
DYPROTO – tools for dynamic business processes

René Wörzberger* and Thomas Heer

Informatics 3 – Software Engineering,
RWTH Aachen University,
Ahornstraße 55, D-52074 Aachen, Germany
E-mail: woerzberger@i3.informatik.rwth-aachen.de
E-mail: heer@i3.informatik.rwth-aachen.de
*Corresponding author

Abstract: In this article, we present research results about tools for supporting dynamic business processes. This research work has been conducted in a three-year cooperation between our department and an IT service provider for insurance companies. Our partner's process management system (PMS) is rather aligned with static processes, whose structure is not changed at process run time. Therefore, we contribute an approach for obtaining dynamic process execution support based on this static PMS by automatically augmenting existent WS-BPEL process definitions and run time data. Dynamically changeable processes are presented to process participants as graphical models by a process model editor. This editor aids process participants with performing dynamic changes in as much as it is aware of explicit and implicit technical and professional process knowledge and detects violations against this knowledge in dynamically changed process instance models. We delineate how explicit process knowledge can be graphically modelled and exploited in automatic checks using OCL-constrained and integrated meta-models. Checks versus explicit knowledge are complemented by checks versus implicit knowledge which is contained in other process definition and process instance models. These checks include process similarity computations based on graph grammar formalisms and tools.

Keywords: dynamic business processes; tools; WS-BPEL; dynamics; graphical modelling; process knowledge; object constraint language; OCL; meta-models; graph grammars; process similarity.

Reference to this paper should be made as follows: Wörzberger, R. and Heer, T. (2011) 'DYPROTO – tools for dynamic business processes', *Int. J. Business Process Integration and Management*, Vol. 5, No. 4, pp.324–343.

Biographical notes: René Wörzberger received his Doctoral degree in Informatics from RWTH Aachen University in 2010. He took part in a three-year research cooperation between his department and Generali Deutschland Informatik Services GmbH funded by the Deutsche Forschungsgemeinschaft (DFG). His research focuses on tool support for dynamic business processes. He particularly takes existing commercial PMS into account as well as manifold constraints on process models.

Thomas Heer received his Diploma in Informatics from RWTH Aachen University in 2006. Since then, he is a doctoral candidate and partakes in a three-year research cooperation between his department and Comos Industry Solutions©, also funded by the Deutsche Forschungsgemeinschaft (DFG). His research is in tool support for dynamic development processes. Time and cost measurement as well as rights management for dynamic processes and integration of heterogeneous PMSs are among his research interests.

1 Introduction

Work processes tend to be unpredictable. In the following, we stress this property by naming these processes *dynamic processes*. Dynamics can be found in nearly all processes, e.g., highly creative development processes but also business processes. These processes touch the interest of many stakeholders and are at the same time prone to error due to their inherent complexity.

However, there is no common solution that provides best tool support for all scenarios of dynamics in processes. Even worse, many state of the art tools are rather agnostic of

dynamics in processes or just provide little means to suitably support these processes (Weber et al., 2008). For this reason, there are many research projects in this area.

1.1 Preceding works

Our group deals with tool support for dynamic processes for about 15 years. In the past, we mainly focused on dynamic development processes, particularly in the domain of mechanical engineering (Eversheim et al., 1997) and chemical engineering (Nagl and Marquardt, 2008) and developed an adaptable and human-centred environment for

the management of design processes (AHEAD) (Heller et al., 2008). AHEAD particularly provides functionality to cope with evolving and distributed development processes.

Though AHEAD is a working prototype, its dependency on diverse academic frameworks and run time environments impedes its application in industry. Moreover, AHEAD concentrates on the management of human-centred processes. With AHEAD, it is neither possible to execute automated process parts, e.g., activities which are executed automatically by some software system nor does AHEAD fit well into an existing process management tool infrastructure due to its incompliance to existing technical and process management standards.

1.2 Transfer to industry

In the past three years, we transferred concepts of AHEAD into industry. This transfer is two-fold: first, we realised the process management environment for engineering design processes (PROCEED) system, which constitutes an additional layer on top of the commercial chemical engineering tool Comos™ (Heer et al., 2008).

