Goal-Oriented Model-Driven Business Process Monitoring Using ProGoalML

Falko Koetter[1] and Monika Kochanowski[2]

[1] University of Stuttgart IAT, Germany
falko.koetter@iao.fraunhofer.de
[2] Fraunhofer IAO, Germany
monika.kochanowski@iao.fraunhofer.de

Abstract. In today's fast changing business world, the fulfillment of process goals needs to be constantly evaluated and adjusted. But processes are often carried out by systems which are not process aware. aPro is a modular architecture for business process optimization. In aPro process models can't be guaranteed to be executable but need to be monitored. In this paper, we propose a modeling language for process metrics, key performance indicators and goals and use the interchange format ProGoalML to automate creation and setup of monitoring infrastructure.

Keywords: business process management, process monitoring, business process goals, process adaptation, business intelligence.

1 Introduction

In today's commerce business processes are volatile and interconnected, causing the necessity to react to changes in a timely and correct fashion [8]. But only changes impacting the goals of the process necessitate an adjustment of the process. Thus, to assess the need for change, the goals of a process as well as their degree of fulfillment have to be known. Monitoring solutions today[21,2,3] focus on executable processes.

However, our work in the industry as well as other sources[16][14] found executable process models to be the exception rather than the norm. Thus, business process models are often disconnected from process execution and serve only documentation purposes. While switching to executable processes is often not possible due to existing systems or lack of IT support in single process steps, there still is a need for business process monitoring and optimization[14]. Companies need a way to implement missing capabilities without abandoning existing solutions. Thus, the contribution of this work is to provide a model-driven approach for monitoring business processes which takes into account the current state of BPM adoption.

We propose aPro, a modular *A*rchitecture for business *PR*ocess *O*ptimization (Figure 1). aPro is based on the business process lifecycle as described in [19], containing components for process modeling, execution, monitoring, analysis and

W. Abramowicz et al. (Eds.): BIS 2012, LNBIP 117, pp. 72–83, 2012.

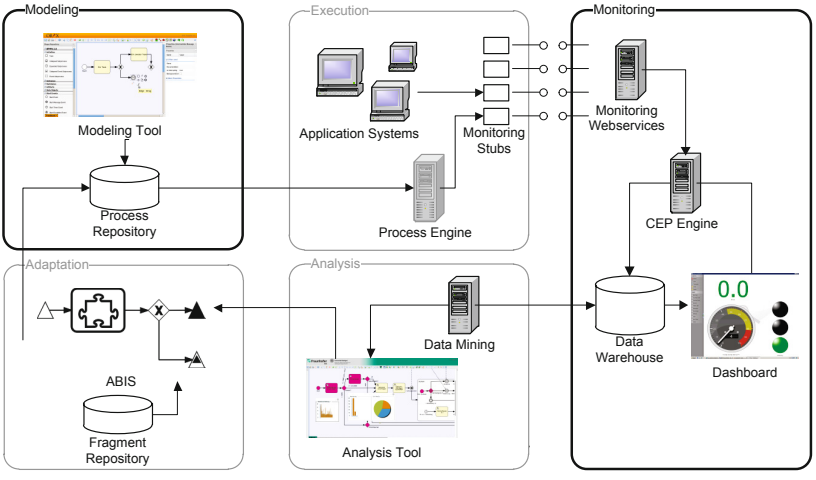

Fig. 1. Overview of the aPro architecture. Bold parts highlight focus of this work.

adaptation. During *Modeling* a process model is created using a modeling tool and stored in a process repository. Then during *Execution* the process is either executed by a process engine or by a collaboration of (legacy) application systems, which are not process-oriented. During execution, measurement of relevant metrics has to be performed, e.g. synchronous or asynchronous. In any case, a measurement is compiled into a call to a *monitoring* web service, which transfers the data to a Complex Event Processing (CEP)[12] Engine[1]. It correlates the measurements of a process instance, calculates key performance indicators (*KPI*s) and checks goal fulfillment. The CEP Engine then provides the processed data to the next steps: Real-time data is displayed on a dashboard and long-term data is stored in a data warehouse. During *analysis* data mining is used to find deficits and identify possible adjustments of the process. These adjustments are then used to *adapt* the process using for example ABIS[20], a tool for adaptive business processes.

