Data–Driven Process Control and Exception Handling in Process Management Systems

Stefanie Rinderle[1] and Manfred Reichert[2]

[1]Dept. DBIS, University of Ulm, Germany, rinderle@informatik.uni-ulm.de
[2]IS Group, University of Twente, The Netherlands, m.u.reichert@utwente.nl

Abstract. Business processes are often characterized by high variability and dynamics, which cannot be always captured in contemporary process management systems (PMS). Adaptive PMS have emerged in recent years, but do not completely solve this problem. In particular, users are not adequately supported in dealing with real–world exceptions. Exception handling usually requires manual interactions and necessary process adaptations have to be defined at the control flow level. Altogether, only experienced users are able to cope with these tasks. As an alternative, changes on process data (elements) can be more easily accomplished, and a more data–driven view on (adaptive) PMS can help to bridge the gap between real–world processes and computerized ones. In this paper we present an approach for data–driven process control allowing for the automated expansion and adaptation of task nets during runtime. By integrating and exploiting context information this approach further enables automated exception handling at a high level and in a user–friendly way. Altogether, the presented work provides an added value to current adaptive PMS.

1 Introduction

For several reasons companies are developing a growing interest in improving the efficiency and quality of their internal business processes and in optimizing their interactions with customers and partners. Following this trend, in recent years there has been an increasing adoption of business process management (BPM) technologies as well as emerging standards for process orchestration and process choreography [1]. In particular, BPM technologies enable the definition, execution and monitoring of the operational processes of an enterprise.

Currently, one can observe a big gap between computerized workflows and real-world processes [2–4]. This gap is even increasing during runtime, thus leading to unsatisfactory user acceptance. One reason for this drawback is the inability of existing PMS to adequately deal with the variability and dynamics of real–world processes. For many applications (e.g., logistics, healthcare) process execution cannot be fixed in every detail at buildtime [2, 5]. Regarding a delivery process, for example, the concrete tour for the truck is not known beforehand.

Instead, it should be possible to model the processes only at a coarse–grained level and to dynamically evolve these process skeletons (which set out the rough execution during runtime) at the process instance level.

Another drawback arises from the fact that current PMS do not adequately capture (physical) context data about the ongoing process instances. In particular, real–world data is needed for providing (automated) exception handling support. Due to this missing support in exceptional situations, however, users often have to bypass the PMS. As a consequence, computerized processes do not longer (completely) reflect the real-world processes. For dynamic applications like logistics or healthcare, as mentioned, this fact can quickly lead to a non-negligible (semantic) gap between the processes at the system level and those taking place in the real world. To overcome the discussed limitations one of the greatest challenges is to provide automatic support for expanding and adapting ongoing process instances at runtime by avoiding user interactions as far as possible.

In this paper we provide a formal framework for the automated and data–driven evolution of processes during runtime. This includes data–driven expansion of process task nets as well as data–centered exception handling, i.e., process adaptations necessary to deal with exceptional situations are carried out by modifying data structures (e.g., a delivery list of goods). This data change is then propagated to the running process by the concept of data–driven expansion, and not by directly applying (user–defined) changes on the control flow schema of the concerned process instance. This requires availability of data about real–world processes in order to provide automated support. Particularly, we also have to integrate and exploit process context information (e.g., data about physical objects) in order to automatically derive exception handling strategies at a semantically high level. This paper completes our previous work on the ADEPT framework for adaptive process management [6, 7]. On top of this framework we introduce the concepts mentioned above. However, the described approach could be applied in connection with other adaptive PMS as well (e.g., WASA [8]).

In Section 2 we present a motivating example stemming from the logistics domain. A formal framework for data–driven task net expansion is given in Section 3. In Section 4 we discuss exception handling strategies followed by architectural considerations in Section 5. Section 6 discusses related work. We close with a summary and an outlook in Section 7.

2 Motivating Example (and Basic Concepts)

In this section we introduce our running example used throughout the paper in order to illustrate our approach.

2.1 Example Description

As usual, we distinguish between buildtime and runtime aspects of a business process. This is reflected by the separation of process specifications at the type level (buildtime) and the instance level (runtime).

Process Description at Type Level: We use a logistics process, namely the delivery of a set of furnitures to a number of customers by one truck. Let us assume that a planning component has already determined the list of customers who shall be visited by the truck, and that the order of the list sets out the sequence in which the goods are to be delivered to the customers. Consider this information as input to the logistics process depicted in Fig. 1 (via external data element `cust_list`). Based on it a `delivery_list` is built up containing the data needed for delivering the goods (customer name & address, list of the goods to be delivered which have been previously scanned via their bar code). In parallel to this, the truck is prepared. Throughout the processes, the truck position (data element `truck_pos`) is provided by an external tracking component, whose data are continuously updated by a GPS system – we denote this process data element therefore as *external*. In general, such process context information is stemming from physical objects related to the associated process. Examples for physical objects are truck or good with their associated context information location (by GPS system) or barcode.