Second and subject of this article, we have built a tool suite for dynamic business processes name DYPROTO on top of the commercial process run time environment WPS (IBM WebSphere Process Server v6.1). In the DYPROTO project, we cooperate with Generali Deutschland Informatik Services GmbH (GDIS), an information technology service provider for the Generali insurance group. Hence, in the DYPROTO project, we shift our focus from development processes to business process in the insurance domain.

1.3 New challenges

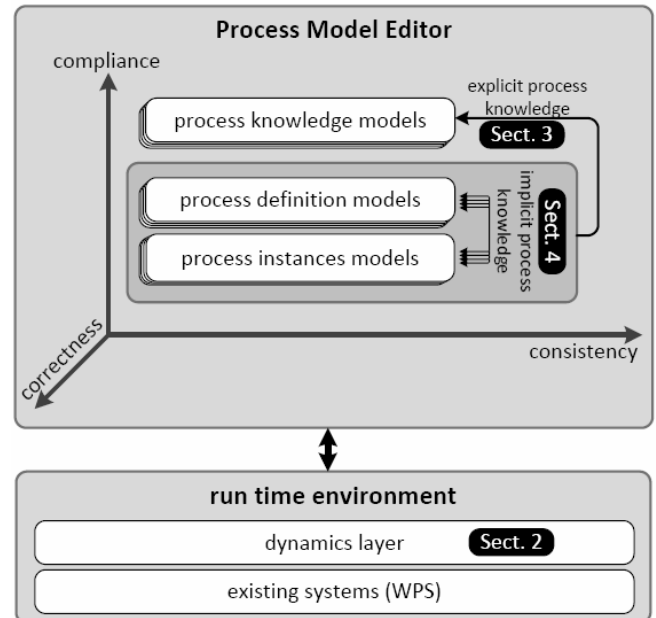
In the DYPROTO project, we faced several challenges which are thrilling from a research perspective. Three of these challenges are dealt with in Sections 2 to 4, which are positioned and related to each other as shown in Figure 1.

First, WPS is a main building block in solutions of GDIS. Thus, DYPROTO had to be integrated with the WS-BPEL-engine WPS. There are numerous academic and some commercial process management system (PMS), which support dynamics changes to running processes to various degrees (Weber et al., 2008). Anyway, replacing the rather ‘static’ WPS by one of these is not an option for our partner due to the investments in WPS-based solutions which have already been made. So, we had to retrofit WPS with capabilities for dynamic changes.

In order to conduct dynamic changes, process participants need to know the entire process structure. Thus, the respective *process instance models*, i.e., models of running processes have to be presented in an editable and graphical form. The same holds true for process models on higher layers of abstraction, e.g., *process definition models*, which precisely describe a certain process type, but also for *process knowledge models* which cover explicit knowledge about processes in a certain domain, e.g., the insurance domain. Thus, DYPROTO comprises a process model editor (PME), which can display process models of various

kinds and supports the editing of these models. Furthermore, the dynamics layer enables dynamic changes but does not provide any means to preserve constraints, which are imposed on process models. Therefore, the PME provides constraint checking functionality.

Figure 1 Challenges and paper structure



We distinguish between the following constraints kinds which are supported by the PME:

- *Correctness* constraints stem from technical necessities formulated in explicit rules. Many of these are well-known from compiler theory but also apply for process models, e.g., process variables have to be written (initialised) before being read. In the worst case, violations to correctness constraints lead to uncontrolled termination of the respective process instance.
- *Compliance* constraints are defined by process knowledge models which specify explicit professional and domain specific process knowledge to which other process models must adhere. For example, process knowledge models might require that certain process activities depend on each other and require a certain precedence regarding their execution order.
- *Consistency* refers to potential contradictions between process models. One can compare a process model to other similar process models in order to reveal inconsistencies, i.e., differences, between them with respect to the occurrences of certain activities and their execution order. Hence, consistency checks use implicit knowledge in other process models.

