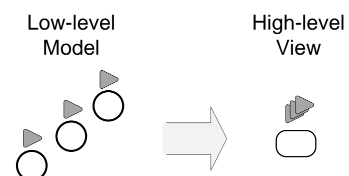


Figure 11. Instance state aggregation.

4 Advanced Aspects of State Propagation

In this Section, we discuss advanced aspects of state propagation. These aspects depict the prospects of the approach, and they point out future directions in this field of research.

4.1 Complex State Propagation Patterns

Based on the elementary state propagation patterns presented in Section 4, complex patterns and projections can be defined. For example, we can formulate a complex *state distribution pattern*, which is based on the direct state propagation pattern.

Rationale: One single activity in a low-level model implements several activities in a high-level model. These high-level activities should be annotated with a state in the high-level view. To show the state of the high-level activities, the state of the low-level activity is distributed over various high-level activities, see Figure 12.

Examples: A high-level BPMN diagram contains two tasks t_1 and t_2 (e.g. *send request* and *receive response*) which are implemented by one single Web service invocation in a low-level BPEL process. The state of t_1 and t_2 is then the state of the low-level BPEL *invoke* activity.

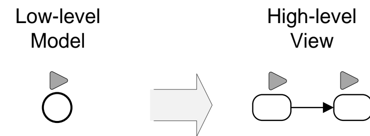


Figure 12. State distribution.

The state distribution pattern is just one example to illustrate that arbitrary forms of state propagation can be defined based on the presented elementary patterns. Complex state propagation forms can also be composed by chaining different state propagations, e.g., a high-level state can be defined by combining two different input states, with one already altered: $s(h1) = combine(s(a_1), alter(s(a_2)))$.

4.2 Definition of State Projections

In the following, we show an exemplary state projection for the abstract process modeling and monitoring scenario (Figure 13a) and the enterprise application integration scenario (Figure 13b). A subset of the presented patterns is used to define the projection of states from low-level execution to a high-level instance view and to the EAI model respectively. In these two scenarios the executable low-level models are automatically generated out of the high-level models. The generation of the state projection definition is enabled by automatically annotating the executable code with the information which low-level elements represent which elements in the high-level model.

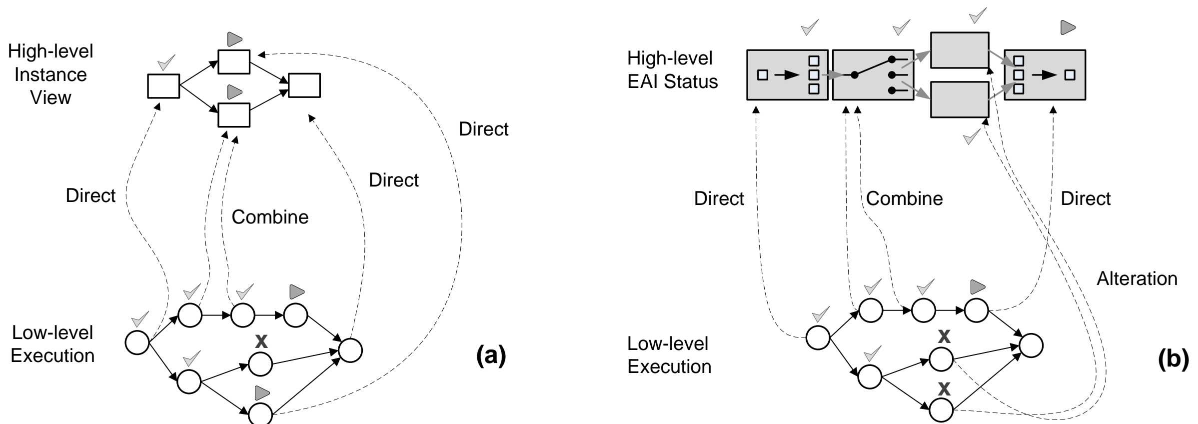


Figure 13. Exemplary state projections: DWDL monitoring (a) and EAI pattern monitoring (b).

However, support for scenarios in which the low-level models are created manually or semi-automatically is harder to achieve. This concerns the *definition* of the projection as well as its *maintenance*. Regarding the definition of the state projection one possibility besides specifying the projection manually, is to make use of techniques from recommender systems. For example, a simple state projection definition could possibly be generated for a high-level and a low-level model input based on similarity of activity labels or similarity of associated metadata. Still, even such a recommended state projection likely needs to be refined manually. Regarding the maintenance of the state projection, we call a state projection sound (or intact) if and only if (i) the projection is surjective, i.e., a state propagation rule exists for all activities in the high-level model, and (ii) any resulting high-level instance state conforms to particular consistency rules, for example "an activity can only be in the status *Running* when its predecessor has successfully completed" etc. When defining or maintaining a state projection, projection soundness needs to be considered and it has to be ensured that only well-defined state propagation rules are created.