The delivery list is handed to the truck driver responsible for the tour who then loads the truck correspondingly. The associated `load truck` activity is a *multiple instance activity*, i.e., at runtime it has to be expanded into several activity instances of which each represents the loading of the items for a certain customer (cf. Fig. 2). The number of running instances and the tour itself (described by multiple instance activity `deliver goods` at type level) are also figured out during runtime according to the order set out by data element `cust_list`, i.e., this activity is expanded into several activities each of them describing a single customer delivery. We call this data–driven approach *expansion*. Note that, in addition, `deliver goods` is a complex activity (cf. Fig. 2). This results in a runtime expansion into subprocesses each of them consisting of a sequence of the two activities `unload goods` and `sign delivery report` (cf. Fig. 2). Finally, when the truck driver has finished his tour he is supposed to summarize all single delivery reports collected during the tour in order to create a tour delivery report. Afterwards the truck is returned to the truck company.

Process Expansion at Instance Level: Regarding the expansion of the described multiple instance activities `load truck` and `deliver goods` (see Fig. 1), several issues arise. The first one refers to expansion time, i.e., the time when the multiple instance activities are expanded during instance execution (at process instance level). Basically there are two possibilities: either the expansion takes place when the process instance is started or when the multiple instance activity becomes activated. In Fig. 2, for example, in both cases, the expansion time is set to activity activation time. Therefore, for process instance $I1$, `load truck` has been expanded into three activity instances according to the content of the delivery list. These activity instances describe loading the goods for three customers 1, 2, and 3. By contrast, `deliver goods` has not been expanded yet. For process instance $I2$, however, the expansion of activities `load truck` and `deliver goods` (for customer 1 and 2) has already taken place. When expand-

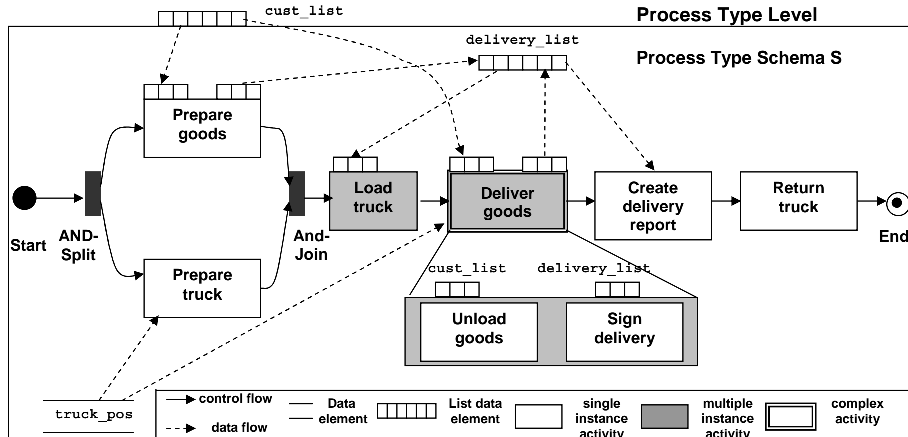

Fig. 1. Logistics Process at Type Level

ing `deliver goods` two activity sequences (consisting of basic activities `Unload goods` and `Sign delivery`) have been inserted at the instance level.

In addition to sequential expansion (as for process instances $I1$ and $I2$ in Fig. 2) parallel expansion will be possible as well if the single activity instances shall be organized in parallel. In addition to this, it is further possible to specifiy in which order the data elements are fetched from the list element responsible for the expansion. Two standard strategies (FIFO and LIFO) are considered in this paper, but others are conceivable as well. More advanced strategies could depend on planning algorithms (especially within the logistics area).

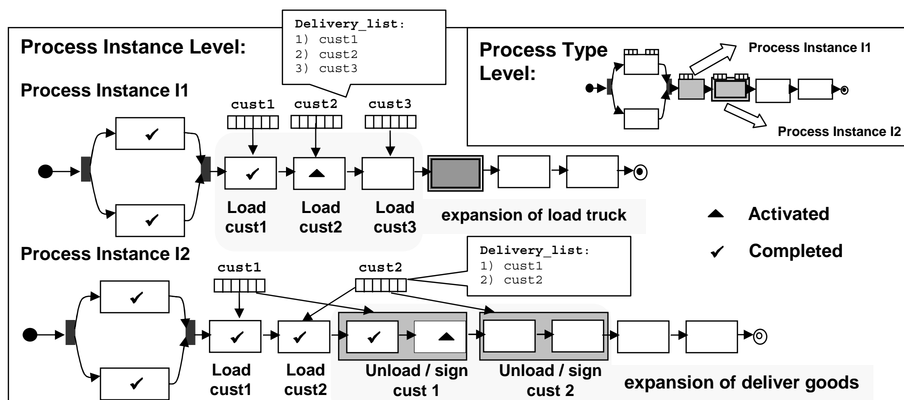

Fig. 2. Expansions of Logistics Process at Instance Level

Changes of the process context and the data structures often require process adaptations. The approach of activity expansion during runtime integrates build-time flexibility into the process meta model[1]. In the logistics process, for example, an additional delivery can be realised by inserting the associated data into the delivery list before activation time of `load truck` and `deliver goods`. This results in the desired process structure and is based on the expansion mechanism and not on the application of an end–user defined control flow change.