In this work, we focus on the steps modeling and monitoring, though other components already exist[10]. For defining process goals, we extend BPMN 2.0[15] with modeling elements for metrics, KPIs and goals. Multiple components are involved in aPro, and each of them needs to be configured and adapted independently. As this would result in prohibitive effort, we generate all configuration files from the defined goals, KPIs, metrics and process model. ProGoalML, the *Pro*cess *Goal M*arkup *L*anguage, serves as an intermediary between the different files as shown in Figure 2. We focus on the highlighted parts, describing modeling, creation of a ProGoalML file and automatic generation of measurement and result schemata as well as CEP rules, thus encompassing all steps necessary to perform basic process monitoring. This paper is structured as follows. We first describe the modeling steps and give a motivational example in Section 2. In Section 2.2 we explain the structure of ProGoalML. Section 3 describes the measuring of metrics

and the creation of CEP rules and result schemata. Section 4 describes a prototypical implementation as well as evaluation. In Section 5 we examine related work. Section 6 gives a conclusion and outlines future work.

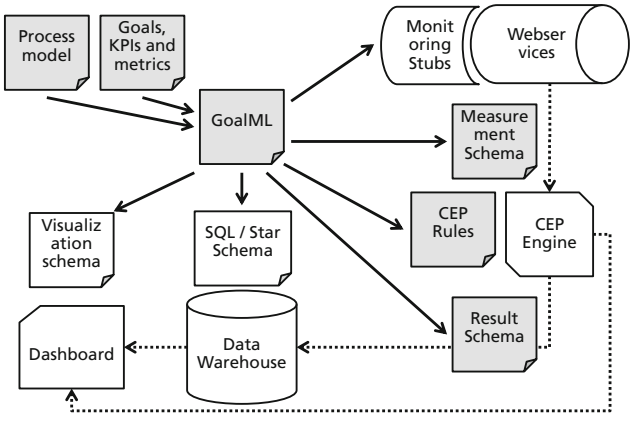

Fig. 2. Overview of documents created with ProGoalML. Dotted lines indicate flow of monitoring data between components. Grey parts highlight focus of this work.

2 Modeling Goals, KPIs and Metrics Using ProGoalML

In order to define a goal model and create a ProGoalML document, aPro introduces additional modeling elements used in conjunction with a BPMN diagram (see Figure 3).

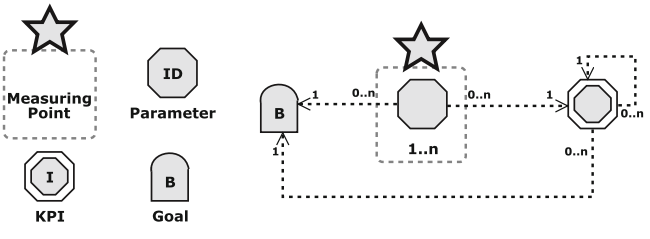

Fig. 3. ProGoalML modeling elements

To measure metrics during execution, *measuring points* are used. A *measuring point* can be attached to any BPMN element, at which a measurement is to be taken. It contains one or more *Parameters* which are to be measured. A *Parameter* has a name and a primitive data type. A special case is the ID type, which is used to correlate measurements of a process instance. Based on Parameters, *KPI*s can be calculated. A KPI defines a function which references Parameters or other KPIs. A *goal* is similar to a KPI as it is defined by a function as well. However, this function returns a Boolean value indicating whether the goal has been fulfilled or not. Abbreviations are used to indicate which type a measurement, goal or KPI has.