In this article, we contribute diverse novel and prototypically implemented approaches which deal with the challenges mentioned above. The main concepts are not just valid in our specific project context but applicable for similar problems. We do not provide yet another paradigm

or formalism along with a prototype built from scratch that particularly capture dynamic process changes at process run time. Instead, in Section 2, we delineate how to overcome the insufficient flexibility of WPS by an additional *dynamics layer* that extends the unmodified WPS. Section 3 describes how both *correctness* and *compliance* checks are uniformly realised via an integrative meta-modelling approach. In Section 4, we describe how *consistency* checks are formalised and computed based on *implicit* knowledge which is mined by exertion of graph grammars.

Each of the main Sections 2 to 4 is ended with a discussion and a comparison to related works. A conclusion is given in Section 5.

2 Flexibility for existing systems

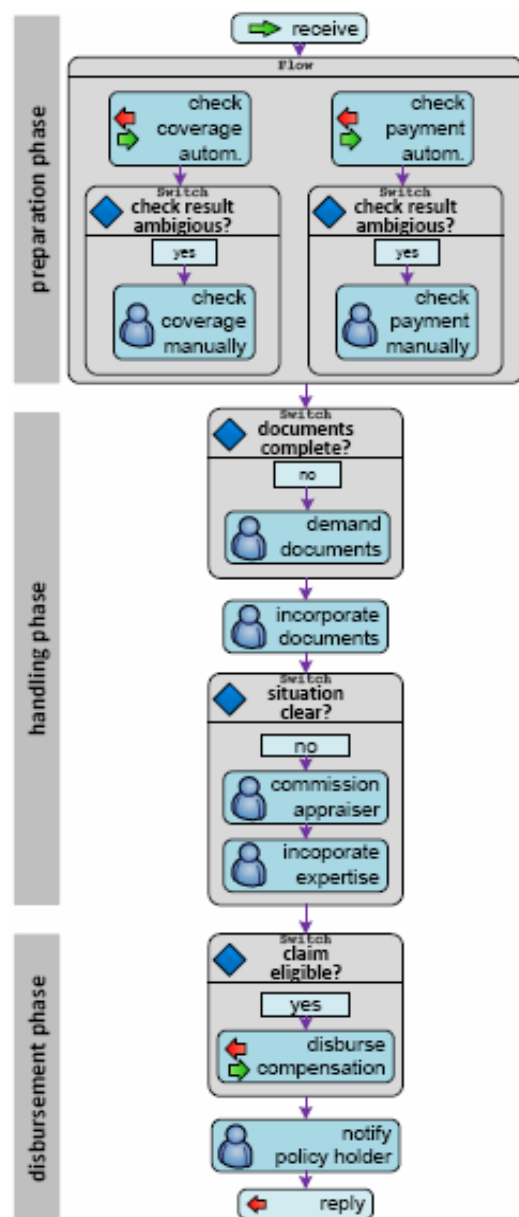
Whenever we talk about processes, we bear in mind some complex material or immaterial system which is changed in a certain way while time passes. This holds true, e.g., for operating system processes which alter data but also for work processes. For instance, in the insurance domain processes for issuing or renewing policies change legal states between the insurance company and the respective policy holders. Supporting work processes by PMS requires formal process models, which particularly describe the steps (activities) that have to be carried out during the respective processes and their sequencing.

In order to be supported by a PMS, a process of a certain type (e.g., settlement of a claim due to rupture of a supply water pipe) has to be modelled in a business process modelling language of an appropriate precision. Such a model is usually called *process definition (model)*. For every process case of the respective type a PMS holds process case specific data called process instance, which refers to a particular process definition. Basically, a process instance contains the execution state of activities as well as process data produced or consumed by the activities.