2.2 Exceptional Cases

User acceptance can be further increased by strengthening the data–centered view on processes. In addition to data–driven expansion of activities our approach includes process context information, about "physical objects" (e.g., bar code of the goods to be delivered or the truck position determined by a GPS system). Context information can be extremely helpful when dealing with exceptional situations. Assume that a truck crashes during delivery. Then a solution for this problem can be figured out using the context information about the truck position. Other examples for exceptions comprise a wrong truck load or a rejection of the delivery by the customer (e.g., because of quality problems).

Generally, the provision of automatic exception handling strategies is highly desirable for application processes which are "vulnerable" to exceptions. In addition, it must be possible to define such automatic strategies at a semantically high level in order to increase user acceptance. So far, it has been either not possible to deal with exceptional situation at all or users have been obliged to interfere by adapting the affected process instances. However, such modifications require a lot of knowlegde about the process. Using the concept of data–driven expansion instead, exception handling can be (partially) based on the data (e.g., by changing the customer order within the delivery list). Consequently, the system is enabled to automatically transform these modifications into changes of the process structure.

For finding such auomated, high–level exception handling strategies the ability to exploit context data is indispensable. Consider, for example, process instance I2 depicted in Fig. 2. Assume that during the delivery of goods to customer_2 the truck has a breakdown. In this situation it would be not desirable to interrupt the process and roll it back to the starting point since the other customer(s) have been served properly so far. Exploiting context information, in particular truck positions, it could be a more favorable solution to send an alternative truck to the troubled one, pick up the goods, and deliver them to customer_2. Generally, physical context information is helpful for this (and must therefore be somehow respresented at process type level and be gathered at process instance level). Other examples for exceptional situations during execution of the logistics process comprise an incomplete or incorrect loading / unloading of goods, quality defects (e.g., wrong colour of furniture) resulting in such customer refusal, or absence of the customer when the goods are delivered.

[1] We also offer the possibility to adapt process instances ad–hoc by applying instance–specific changes (cf. Section 3)

2.3 Requirements

Altogether, we need a runtime system which allows for a **data–driven** process management. In detail, it must be possible to

- dynamically expand task nets in a data–driven way
- increase process flexibility by automatically translating data structure changes to corresponding process instance adaptations
- integrate context data within the process model
- make use of context information in order to automatically derive exception handling strategies

3 Framework for Dynamically Evolving Process Structures

In this section we present a formal framework for automatically evolving process instances during runtime. The formal foundation is needed in order to present an algorithm for task net expansion, which automatically ensures the correctness of the resulting task net as well as properly working exception handling strategies.

3.1 Process Type Schema

We enrich the standard definition of task nets (like, e.g., activity nets) by introducing the concepts of *list–valued data elements* and the concept of *expansion of multiple instance activities*.

Definition 1 (Process Type Schema). *A tuple S = (N, D, CtrlE, DataE, EC, Exp) is called a process type schema with:*

- *N is a set of activities*
- *D is a set of process data elements. Each data element $d \in D$ has a type $T \subseteq \mathcal{A} \cup \mathcal{L}$, where $\mathcal{A}$ denotes the set of atomic data types (e.g., String, number, etc.) and $\mathcal{L}$ denotes the set of list data types*
- *CtrlE $\subset N \times N$ is a precedence relation (note: $n_{src} \rightarrow n_{dest} \equiv (n_{src}, n_{dest}) \in$ CtrlE)*
- *DataE $\subseteq N \times D \times$ NAccessMode is a set of data links between activities and data elements (with NAccessMode = {read, write})*
- *EC: CtrE $\mapsto$ Conds(D) $\cup$ {Null} assigns to each control edge an optional transition conditions where Conds(D) denotes the set of all valid transition conditions on data elements from D*
- *Exp $\subseteq N \times D \times$ {SEQ, PAR} $\times$ {LIFO, FIFO} $\times$ Time denotes the subset of multi instance activities from N (expanded during runtime based on the specified configuration parameters). For $e = (n, d, mode, str, time) \in Exp$:*
  - *$n \in N$, $d \in D$ with dataType(d) $\subseteq \mathcal{L}$*
  - *mode $\in$ {SEQ, PAR} denotes the multi instantiation mode, i.e., whether the activity instances created at expansion time are carried out in sequence*

or in parallel.
• str ∈ {LIFO, FIFO} denotes the strategy in which list data elements are picked (which is relevant if mode = SEQ holds), and
• time ∈ Time denotes the point in time at which the multi instantiation is carried out; possible configurations are, for example, time = $actT_n$ or time = sT. While the former indicates that expansion takes place when activity n becomes activated, the latter configuration states that expansion is done already at process start time. (More configurations are conceivable, but are outside the scope of this paper).