We define a *Parameter* p as a tuple $p \in P = (n, t, E_p)$

where n is the name of the *Parameter* and the type $t \in T$, with $T = \{$Boolean, Enumeration, Integer, Double, String, ID, Long$\}$ as the set of types. If $t = Enumeration$, then E_p is the set of possible enumeration values, else $E_p = \emptyset$.

We define a *measuring point* m as a tuple

$$m \in M = (e, P_m)$$

where e is the BPMN element the measuring point is attached to and P_m is the set of Parameters belonging m. As a parameter may only belong to a single *measuring point*, given P as the set of all parameters the following must hold

$$\forall m_1, m_2 \in M : P_{m_1}, P_{m_2} \subset P \wedge m_1 \neq m_2 \rightarrow P_{m_1} \cap P_{m_2} = \emptyset$$

We further define a KPI k as a tuple

$$k \in K = (n, f, V_k)$$

where n is the Name of the KPI, f f is the function to calculate the KPI and $V_k \subset P \cup K$ is the set of input variables. Similarly, a goal is defined as

$$g \in G = (n, f, V_g)$$

where n is the name of the goal, f is a function returning a Boolean and $V_g \subset P \cup K$ is the set of input variables. For the purpose of KPI calculation it is necessary that there are no cyclic input variables, as otherwise no order of calculation may be found. To ensure this, the following must hold:

$$\forall k \in K : \forall v \in V_k : \nexists v_0, v_1, ..., v_n \in K : (v_o = v_n = v \wedge \forall i \in [0, n) : v_i \in V_{v_{i+1}})$$

2.1 Motivational Example

Figure 4 shows a simplified claim handling process of a car insurance company which checks if a claim is justified. In the first step the claim is entered by an employee in a claims management system containing among others the stipulated amount. The second step is performed by an expert system and calculates a reference amount for the claim based on the address given by checking repair shop and rental car prices. In the last step a report is generated in the claims management system and a decision about the claim is made. Either the claim is accepted, accepted with a reduced amount, rejected or an error occurred.

In order to monitor this process, *measuring points* are defined at the activities. The first measuring point at *Enter Claim* contains two *IDs*, *ClaimID* and *Address*, as well as two other parameters: *Timestamp* of type long indicating the time the claim was entered and *Amount* of type double, the amount stipulated by the claim. The second measuring point at *Calculate Reference Amount* contains an ID named *Address*, which is the same address as in the first step and another parameter *Amount* of type double, the reference amount calculated by the expert system. The third measuring point at *Decide Claim* contains the same *claimID* as the first measuring point and two other parameters: A *Timestamp* indicating when the claim was decided and the *Result* of the Decision as an Enumeration containing the values $ACCEPTED$,$PARTIALLY\_ACCEPTED$,

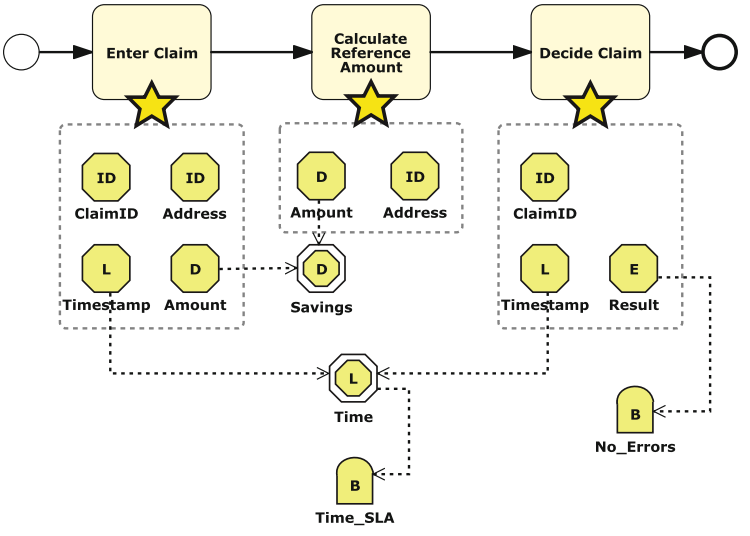

Fig. 4. Example process annotated with measuring points, KPIs and goals

REJECTED and *ERROR*. Measuring points are named after the elements they are attached to in order to uniquely define their parameters and measurements.