Figure 2 depicts a simplified process definition of a *claim settlement process* in an insurance company. This process model serves as a running example throughout this article. In the figure, we use a graphical notation for the underlying process definition language WS-BPEL. We refrained from using BPMN as graphical notation due to existing mismatches between BPMN and WS-BPEL (Recker and Mendling, 2006). Instead, our notation is aligned with the commercial WS-BPEL editor IBM WebSphere Integration Developer (WID) v6.1TM and directly reflects the block-structure of WS-BPEL process definitions. Receive and reply activities mark the start and end of the process, respectively. The process consists of three phases: in the *preparation phase*, initial checks are conducted, which verify if the claim is actually covered by some contract between the policy holder and the insurance company and whether the policy holder has fallen behind in payment. These checks are done automatically (👉) but might yield an ambiguous result. In this case, they are also done manually (👤) by some insurance clerk. Since the

activities are placed into a Flow-activity (big box) they can be executed in parallel. Yet, manual activities are executed after automatic ones which are defined using Links (arrows between activities). In the *handling phase*, documents that back the claim are checked for completeness and demanded in addition if necessary. The boxes headed with a ♦ represent Switch-activities, i.e., demand documents is executed if and only if activity documents complete yields no. If the situation is not clear on the basis of the documents, an external appraiser is commissioned, who returns an expertise, which must be incorporated into the internal process. If it turns out in the *disbursement phase* that the claim is eligible, the insurance company disburses the compensation. In any case, the policy holder is notified at the process' end.

Figure 2 Simplified model of an insurance process (see online version for colours)



2.1 Dynamics and flexibility

Dynamics in processes can only be met in process management systems (PMSs) by *flexibility*. However, flexibility can be achieved in different ways.

Build time flexibility is a property of the used process modelling language for process definition models and refers to the constructs a process modeller can use at process build time. For example, WS-BPEL provides constructs like `Flow` and `Link` for concurrency and `Switch` and `While` for optional and iterative activities, respectively.

However, build time flexibility does not suffice for those types of processes where the set of reasonable activity sequences cannot be determined at process build time. By the term *run time dynamics*, we subsume all functions of a PMS that allow for changes in process instance models of running processes like adding additional activities, removing unnecessary activities or reiterating finished activities.

With regard to the example process, an activity ‘determine payee’ might be dynamically added, if the policy holder has no valid bank account. If the policy holder dies during the process and has no legal heirs, disbursement should not take place in any case and should therefore be dynamically removed. Moreover, expertises from an appraiser might be insufficient and require a repetition. The process definition of Figure 2 fails to support the cases mentioned above.

2.2 Retrofitting run time dynamics by simulation

Each software system is either built from scratch or derived from an existing one. This also applies to PMS, which (are supposed to) support run time dynamics. For instance, the Aristaflow-BPM-Suite (Atkinson and Dadam, 2007) has been developed with the initial design goal to offer run time dynamics. However, existing PMS-solutions of our cooperation partner GDIS are based on commercial products, in particular on the IBM WebSphere Process Server v6.1™ (WPS). Since investments to this solution had to be saved, replacing WPS with a new PMS, which a-priorily supports run time dynamics, was not an option. Instead, we pursued an *a-posteriori approach*, i.e., we retained WPS but extended it by a dynamics layer (Wörzberger et al., 2008a). Briefly speaking, this additional layer retrofits WPS with run time dynamics.

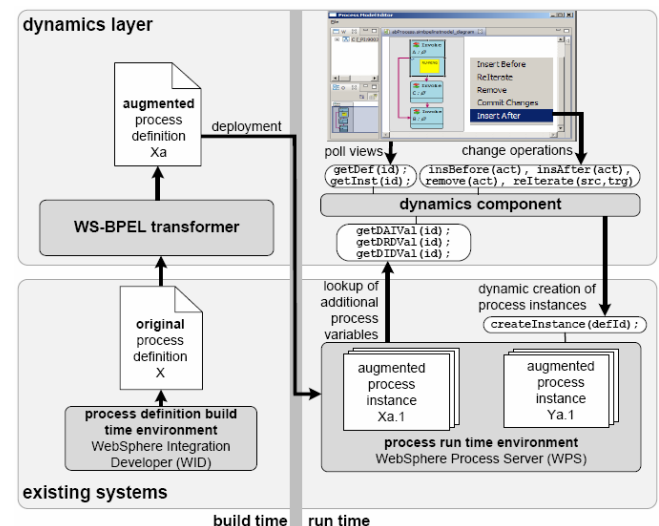
Figure 3 depicts the coarse architecture of the dynamics layer in conjunction with the existing systems WPS and WID. Since the existing systems strictly distinguish between process definitions (process build time) and process instances (process run time) also the dynamics layer consists of two components: the *WS-BPEL transformer* (build time) and the *dynamics component* (run time).