5 Generic Architecture for Viewing-enabled Process Monitoring

In this section, we present a generic architecture for business process monitoring on different levels of abstraction. Implementation of this architecture in a monitoring prototype is ongoing work. An initial version of the prototype, capable of view-based monitoring of BPEL processes, is presented in Schumm et al. (2011). The architecture is shown in Figure 14. The main components in the architecture are a Web-based client, a view designer, an application server, and a process engine. These components can be distributed to different machines for scalability reasons. In the following, we elaborate on the components of this architecture.

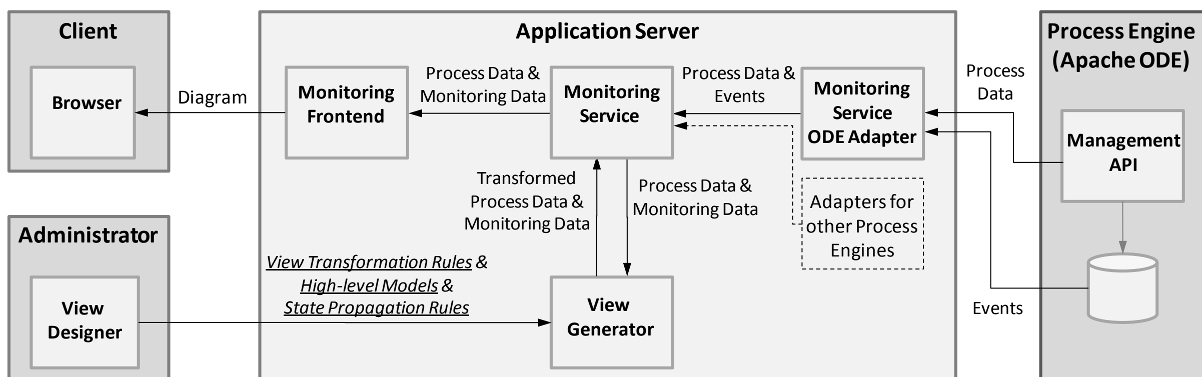


Figure 14. Generic architecture for viewing-enabled process monitoring.

The *ViewDesigner* (Figure 14, lower left) is used by an administrator for the definition of monitoring views. The monitoring view definitions consist of (i) *view transformation rules* to create a custom view on the process model using view transformations, (ii) *high-level models* to support monitoring process models which cannot be generated automatically using view transformations, and (iii) *state propagation rules* to project execution states from low-level models onto the high-level models. These definitions are deployed on the application server. This covers two approaches: If the intended level of abstraction can be met with view transformation techniques, then view transformation rules can be defined and deployed by the administrator. For example, particular activities can be omitted, highlighted, or aggregated this way. However, if the monitoring view (i.e., the process model that should be shown to the user) differs significantly in structure and notation from the low-level model, then the state propagation approach needs to be taken. In that case an administrator could define and deploy a "Chevron" process model (used for example in Microsoft PowerPoint). In addition, state propagation rules need to be deployed that define the state projection from a particular BPEL process (low-level) to that Chevron model (high-level). The Browser on the *Client* is then used to display

generated process instance diagrams to the user. These diagrams are supplied as *Scalable Vector Graphics* (SVG) images. The main reason for providing a Web-based client is that it allows providing monitoring functionality to arbitrary user groups without installation of additional software. This enables a business user in the same way as a customer to monitor a low-level process based on a model and notation that is feasible for the user and the intended level of abstraction.

Therefore, the *ViewGenerator* provides two functionalities. The first is to create a view on a given process model, based on process view transformation techniques. Building on the basic patterns of process view transformations discussed in Schumm et al. (2010), abstract views on business processes can be created. The second functionality is to evaluate state projection definitions (i.e., sets of state propagation rules) in order to project the execution states of low-level instances to the execution states of high-level models which have been deployed.

The *MonitoringFrontend* provides access to the application and presents the SVG file to the client. The frontend requests the needed process data (process models deployed on the process engine and their process instances and activities) and the generated SVG file from the *MonitoringService*.

In turn, the *MonitoringService* provides interfaces for getting the process data and events that occur during the execution of a process instance. These interfaces hide the concrete implementation which is needed to communicate with the process engine. The main tasks of this component are (i) the parsing of the process model documents, (ii) the aggregation of the events to monitoring data to obtain the status of the activities of a process instance, (iii) their assignment to the corresponding activities in the process model, and (iv) the generation of the SVG file. The data of already parsed process model documents can be cached in this component to reduce the computing time. The *MonitoringService* can retrieve the process model documents from the process engine. To support monitoring on different levels of abstraction, the *MonitoringService* can invoke the *ViewGenerator* component.

The *MonitoringServiceODEAdapter* provides a concrete implementation to retrieve monitoring data from a specific process engine. Figure 14 shows an exemplary adapter for the Apache ODE process engine (Apache Software Foundation 2010). Using this adapter, the monitoring of processes based on BPEL can be enabled. This adapter can be exchanged in order to integrate the monitoring application with other process engines. In our architecture, the Management API of the process engine is used to retrieve the process data via Web service calls. To retrieve all required events of a concrete process instance, the database of the process engine needs to be queried, all events are persisted there.