Based on these three measuring points two *KPIs* and two *goals* are calculated. The first KPI is the process execution *time* measured in seconds starting from the moment the claim has been entered, calculated from both timestamp values using the function

$$\text{Time} := (\text{Decide\_Claim.Timestamp} - \text{Enter\_Claim.Timestamp})/1000.0$$

Based on this KPI, a *goal* named *Time_SLA* is defined, mandating the execution time to be below 60 seconds:

$$\text{Time\_SLA} := \text{Time} < 60.0$$

The second KPI, *Savings* achieved by the expert system is calculated using the function

$$\text{Savings} := \text{Enter\_Claim.Amount} - \text{Calculate\_Reference\_Amount.Amount}$$

The second *goal* mandates that the process finishes without an error:

$$\text{No\_Errors} := \text{Decide\_Claim.Result} \neq \text{"ERROR"}$$

Note that when parameters from different measuring points are used in a function, the measurements have to be *correlated* with each other using IDs.

2.2 ProGoalML

After *measuring points*, *KPIs* and *goals* have been modeled, a ProGoalML document has to be created to serve as an input for configuration document creation as shown in Figure 2.

Abridged ProGoalML document from motivational example(Figure 4)

```
<Progoalml version="1.0">
    <Meta>...</Meta>
    <GoalModel>
        <MeasuringPoint name="Calculate_Reference_Amount">
            <RefBpmn>Calculate_Reference_Amount</RefBpmn>
            <Parameter name="Address">
                <DataType>ID</DataType>
            </Parameter>
            <Parameter name="Amount">...</Parameter>
        </MeasuringPoint>
        <MeasuringPoint name="Decide_Claim">...</MeasuringPoint>
        <MeasuringPoint name="Enter_Claim">...</MeasuringPoint>
        <KeyPerformanceIndicator name="Savings">
            <Formula>
                Enter_Claim.Amount - Calculate_Reference_Amount.Amount
            </Formula>
            <RefParameter>
                <ParameterName>Amount</ParameterName>
                <MeasuringPointName>Enter_Claim</MeasuringPointName>
            </RefParameter>
            ...
            <DataType>double</DataType>
        </KeyPerformanceIndicator>
        <KeyPerformanceIndicator name="Time">...</KeyPerformanceIndicator>
        <Goal name="No_errors">
            <Formula>Result != "ERROR"</Formula>
            <RefParameter>...</RefParameter>
            <DataType>boolean</DataType>
        </Goal>
        <Goal name="Time_SLA">...</Goal>
    </GoalModel>
    <ProcessModel>...</ProcessModel>
</Progoalml>
```

A ProGoalML document consists of a goal model, a process model and metadata like title and creation date. The *process model* is created by removing all aPro-related elements from the process and serializing the resulting standard BPMN 2.0 model. The *goal model* consists of measuring points, KPIs and goals. A *measuring point* contains its parameters as well as a reference to the BPMN element it belongs to. A parameter consists of a name, a data type and, if it is an enumeration, all possible enumeration values. KPIs consist of a data type, a formula and references to all input variables. Parameters are referenced by their name and the name of the measuring point they belong to. Similar to KPIs goals consist of a data type, a formula and references to all input variables, though the data type has to be *boolean*. However, in future work other data types may be supported to measure partial fulfillment of goals.

3 Measuring and Result Creation

As shown in Figure 1, monitoring data has to be gathered from *application systems*. To instrument these diverse application systems, different monitoring stubs are necessary, ranging from simple web service calls to periodically evaluating local log files. For each measuring point a separate monitoring stub may be created befitting the particular system performing the corresponding step.