WID is a build time tool that supports process modellers to build process definitions in WS-BPEL, which can be deployed to the run time environment WPS. From an original process definition X, we just demand standard cases. Dynamics simulating structures, i.e., special `Invoke`-,

`Switch`- and `While`-activities, for handling dynamic processes are then automatically added to the process definition X by an XSLT-based WS-BPEL transformer yielding an *augmented process definition Xa*.

At run time, the instance data of the augmented processes is extended by a dynamics component which contains additional process variables that control the behaviour of the additional control flow activities. By default, these additional variables are set in a way that the augmented process behaves like the original one. However, these variables can be indirectly altered at run time by process participants via process instance model manipulations. This simulates the dynamic modification of the running process, e.g., by rerouting the control flow to a newly created process instance. Please note that of course the dynamics simulating structures are hidden from process participants to preserve the impression of a structural modification in the process instance model.

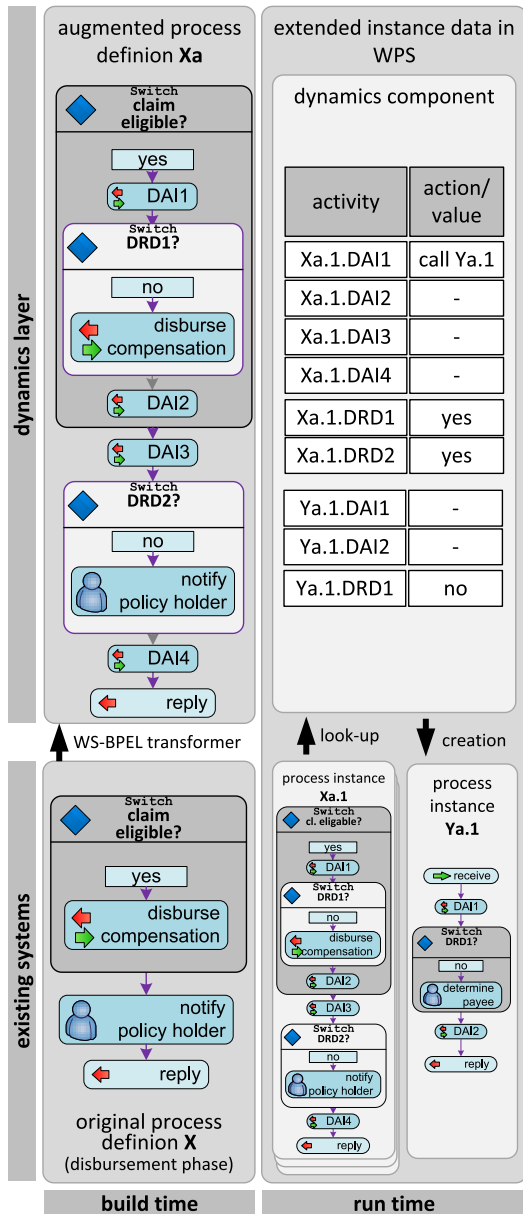
Figure 3 Dynamics layer on top of WPS (see online version for colours)



2.3 Dynamics patterns

The realisation of the dynamics layer is aligned with patterns of modifications to process instance models. *Three dynamics patterns* are essential for run time dynamics: sequentially embedding activities into the control flow of a running process (dynamic adding), conversely detaching existing activities from a process (dynamic removing) and manipulation of the process’ control flow state to allow for repeated execution of activities which have already been carried out (dynamic reiteration). Dynamic adding and removing are included in the ‘adaptation patterns’ identified in the survey work of Weber et al. (2008). Together with dynamic reiteration, we think these patterns suffice for most necessities process participants might have at process run time. They deliberately do not match the possibilities a process modeller has at process build time, e.g., optimisation by parallelisation of activities.