Data elements can be gathered manually or by exploiting context information, e.g., the barcode of goods (cf. Fig. 1). It is also possible to have context data elements which are continuously adapted (but not read) during process execution (e.g., the truck position obtained by a GPS system in Fig. 1). This context data may be used in order to figure out an exception handling strategy (cf. Sect. 4). The process type schema depicted in Fig. 1 comprises multi instance activites `Load truck` and `Deliver goods`, i.e., we obtain Exp = {(`Load truck, delivery_list, SEQ, FIFO, actT`), (`Deliver goods, delivery_list, SEQ, FIFO, actT`)}. Note that the specification whether a LIFO or FIFO strategy is used only makes sense if the expansion strategy is set to sequential mode.

In addition, we need a set of change operations defined on task nets with precise semantics in order to provide exception handling strategies as, for example, sending a new truck after a truck crash (what would be carried out by inserting an activity `send truck` into the affected task net). Table 1 presents a selection of such change operations. As shown in [6, 2] these change operations all have formal pre– and post–conditions based on which the correctness of a task net is automatically ensured when applying the modifications.

3.2 Process Instances

Based on a process type schema S process instances can be created and started at runtime. Due to the dynamically evolving process structure the particular *process instance schema* may differ from the process type schema the instance was started on. This is reflected by a set Δ_E containing change operations (cf Tab. 1) which may have been applied at different points in time during instance execution and reflect the instance–specific dynamic expansion of S. Furthermore a set of change operations Δ_I is stored which reflects the ad–hoc modifications applied to process instance I (by users) so far. In order to obtain instance–specific schema S_I the merge of the so called change histories Δ_E and Δ_I is applied to S by considering the particular time stamp of each single change operation.

Definition 2 (Process Instance Schema). *A process instance schema S_I is defined by a tuple (S, Δ_E, Δ_I) where*

- *S denotes the process type schema I was derived from*
- *Δ_E denotes an ordered set of change operations which reflect the expansion of S depending on the specified activation time (cf. Fig. 3)*

Table 1. *A Selection of High-Level Change Operations on Activity Nets*

Change Operation Δ Applied to Schema S	**Effects on Schema S**
insertAct(S, X, M_{bef}, M_{aft})	insertion of activity X between activity sets M_{bef}, M_{aft}
Subtractive Change Operations	
deleteAct(S, X)	deletes activity X from schema S
Order-Changing Operations	
moveAct(S, X, A, B)	moves activity X from current position to position between activities A and B
Data Flow Change Operations	
addDataElements(S, dElements)	adds set of data elements dElements to S
deleteDataElement(S, d)	deletes data element d from S
addDataEdge(S, (X, d, mode))	adds data edge (X, d, mode) to S (mode $\in$ {read, write})
deleteDataEdge(S, dL))	deletes data edge dL from S
relinkDataEdge(S, (d, n, [read|write]), n')	re–links read/write data edge from/to data element d from activity n to activity n'
List Data Change Operations	
addListElement(S, d, d_{new}, d_i, d_{i+1})	adds element d_i to list data d between elements d_i and d_{i+1}
deleteListElement(S, d, d_{del})	deletes element d_{del} from list data d
moveListElement(S, d, d_{move}, d_i)	moves d_{move} within list data d after list element d_i

– $\Delta_I = (op_1, .., op_n)$ *comprises instance–specific change operations(e.g., due to ad-hoc deviations).*

The activity set, data set, and edge sets of S_I *(i.e.,* $S_I := (N_I, D_I, CtrlE_I, DataE_I)$*) are determined during runtime.*

Process instance information consists of the process instance schema and the process instance state expressed by respective activity markings. In Def. 3 we add the runtime information (instance state) to the instance schema and present an expansion algorithm based on the process instance state. As described in Def. 2 the deviation of a process instance I from its original process type schema S is reflected by the merge of change histories Δ_E and Δ_I. In particular, the application of the change operations contained in Δ_E to S results in the expanded process instance schema. How Δ_E is determined is described in the following definition. In addition, there may be instance–specific changes Δ_I, for example, applied to overcome exceptional situations. We include these instance–specific changes within Def. 2 since we want to present semantic exception handling strategies which are mainly based on such ad–hoc changes. As provided in the ADEPT framework certain state conditions have to hold when applying change operations at the process instance level in order to ensure a correct instance execution in the sequel. These conditions mainly preserve the history of the previous instance execution. It is forbidden, for example, to delete an already completed activity. For details we refer to [6].