In any case, whenever a measurement occurs, the monitoring stub calls its corresponding monitoring web service, transmitting all measured parameters. The monitoring web service then transfers the measurement to the CEP engine as an *event*. This way, CEP engine and monitoring stub are decoupled, resulting in simplified and engine-independent stub code.

Measurement schema for Calculate_Reference_Amount *(see Figure 4)*

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
   <xs:element name="Calculate_Reference_Amount"
        type="Calculate_Reference_Amount"/>
   <xs:complexType name="Calculate_Reference_Amount">
      <xs:sequence>
         <xs:element name="refBPMN" type="xs:string"/>
         <xs:element name="Address" type="xs:id"/>
         <xs:element name="Amount" type="xs:double"/>
      </xs:sequence>
   </xs:complexType>
</xs:schema>
```

For each measuring point an XML schema is generated, describing the structure of a measurement, called a *measurement schema* (see Figure 2). It defines the interface between monitoring stub, web service and CEP engine. In our example (see Figure 4) three schemata are generated in total.

In order to get monitoring results the CEP engine will gather measurements and calculate KPIs and goals. As KPIs and goals may be calculated from Parameters belonging to multiple *measuring points*, it is necessary to correlate all measurements from a process instance in order to obtain a matching set of input variables. For example in Figure 4 the *Savings* are calculated from two separate *measuring points*. For correlation Parameters of the type $t = ID$ (i.e. IDs) are used. IDs with the same name are considered to have identical values in a single process instance, thus can be used to correlate their *measuring points* with each other, finding measurements which belong to the same process instance. Utilizing transitivity, measuring points with different IDs may be correlated as well. Consider our motivational example (see Figure 4). Two kinds of ID are used, *CaseID* and *Address*. The measuring point at *EnterClaim* contains both IDs, so it can be correlated to both other measuring points, which contain one of the IDs each. Thus, all three measurements which occur in a process instance can be correlated and KPIs spanning multiple measurements can be calculated.

We define the coverage class of a measuring point m_x as follows

$C_{m_x} = \{m \in M | \exists m_o, ..., m_n \in M : m_n = m_x \wedge \forall i \in [0, n) : \exists p_1 \in P_{m_i}, p_2 \in P_{m_{i+1}} : n_{p_1} = n_{p_2} \wedge t_{p_1} = t_{p_2} = ID\}$

KPIs and goals may only use input variables belonging to one coverage class.

If a measuring point m_1 is contained in the coverage class of another measuring point m_2, their coverage classes are identical:

$$m_1 \in C_{m_2} => m_2 \in C_{m_1} => C_{m_1} = C_{m_2}$$

Thus, in our example, all coverage classes are identical and contain all three measuring points.

Algorithm to find distinct coverage classes

$I := \{x | x \in P \wedge t_p = \text{"ID"}\}$
$C := \emptyset$
$f : I \mapsto C_m : f(id) := \emptyset$
foreach $(m \in M)$
    $C_m := \{m\}$
    foreach $(p \in P_m \cap I)$
        if $(f(p) \neq \emptyset)$
            $C := C \backslash \{f(p)\}$
            $C_m := C_m \cup f(p)$
        endif
    endfor
    $C := C \cup \{C_m\}$
    foreach $(p \in \{p | p \in I \wedge \exists x \in C_m : p \in P_x\})$
    $f(p) := C_m$
    endfor
endfor

To generate result schemata and CEP rules we need to identify the set C of all distinct coverage classes using the algorithm above. Whenever it encounters a measuring point m which has an ID p already present in another coverage class $f(p)$, coverage classes are merged, thus ensuring every measuring point sharing an ID is in the same coverage class and only distinct coverage classes remain in C. For each coverage class a CEP rule for process instance correlation and a result schema is generated, as shown in Figure 2.

A result schema contains all goals, KPIs and parameters of a coverage class. The name of a measurement is assembled from measuring point and parameter names. IDs are named differently, as they have the same values among all measuring points and their names are globally unique.

Abridged result schema for motivational example (compare Figure 4)

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
   <xs:element name="Handle_Claim"
        type="Handle_Claim"/>
```

```
   <xs:complexType name="Handle_Claim">
      <xs:sequence>
         <xs:element name="Case_ID" type="xs:id"/>
         <xs:element name="Address" type="xs:id"/>
         <xs:element name="Enter_Claim.Timestamp" type="xs:long"/>...
         <xs:element name="Savings" type="xs:double"/>...
         <xs:element name="No_Errors" type="xs:boolean"/>...
      </xs:sequence>
   </xs:complexType>
</xs:schema>
```