Figure 4 Examples for dynamic adding and removing (see online version for colours)



2.3.1 Dynamic adding

Support for dynamic adding at run time is prepared by the WS-BPEL transformer at build time by adding additional activities to the augmented process definition Xa between sequential activities of the original process definition X. Since technically these additional activities are Invoke-activities of WS-BPEL, we call them *dynamic adding invocations (DAI)*. At run time, DAIs serve as exit-points in an instance Xa.1 of Xa from which the control flow may be routed to a newly created instance Ya.1 of another (augmented) process definition Ya. At this point, Y and Ya must have been pre-modelled and generated, respectively, in order to be available in the pool of addable fragments. For each DAI, the dynamics component stores run time binding information.

In the example of Figure 4, the DAI1 is set to route the control flow to a newly created process instance Ya.1. This simulates the addition of the activity determine payee just before disburse compensation, e.g., because the policy holder has no valid bank account. After Ya.1 is finished the control flow returns to Xa.1 and proceeds with DRD1. The other DAIs expose default behaviour, i.e., they do not have any effect at all. Wörzberger et al. (2008a) provide a more detailed description of dynamic adding.

2.3.2 Dynamic removing

Dynamic removing of activities is realised by additional Switch-activities, i.e., each Invoke-activity of the original process definition X is nested into a Switch-activity by the WS-BPEL transformer. We call this special Switch-activity a *dynamic removing decision (DRD)*. Again, at run time, dedicated variables in the dynamics component control the behaviour of the DRDs.

In the example of Figure 4, the process participant (insurance clerk) might learn during execution of determine payee that the policy holder has died and has no legal heirs. Thus, activity disburse compensation should not be executed irrespective of the claim's eligibility. Furthermore, execution of notify policy holder is obsolete though less harmful (costly) than disburse compensation. Consequently, the process participant indirectly sets the DRD1 and DRD2 of instance Xa.1 to yes. As a consequence, the nested activities are bypassed which simulates a dynamic removal of the activities.

2.3.3 Dynamic reiteration

Sometimes activities have to be reiterated, e.g., commission appraiser due to an insufficient expertise which is revealed during execution of incorporate expertise. For sake of brevity we left out an example for dynamic reiteration in Figure 4. In principle, dynamic reiteration is realised by a While-activity generated by the WS-BPEL transformer at build time, which surrounds the entire process. With regard to the example, at run time all activities succeeding incorporate expertise and preceding commission appraiser are temporarily removed by means of dynamic removing. This simulates a reiteration of both activities commission appraiser and incorporate expertise.

2.4 Discussion

The concepts of our approach were mainly driven by the constraints of our cooperation with GDIS that is we had to build upon WPS. That is why we did not realise run time dynamics with an entirely new PMS although this a-priori approach surely provides higher degrees of freedom with respect to implementation. Instead, our a-posteriori approach simulates run time dynamics with an additional dynamics layer on top of the unmodified WPS. Admittedly, this approach complicates the realisation concepts significantly and hampers the implementation. However,

building upon a commercial PMS also entails the benefit that the overall system inherits the functionality, run time efficiency and robustness of the underlying commercial PMS.

The very last revision 7.0 of WPS entails some new features which pose a step towards process dynamics. These include the possibility for process modellers to specify the migration of a process instance to a new process definition at build time. Moreover, regions can be marked at build time via non-standard WS-BPEL-constructs that surround runtime changeable process parts. Thus, these new concepts complement our approach as they increase flexibility via additional work during process build time instead of dynamic changes at process runtime.

In the description of dynamic adding, we neglected flow of process data that is input or output of activities in the process. If an activity with formal input parameters is added via dynamic adding, the process participant has to assign actual values to these parameters from local process variables. This is supported by a graphical data mapping tool.

Although we realised our approach on top of WPS, its concepts and major parts of its implementation are independent of proprietary WPS-functions. Moreover, the WS-BPEL transformer is strictly WS-BPEL standard conformable, i.e., it does not introduce non-standard constructs. Porting to another WS-BPEL-engine would just require reimplementing of the interface between dynamics component and the respective PMS. Furthermore, concepts of the WS-BPEL transformer can be reused for transformers of XML-languages comparable to WS-BPEL like XPD or BPML.