Definition 3 (Process Instance). *A process instance I is defined by a tuple (S_I, N^{S_I}, Val^{S_I}) where:*

- $S_I := (N_I, D_I, CtrlE_I, DataE_I)$ *denotes the process instance schema of I which is determined by (S, Δ_E, Δ_I) during runtime (see Fig. 3 below).*
- NS^{S_I} *describes activity markings of I:*
  $NS^{S_I} : N_I \mapsto \{$`NotAct, Act, Run, Comp, Skipped`$\}$
- Val^{S_I} *denotes a function on D_I, formally: $Val^{S_I} : D_I \mapsto Dom_{D_I} \cup \{$`Undef`$\}$. It reflects for each data element $d \in D_I$ either its current value from domain Dom_{D_I} (for list data elements we assign a list of data values respectively) or the value* `Undef` *(if d has not been written yet).*

$\mathcal{I}_{\mathcal{S}_\mathcal{I}}$ *denotes the set of all instances running according to S.*

Applying the following algorithm (cf. Fig. 3) leads to the expansion of multi instantiation activities during runtime according to the associated data structures. First of all, we determine all multi instantiation activities. For those with expansion at instance start the expansion is executed immediately (lines 7, 8) whereas for activities with expansion at activation time method `expInst(S, ..)` is called when their state changes to `Act` (lines 23 – 27). Method `expInst(S, ..)` itself (starting line 16) distinguishes between sequential and parallel expansion. For sequential expansion, moreover, the fetch strategy for data elements (LIFO, FIFO) is taken into account. The expansion itself is realized by adding change operations (cf. Tab. 1) to change transaction Δ_E.

As an example consider process instance I2 (cf. Fig. 2). At first, it is determined that activities `Load truck` and `Deliver goods` are to be expanded at their activation time (what is specified by {`(Load truck, delivery_List, SEQ, FIFO, actT)`, `(deliver Goods, delivery_List, SEQ, FIFO, actT)`}). Assume that data element `delivery_List = `[`cust1, cust2`] contains data for customers 1 and 2. When the state transition NS^{S_I}(`Load truck`) = NotAct $\longrightarrow$ NS^{S_I}(`Load truck`) = Act is taking place (i.e., the activation time of `load Truck` is reached), this activity is expanded by a sequential insertion of activities `Load truck` using a FIFO strategy. Using the algorithm the changes necessary to realize the expansion are automatically calculated based on the available change operations (cf. Tab. 1):

$\Delta_E := \Delta_E \cup$ {insertAct(S_{I2}, load Truck, {AndJoin}, {Deliver goods}),
addDataEdges(S_{I2}, {(cust1, load Truck, read)}),
insertAct(S_{I2}, load Truck, {load Truck}, {deliver Goods},
addDataEdges(S_{I2}, {(cust2, load Truck, read)}),
addDataEdges(S_{I2}, {(delivery_List, load Truck, write),
(delivery_List, load Truck, write)})}

The expansion of activity `Deliver goods` is carried out accordingly when the activity state of `Deliver goods` changes from not activated to activated.

4 Intelligent Exception Handling

As discussed in Sect. 2.2 backward process recovery (e.g., [9–11]) is not always desirable when an exceptional situation occurs. Therefore we want to exemplarily

```
1  input: S, M^SI   output: Δ_E
2  Initialization:
3  Δ_E = ∅;
4  Exp_sT:= {(n, ..., sT) ∈ Exp};
5  Exp_actT:= {(n, ..., act)} ∈ Exp};
6  n_= is start activity of S;
7  // expansion at process instance start
8  NS^SI(n_=) = Act ⇒ expInst(S, Δ_E, Exp_sT);
9  // expansion during at activation time
10 while (∃ e := (n,d,[SEQ|PAR],LIFO|FIFO],actT)∈Exp_actT with NS^SI(n)= NotAct){
11    if (∃ e := (n,d,[SEQ|PAR],[LIFO|FIFO],actT) ∈ N_actT with state transition
12           NS^SI(n) = NotAct ⇒ NS^SI(n) = Act) {
13          expInst(S, Δ_E, {e});
14    }
15 }
16  // ---------- Activity Expansion method expInst(S, Δ_E, N') ------------
17  Δ = ∅;
18  for e := (n,d,[SEQ|PAR],[LIFO|FIFO], …) ∈ N' do {
19    d:= [d_1, …, d_k]; // d is of list type acc. to definition
20    Δ = Δ ∪ addDataElements{S, {d_1, …, d_k});
21    n_succ, n_pred: direct successor / predecessor of n in S;
22    DE_in:= {(d, n, read) ∈ DataE} \ {d};
23    DE_out:= {(d, n, write) ∈ DataE};
24    // sequential expansion
25    if e:= (n,d, SEQ,[LIFO|FIFO], …){
26          for i = 1, …, k do {
27                n_i:= n;
28                Δ = Δ ∪ {insertAct(S, n_i, {n_i-1}, {n_succ})};
29                // FIFO strategy
30                if e:= (n,d,seq,FIFO, …){
31                      Δ = Δ ∪ {addDataEdge(S,(d_i, n_i, read)})};
32                // LIFO strategy
33                if e := (n,d,seq,LIFO,…) {
34                      Δ = Δ ∪ {addDataEdge(S,d_i,n_k-i+1},read)})};
35          }
36    }
37    // parallel expansion
38    if e:= (n,d,PAR, …) {
39          for i = 1, …, k {
40          Δ = Δ ∪ {insertAct(S, n_i, {n_pred}, {n_succ})};
41          }
42    }
43    for dE = (d,n,read) ∈ DE_in {
44          for i = 1, …, k {
45                Δ = Δ ∪ {addDataEdge(S, (d,n_i,read))};
46          }
47    }
48    for dE = (d,n,write) ∈ DE_out {
49          for i = 1, …, k {
50                Δ = Δ ∪ {addDataEdge(S, (d,n_i,write))};
51          }
52    }
53    Δ_E = Δ_E ∪ Δ;}
```