Similar to SQL and tables, the Esper Event Processing Language allows defining a statement on incoming events, called a rule. This rule uses the IDs to correlate all measurements from a process instance, calculates KPIs and goals and creates a result event. For each process instance, the CEP rule creates a result according to the schema. These messages may then be stored in a data warehouse, displayed on a dashboard or further aggregated within the CEP engine (future work).

Abridged CEP rule to create results for motivational example (Figure 4))

```
INSERT INTO Manage_Claim SELECT A.ClaimID as ClaimID, ...,
A.Amount - B.Amount as Savings, ..., C.Result != "ERROR" as No_Errors, ...,
A.Amount as Enter_Claim\.Amount, ..., C.Result as Decide_Claim\.Result
from pattern [every A=event(refBPMN="Enter_Claim")
-> B=event(Address=A.Address and refBPMN="Calculate_Reference_Amount")
-> C=event(ClaimID=A.ClaimID and refBPMN="Decide_Claim")]
```

4 Prototype and Evaluation

To evaluate ProGoalML, we created a prototype for modeling goal models, as well as generating ProGoalML files, measuring and monitoring schemata and CEP rules. The prototype is based on Oryx, a web-based tool for collaborative modeling[7]. It has been used to model the motivational example in Figure 4.

We then created a test driver generating random measurements from a given ProGoalML file. We evaluated the documents created by our prototype in multiple examples and found them to be correct. However, when placing a measuring point inside a loop, multiple measurements per process instance may occur.

Further on, we created an interactive test driver allowing tests with more realistic data. We extended the motivational example to the real-world process it represents and tested it using sample data gained by studying the real system. This resulted in correct results as well and we plan to instrument the production system in order to further evaluate GoalML.

5 Related Work

Goal modeling is a topic in requirements engineering[11], where goals for a (future) system are defined. Compared to ProGoalML goals are defined a priori, before the system in question exists. Goal models are then used to find goal

conflicts, identify necessary requirements and ensure requirement completeness. Goals in requirements engineering are linked and may support or contradict each other[5]. In [9] an overview of goal-oriented requirements engineering is given. Goals may be formalized using temporal logic which may be used as a basis for verification of a system implementing the requirements, e.g. by checking if a statement holds true at all times. In comparison, ProGoalML is used to define goals a posteriori, focusing on measuring their fulfillment rather than ensuring it, which is performed in the later steps of the aPro architecture. As ProGoalML checks goal fulfillment on a process instance level compared to a system level, there is no temporal dimension to goal definition. Aggregating goal fulfillment is performed in the later stages of aPro and subject for future research.

[6] defines a notation for modeling complex events in a BPMN process, similarly to the definition of measurement points in ProGoalML. We experimented using extended BPMN events for monitoring purposes with the BPMN engine *activiti*, but concluded that we need events from a broader range of systems.

In [17] Process Performance Indicators (PPIs), which parallel KPIs in ProGoalML, are defined using an ontology. They as well may be calculated hierarchically from measurements and may span a process instance or the whole process. In the latter case measurements from a process instance are aggregated. We plan to address aggregation of KPIs and goals in future work using further CEP rules. Similarly to ProGoalML, PPIs are to be used across the business lifecycle, but measurement and other steps are not described in detail. A graphical notation for PPIs is planned, but has not been implemented yet.