6 Opportunities and Limitations of the Approach

One of the main problems we observe in the automation of business processes lies in the establishment of a systematic and consistent connection between the different levels of abstraction and granularity of these processes. This refers to the models of business processes, their enactment, monitoring, and analysis. In this paper, we presented patterns of state propagation, which we derived from two different application scenarios. We argue that these patterns ease the specification and implementation of solutions to facilitate process monitoring across the boundaries of process models and languages, not necessarily limited to model-driven approaches. Possibly, state propagation could be used for an end-to-end propagation of the execution state of low-level models for process execution up to abstract views on business levels. However, we did not investigate multi-level state projections, yet.

Both the opportunities and the limitations of the presented approach require further exploration. Regarding the limitations, we observed undesirable projection phenomena in high-level models which require investigation. For instance, we observed token jumps in high-level models due to dead paths in lower level models in BPEL-to-BPMN projections. Furthermore, algorithms for the soundness of projections across different process languages demand for additional effort. At the moment, we are developing a formal basis of the presented approach. We will tackle the formal description of the individual patterns, their composition, and the soundness of state projections by providing an “algebra” of state propagation.

References

Avrilionis, D., Cunin, P.Y., and Fernström, C. (1996). OPSIS: a View Mechanism for Software Processes which Supports their Evolution and Reuse. Proceedings of the 18th International Conference on Software Engineering (ICSE), IEEE Computer Society.

Barbon, F., Traverso, P., Pistore, M., and Trainotti, M. (2006). Run-Time Monitoring of Instances and Classes of Web Service Compositions. Proceedings of the IEEE International Conference on Web Services (ICWS), IEEE Computer Society.

Eshuis, R., and Grefen, P. (2008). Constructing Customized Process Views. In: Data & Knowledge Engineering, 64(2):419–438, Elsevier.

Gartner Research (2002). Business Activity Monitoring: Calm Before the Storm, available at http://www.gartner.com/DisplayDocument?doc_cd=105562, accessed 18/03/2011.

Hohpe, G. and Woolf, B. (2003). Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions. Addison-Wesley Professional, Amsterdam.

Kindler, E. (2004). On the Semantics of EPCs: A Framework for Resolving the Vicious Circle. Proceedings of the 2nd International Conference on Business Process Management, pages 82–97.

Momm, C., Gebhart, M., and Abeck, S. (2009). A Model-Driven Approach for Monitoring Business Performance in Web Service Compositions. Proceedings of the 4th International Conference on Internet and Web Applications and Services (ICIW), IEEE.

Naef, A., Schuler, C., Schuldt, H. (2001). Monitoring komplexer Dienste in unternehmensübergreifenden Prozessen am Beispiel von SAP R/3 Business Workflows. BTW'2001, Springer.

Object Management Group (OMG) (2011). Business Process Model and Notation (BPMN). OMG Available Specification, Version 2.0.

Oracle (2010). Java CAPS Business Process Monitor: Custom Monitoring Tools with Java CAPS, available at http://wikis.sun.com/display/JavaCAPS/Graphical+Business+Process+Monitoring, accessed 18/03/2011.

Organization for the Advancement of Structured Information Standards (OASIS) (2007). Business Process Execution Language 2.0 (BPEL), OASIS Standard.

Reichert, M., Bassil, S., Bobrik, R. and Bauer, T. (2010). The Proviado Access Control Model for Business Process Monitoring Components. Enterprise Modelling and Information Systems Architectures - An International Journal. German Informatics Society (GI).

Rinderle, S., Bobrik, R., Reichert, M., and Bauer, T. (2006). Business Process Visualization – Use Cases, Challenges, Solutions. Proceedings of the International Conference on Enterprise Information Systems (ICEIS), pages 204–211, INSTICC Press.

Scheibler, T., and Leymann, F. (2009). From Modelling to Execution of Enterprise Integration Scenarios: the GENIUS Tool. Proceedings of the 16th Conference on Communication in Distributed Systems (KiVS'09).

Scheibler, T., Mietzner, R., and Leymann, F. (2009). EMod: Platform Independent Modelling, Description and Enactment of Parameterisable EAI Patterns. In: Enterprise Information Systems, 3(3):299–317, Taylor & Francis Inc., Bristol, PA, USA.

Schumm, D., Karastoyanova, D., Leymann, F., and Nitzsche, J. (2009). On Visualizing and Modelling BPEL with BPMN. Grid and Pervasive Computing Workshops: 4th International Workshop on Workflow Management (IWWM), IEEE Press.

Schumm, D., Latuske, G., Leymann, F. (2011). A Prototype for View-based Monitoring of BPEL Processes, Technical Report No. 2011/04, Universität Stuttgart, Fakultät Informatik, Elektrotechnik und Informationstechnik.

Schumm, D., Leymann, F., and Streule, A. (2010). Process Viewing Patterns. Proceedings of the 14th International IEEE Enterprise Distributed Object Computing Conference (EDOC 2010), IEEE Computer Society Press.

The Apache Software Foundation (2010). Apache ODE (Orchestration Director Engine), available at http://ode.apache.org/, accessed 18/03/2011.

Völter, M., and Stahl, T. (2006). Model-Driven Software Development. Wiley & Sons.

Zur Muehlen, M. (2004). Workflow-based Process Controlling. Logos Verlag Berlin.