2.5 Related work

There are several works on flexible PMS, which we relate to our approach in the following. All other works share a common difference compared to ours: the respective prototypes are built from scratch independent of an existing static (commercial) PMS. To do so is appealing from a research point of view as these prototypes can be strongly aligned with existing formalisms. In comparison to other works, we do not provide a novel formalism for run time dynamics but a realisation strategy that a-posteriorly extends static PMS by run time dynamics.

The common difference mentioned above particularly holds true for our predecessor project AHEAD, which is not based on any other (commercial) PMS and therefore lacks certain standard functionalities, e.g., interfacing with standard middlewares. Moreover, AHEAD rather focuses on the management of development processes which typically require continuous detailing and evolution of initially sparse process models instead of on-demand deviations which are typical for business processes.

The ADEPT-approach by Reichert and Dadam (1998) founds on a newly developed, expressive language which is suitable for the modelling of process definitions and process instances. The semantics of this language is precisely

defined via formal mappings to mathematical structures (Reichert, 2000). Run time dynamics are formalised in terms of these structures. Currently, concepts of ADEPT are transferred to a commercial PMS (ADEPT2) (Dadam and Reichert, 2009; Atkinson and Dadam, 2007). Yet, ADEPT2 is implemented from scratch and not based on an existing PMS.

The WIDE-project by Casati (1998) or the ‘pockets of flexibility’-approach by Sadiq et al. (2001) also provide support for run time dynamics. Again, the according prototypes are built independent of existing systems.

Weber et al. (2008) provide a systematic comparison of some academic and commercial systems with regard to flexibility. According to their categorisation, our approach provides support for ‘serial insert process fragment’ and ‘delete process fragment’. These basic patterns can be combined to ‘move’ and ‘swap process fragment’ like Weber et al. delineate. However, parallel and conditional insertion is not supported, yet. Implementing parallel dynamic adding is easy if parallelisation of an added activity *A* is restricted to a single activity *B*. Arbitrary positions for AND/OR-Splits and -Joins would require significant extensions to the WS-BPEL-transformer.

3 Explicit process knowledge

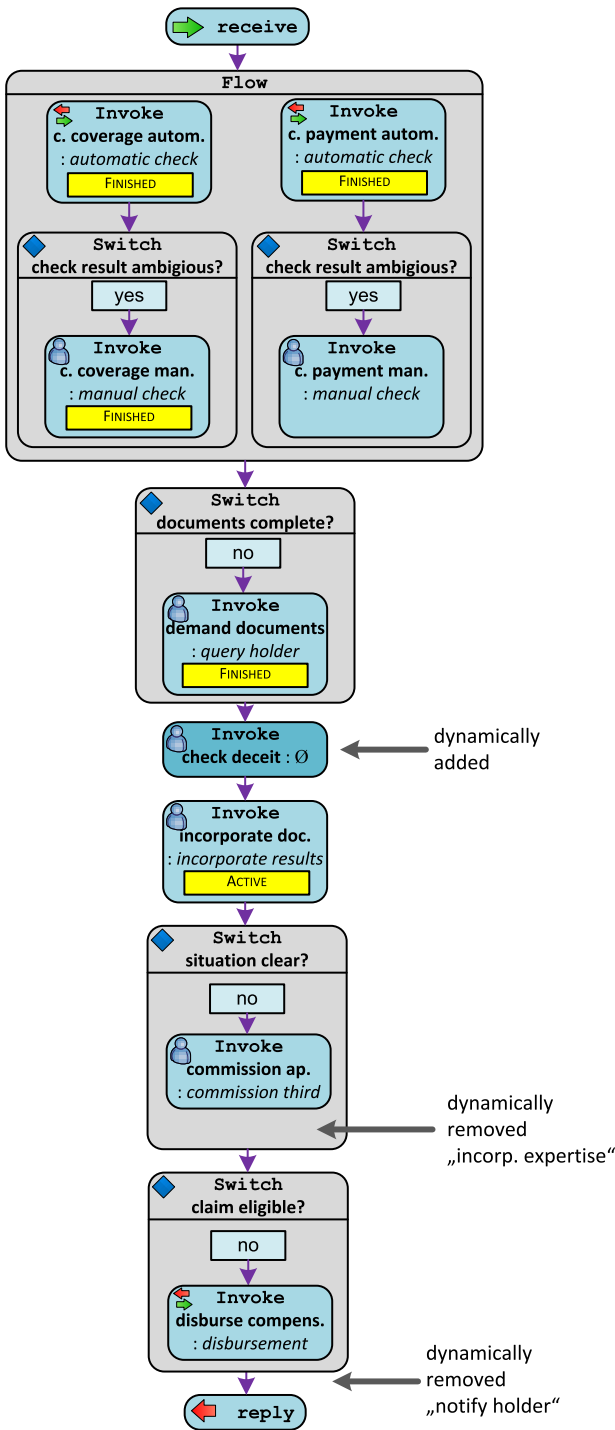
In the last section, we described an a-posteriori approach for extending an existing PMS with run time dynamics by an additional dynamics layer. With run time dynamics, process participants are able to add activities to, remove activities from or reiterate activities in running processes in order to handle cases which have not been modelled in the respective process definition. We concealed that although run time dynamics are essential for dynamic processes they induce new problems. Here, we distinguish between *technical problems*, which relate to violations of *correctness* constraints in process models and might cause breakdowns in PMS, and *professional problems*, which relate to *compliance* constraints enforcing laws or company specific quality standards.