Fig. 3. Algorithm: Activity Expansion during Runtime

discuss two alternatives for such backward strategies. The first approach refers to data–driven exception handling, the second one is based on exploiting process context information.

4.1 Data–Driven Exception Handling

The expansion of multi instance activities is based on the input data of the particular activity, i.e., a data list setting out the number and order of the activities to be inserted and executed during runtime. This concept provides flexibility since certain process instance changes can be adopted by modifying the input data of multi instance activities what leads, in turn, to changed expansion and execution during rutime. One example is depicted in Fig. 4: Currently, for

process instance I the truck is on the way to deliver the goods of customer2 (the goods for customer1 have been already delivered). Then an exceptional situation is arising since customer2 is not present at home wherefore the goods cannot be unloaded. After receiving the truck driver's call the headquarter figures out to solve the problem by first delivering the goods for customer3 and then try to deliver the goods for customer2 again. This solution elaborated at a semantically high level can now be easily brought to process instance I: Changing the order of a data list associated with customer2 and customer3 (by applying change operation `moveListElement` $(\texttt{S}_I,\ \ldots)$) leads to an automatic adaptation of the delivery order within the process (cf. Fig. 4). Note that this is solely based on data flow changes; i.e., by re–linking the connected data elements `cust2` and `cust3` the delivery order is automatically swapped.

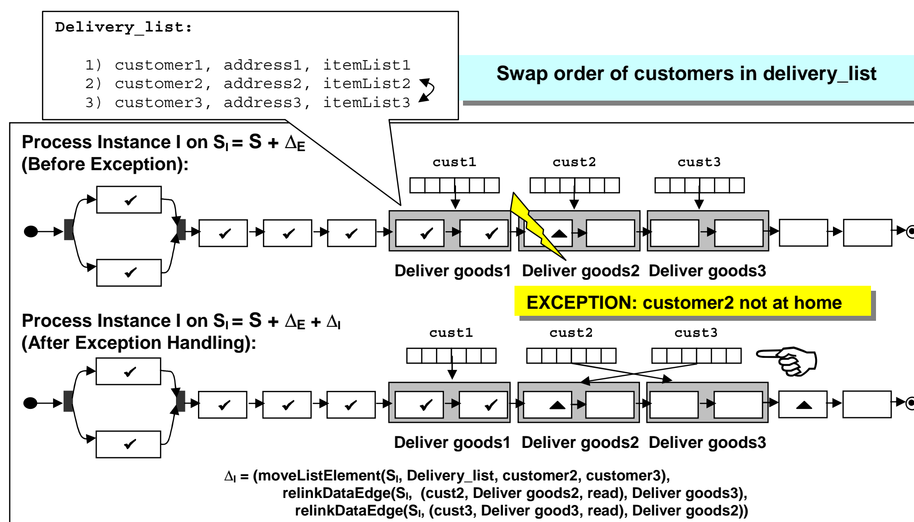

Fig. 4. Data–Driven Change of Delivery Order

For all change operations on data lists like adding, deleting, and moving data elements (cf. Tab. 1), the associated data flow changes (adding and deleting data elements in conjunction with adding, deleting, and moving data edges) can be determined. In this paper, we have exemplarily presented the data flow change operation associated with swapping data list elements. Note that data list modifications as any other change operation can only be correctly applied if certain state conditions hold. For example, for the scenario depicted in Fig. 4 it is not possible to move the list data element for customer1 since associated activity `deliver goods1` has been already (properly) completed. Nevertheless the mechanism of data list adaptations and expansion during runtime provides a powerful way for user–friendly exception handling.

4.2 Exception Handling Using Context Information

In addition to data–driven exception handling, context information can be also useful for dealing with exceptional situations. More precisely, context information can be used in order to derive a reasonable forward recovery strategy, i.e., the application of certain ad–hoc changes to the concerned process instance. Assume, for example, the scenario depicted in Fig. 5 where the truck has a crash during the delivery of the goods for customer 2. Cancelling the instance execution (followed by a rollback) is not desired since the goods for customer 1 have been already delivered properly. Therefore a forward strategy is figured out making use of context data `truck position` which is constantly updated by a GPS system. The truck position can be used to send a new truck to the position of the troubled one what can be expressed by dynamic instance change Δ_I (cf. Fig. 5) comprising the insertion of new activity `send truck`. The new truck then continues the delivery for customer 2 and the execution of process instance I can be finished as intended. Due to lack of space we omit further details.