Similarly to creating KPIs from multiple values, [4] defines Service Level Objects in a hierarchical fashion in order to find causes of service level violations. Monitoring data is correlated using hierarchically structured event logs, a technique not applicable for aPro, as event logs may not exist.

In [3] a method for run-time validation of WS-BPEL processes is presented. The process is augmented with rules like pre- and post-conditions to use a *monitoring manager* as a proxy for service calls which polices rule compliance. In comparison to ProGoalML rules need to be written manually and separate from the process. Similarly, [2] monitors WS-BPEL processes by extending the runtime engine and transmitting events to a monitoring engine. Instance monitors for monitoring single instances and class monitors for gathering statistics across instances may be specified using *Monitoring Rules*, which, similarly to ProGoalML are then used to create Java Code. Rules for monitors are similar to CEP rules[12] and deliver numeric or Boolean values like KPIs and goals, but have to be specified manually separate from the process. Both approaches require a process engine.

[21] presents a top-down approach for modeling process performance metrics of a BPEL process. Process and PPM model are transformed to a monitoring model for use in a *Business Activity Monitoring* (BAM) tool and an event filter for a specific process engine, selecting events sent to the BAM tool. As in [2] instance and aggregate performance metrics are differentiated. Like ProGoalML modeling is initially platform-independent and then translated in platform-specific documents, but still requires a process engine.

[13] describes a model driven approach for monitoring of BPEL processes. The process is modeled across multiple abstraction levels, each containing an additional monitoring model. KPIs are derived from templates to facilitate reuse. The platform-independent monitoring model is modeled using Eclipse Modeling Framework and transformed to platform specific event and monitoring models. Similar to ProGoalML, the complexity of the underlying platform is hidden from the modeler, but only specific systems may be monitored (IBM Websphere).

In [18] a framework for non-intrusive monitoring is described. Similar to aPro monitoring is separated from execution and monitoring data is acquired by polling for events. A *monitoring policy* containing a process model, input event descriptions (comparable to monitoring schemata) and requirements to monitor (comparable to goals) is used to configure the monitoring framework. While this approach is less intrusive than aPro due to the lack of monitoring stubs, it requires events to be already generated in the execution environment. Creation of a *monitoring policy* is not automated and thus needs multiple documents to be written.

6 Future Work and Conclusion

Work on the prototype is ongoing. As shown in Figure 2 we plan automatic creation of a data warehouse and automatic configuration of a process dashboard already developed. Further research on CEP rule generation will be necessary, as multiple iterations are not handled now and aggregated data (e.g. averages of KPIs) will be shown on the dashboard. Additionally, we will research automatic generation or configuration of monitoring stubs and web services for data collection. Further on, we plan to support process analysis and integrate ABIS to achieve process adaptation (see Figure 1).

In this paper, we gave an overview of the business process monitoring requirements of aPro and designed a goal modeling notation. We showed how the goal model is transformed into a ProGoalML document and how this document is used to create necessary configuration documents. We detailed the creation of XML schemata for measurements and results as well as the creation of CEP rules to transform measurements to results. We implemented these concepts in a prototype and validated them using a test driver. In the future, our findings will be part of aPro, allowing business process optimization and fast setup of process monitoring without extensive technical knowledge.