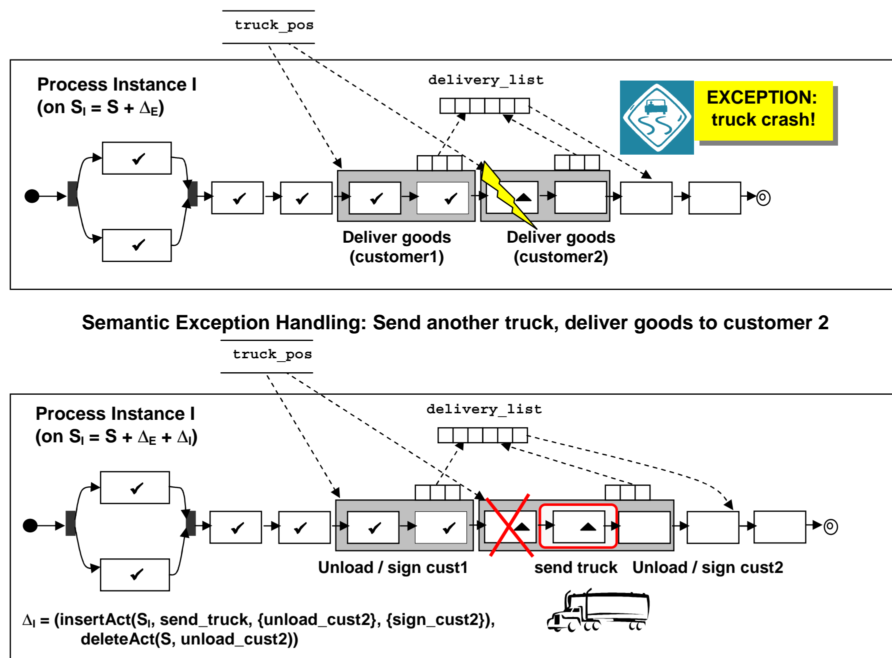

Fig. 5. Exception Handling Using Context Information after Truck Crash

5 Architectural Considerations

We sketch the basic components of our overall system architecture (cf. Fig. 6): Basic to the described features is an adaptive process engine which we have realized in the ADEPT project. It allows for flexible process adaptation at runtime (cf. [12]). In particular, it offers powerful programming interfaces on top of which data-driven expansion of task nets and automated exception handling can be realized. The former feature requires an extended process execution engine (e.g., implementing the expansion algorithm), the latter one requires additional mechanisms for exception detection and handling.

As illustrated determining the position of a physical object is highly relevant for logistics processes. The incorporation of this kind of context information requires an integrated tracking system. Currently, there are various technologies available which can help to trace the position of an object, such as GPS, GSM, RFID, WiFi and more recently UWB [13]. They have different strengths and weaknesses in terms of resolution, availability, cost etc. Moreover, they differ in how the location is being represented, and in environment applicability. An integrated tracking component must abstract from such details and enable seamless and technology-independent tracking outside and inside buildings.

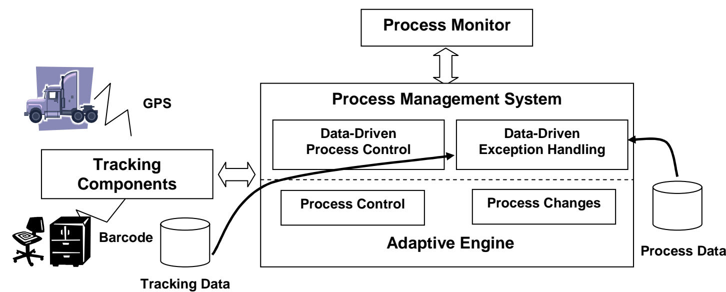

Fig. 6. System Architecture

6 Discussion

Multi instantiation of activities has been addressed by workflow pattern approaches [14, 15]. The most similar patterns are the multi instantiation patterns with and without a priori runtime knowledge as defined in [14]. Although, in [15] the authors suggest a function to compute the number of times an activity is to be instantiated (sequentially or in parallel) the concrete specification of such a function is missing. Therefore the approach presented in this paper can be seen as a first implementation of the multi instantiation pattern without a priori runtime knowledge in practice, i.e., based on associated data structures.

An increase of process flexibility based on a data–centered perspective is offered by the case–handling paradigm [16]. Case–handling enables early review and editing of process data, thus providing a higher degree of flexibility when compared to pure activity-centered approaches. However, it is not possible to dynamically expand process instances during runtime and to use this mechanism for supporting exception handling. A buildtime approach for the automatic generation of processes based on product structures has been presented in [17]. However, no concepts for process expansion during runtime are provided.

Application–specific approaches for automated process changes have been presented in AgentWork [3, 18], DYNAMITE [19], and EPOS [20]. AgentWork [3, 18] enables automatic adaptations of the yet unexecuted regions of running process instances as well. Basic to this is a temporal ECA rule model which allows to specify adaptations independently of concrete process models. When an ECA rule fires, temporal estimates are used to determine which parts of the running process instance are affected by the detected exception. Respective process regions are either adapted immediately (predictive change) or - if this is not possible - at the time they are entered (reactive change). EPOS [20] automatically adapts process instances when process goals themselves change. Both approaches apply planning techniques (e.g., [4, 21]) to automatically "repair" processes in such cases. However, current planning methods do not provide complete solutions since important aspects (e.g., treatment of loops or data flow) are not considered. DYNAMITE uses graph grammars and graph reduction rules for this [19]. Automatic adaptations are performed depending on the outcomes of previous activity executions. Both DYNAMITE and EPOS provide build-in functions to support dynamically evolving process instances.

Context–awareness is also a hot topic in the area of mobile systems, ad–hoc networks, and ambient intelligence (smart surroundings). These approaches can be used as valuable inspiration and input for future research.

7 Summary and Outlook

We have presented a framework for data–driven process control and exception handling on top of adaptive PMS. This approach is based on two pillars: dynamic expansion of task nets and automated support for exception handling using data–driven net adaptation and exploiting context information. The framework for dynamic task net expansion has been formally defined and illustrated by means of an example from the logistics domain. In particular, our expansion mechanism provides a sophisticated way to implement process patterns representing multiple instances with or without a priori runtime knowledge (cmp. Patterns 14 and 15 in [14]). We have also shown how the presented concepts can be used for automated exception handling by adapting data structures. This allows us to handle certain exceptions in a very elegant and user–friendly manner. Finally, further strategies for exception handling based on context information have been discussed. Future research will elaborate the concepts of exception handling based on context information. In particular we will analzye the question how exception

handling strategies can be automatically derived and suggested to the user. Furthermore we want to extend the research on a more data–driven view on process control and exception handling in order to bridge the gap between real–world applications and computerized processes.

References

1. Dumas, M., v.d. Aalst, W., ter Hofstede, A.: Process–Aware Information systems. Wiley (2005)
2. Reichert, M., Dadam, P.: ADEPT$_{flex}$ - supporting dynamic changes of workflows without losing control. JIIS **10** (1998) 93–129
3. Müller, R.: Event-Oriented Dynamic Adaptation of Workflows. PhD thesis, University of Leipzig, Germany (2002)
4. Berry, P., Myers, K.: Adaptive process management: An al perspective. In: Proc. Workshop Towards Adaptive Workflow Systems (CSCW'98), Seattle (1998)
5. Herrmann, T., Just-Hahn, K.: Organizational learning with flexible workflow management systems. In: WS on Organizational Learning, CSCW96. (1996) 54–57
6. Rinderle, S., Reichert, M., Dadam, P.: Flexible support of team processes by adaptive workflow systems. Distributed and Parallel Databases **16** (2004) 91–116
7. Rinderle, S., Reichert, M., Dadam, P.: Correctness criteria for dynamic changes in workflow systems – a survey. DKE **50** (2004) 9–34
8. Weske, M.: Formal foundation and conceptual design of dynamic adaptations in a workflow management system. In: HICSS-34. (2001)
9. Elmagarmid, A.: Database Transaction Models for Advanced Applications. Morgan Kaufman (1992)
10. Schuldt, H., Alonso, G., Beeri, C., Schek, H.: Atomicity and isolation for transactional processes. TODS **27** (2002) 63–116
11. Leymann, F., Roller, D.: Production Workflow. Prentice Hall (2000)
12. Reichert, M., Rinderle, S., Kreher, U., Dadam, P.: Adaptive process management with adept2. In: ICDE'05. (2005) 1113–1114
13. Steggles, P., Cadman, J.: White paper: "a comparison of RF tag location products for real-world applications" (2004)
14. Aalst, W.v., ter Hofstede, A., Kiepuszewski, B., Barros, A.: Workflow patterns. DPD **14** (2003) 5–51
15. Guabtni, A., Charoy, F.: Multiple instantiation in a dynamic workflow environment. In: CAiSE'04. (2004) 175–188
16. v.d. Aalst, W., Weske, M., Grünbauer, D.: Case handling: A new paradigm for business process support. DKE **53** (2004) 129–162
17. v.d. Aalst, W.: On the automatic generation of workflow processes based on product structures. Computer in Industry **39** (1999) 97–111
18. Müller, R., Greiner, U., Rahm, E.: AgentWork: A workflow-system supporting rule-based workflow adaptation. DKE **51** (2004) 223–256
19. Heimann, P., Joeris, G., Krapp, C., Westfechtel, B.: DYNAMITE: Dynamic task nets for software process management. In: ICSE'96, Berlin (1996) 331–341
20. Liu, C., Conradi, R.: Automatic replanning of task networks for process model evolution. In: ESEC'93. (1993) 434–450
21. Wilkins, D., Myers, K., Lowrance, J., Wesley, L.: Planning and reacting in uncertain and dynamic environments. Experimental and Theoret. AI **7** (1995) 197–227