References

1. Esper - Complex Event Processing, `http://esper.codehaus.org/`
2. Barbon, F., Traverso, P., Pistore, M., Trainotti, M.: Run-Time Monitoring of Instances and Classes of Web Service Compositions. In: International Conference on Web Services, ICWS 2006, pp. 63–71 (September 2006)
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. Bodenstaff, L., Wombacher, A., Reichert, M., Jaeger, M.C.: Monitoring Dependencies for SLAs: The MoDe4SLA Approach. In: IEEE 5th Int'l Conference on Services Computing, pp. 21–29. IEEE Computer Society Press (July 2008)
5. Dardenne, A., van Lamsweerde, A., Fickas, S.: Goal-directed requirements acquisition. Science of Computer Programming 20(1-2), 3–50 (1993)
6. Decker, G., Grosskopf, A., Barros, A.: A Graphical Notation for Modeling Complex Events in Business Processes. In: 11th IEEE International Enterprise Distributed Object Computing Conference, EDOC 2007, p. 27 (October 2007)
7. Decker, G., Overdick, H., Weske, M.: Oryx – An Open Modeling Platform for the BPM Community. In: Dumas, M., Reichert, M., Shan, M.-C. (eds.) BPM 2008. LNCS, vol. 5240, pp. 382–385. Springer, Heidelberg (2008)
8. Gartner: Gartner Reveals Five Business Process Management Predictions for 2010 and Beyond, http://www.gartner.com/it/page.jsp?id=1278415
9. Kavakli, E., Loucopoulos, P.: Goal modeling in requirements engineering: Analysis and critique (2004)
10. Koetter, F., Weidmann, M., Schleicher, D.: Guaranteeing Soundness of Adaptive Business Processes Using ABIS. In: Abramowicz, W. (ed.) BIS 2011. LNBIP, vol. 87, pp. 74–85. Springer, Heidelberg (2011)
11. van Lamsweerde, A.: Goal-oriented requirements engineering: a guided tour. In: Proceedings of Fifth IEEE International Symposium on Requirements Engineering 2001, pp. 249–262 (2001)
12. Luckham, D.C.: The Power of Events: An Introduction to Complex Event Processing in Distributed Enterprise Systems. Addison-Wesley, Boston (2001)
13. Momm, C., Gebhart, M., Abeck, S.: A Model-Driven Approach for Monitoring Business Performance in Web Service Compositions. In: Proceedings of the 2009 Fourth International Conference on Internet and Web Applications and Services, pp. 343–350. IEEE Computer Society, Washington, DC (2009)
14. Neubauer, T.: An Empirical Study about the Status of Business Process Management. Business Process Management Journal 15(2), 166–183 (2009)
15. Object Management Group (OMG): Business Process Model and Notation (BPMN) Version 2.0 (2009), http://www.omg.org/spec/BPMN/2.0/
16. Patig, S., Casanova-Brito, V., Vögeli, B.: IT Requirements of Business Process Management in Practice – An Empirical Study. In: Hull, R., Mendling, J., Tai, S. (eds.) BPM 2010. LNCS, vol. 6336, pp. 13–28. Springer, Heidelberg (2010)
17. del-Río-Ortega, A., Resinas, M., Ruiz-Cortés, A.: Defining Process Performance Indicators: An Ontological Approach. In: Meersman, R., Dillon, T.S., Herrero, P. (eds.) OTM 2010, Part I. LNCS, vol. 6426, pp. 555–572. Springer, Heidelberg (2010)
18. Spanoudakis, G.: Non Intrusive Monitoring of Service Based Systems. International Journal of Cooperative Information Systems 15, 325–358 (2006)
19. Weber, B., Sadiq, S., Reichert, M.: Beyond Rigidity - Dynamic Process Lifecycle Support: A Survey on Dynamic Changes in Process-aware Information Systems. Computer Science - Research and Development 23(2), 47–65 (2009)
20. Weidmann, M., Koetter, F., Kintz, M., Schleicher, D., Mietzner, R.: Adaptive Business Process Modeling in the Internet of Services (ABIS). In: Internet and Web Applications and Services, ICIW (2011)
21. Wetzstein, B., Strauch, S., Leymann, F.: Measuring Performance Metrics of WS-BPEL Service Compositions. In: Fifth International Conference on Networking and Services, ICNS 2009, pp. 49–56 (April 2009)