This section deals with concepts for a PME that helps process participants to avoid problems when conducting dynamic changes to a process. The PME is built-in a model driven fashion using rigid integrated meta-models and complex syntactic restrictions. We show that this approach is suitable both for detection of technical as well as professional problems.

3.1 Process models

Process models are useful for communicating information about processes among humans. They are essential for communicating information about processes between humans like process modellers or process participants and PMSs. As mentioned in Section 2, different kinds of process models are relevant in PMS and are described in the following.

Figure 5 Process instance model example with dynamic changes (see online version for colours)



3.1.1 Process instance models

Actual process cases are presented to the process participants by the PMS in the form of *process instance models*. Usually, these models are quite abstract as they just provide a tabular cutout of the process case containing the process case’s state, i.e., the states of the process’ activities. In PMS that provide run time dynamics, process participants also have to know about the whole process structure in order to perform dynamic changes. Hence, in these PMS, process cases are modelled best via graphical process instance

models which reflect the process state but also the structure of a process.

Figure 5 provides an example of a process instance model at a certain point in time during process execution. This modelled process instance was originally instantiated from the process definition of Figure 2 yet deviates from that definition due to some dynamic changes. Activities in this model possess state information, i.e., one of the states waiting, active or finished, which is displayed in boxes within the activities. The absence of a box denotes the state waiting. Obviously, the process has already advanced to activity incorporate documents.

The process instance model in Figure 5 deviates from the process definition model of 2 in three cases: first, an activity check deceit has been added to the process. Second, notify policy holder has been removed from the process. In Subsection 3.2, we will see that these dynamic changes are problematic.

3.1.2 Process definition models

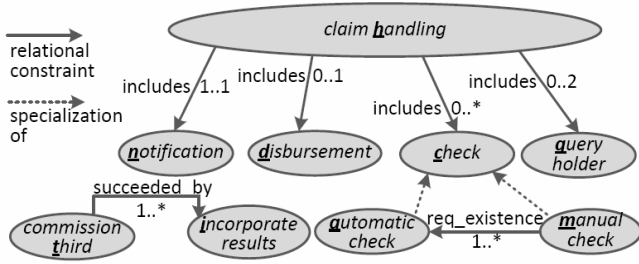
Process definition models model processes of a certain type. An example of a process definition model was already provided by Figure 2. The similarity between the shape of process instance models and process definition models is deliberate. Process instance models contain the control flow structure, e.g., switches, of the process which is associated with a process definition but also reflect the state of a certain process case. Conversely, one can consider a process definition model an abstraction of a concrete process case.

3.1.3 Process knowledge models

Though process definition models already provide an abstraction and thus can be used for arbitrarily many process cases, they are still bound to a certain process type, e.g., settlement of a car damage claim. Therefore, process definitions are unsuitable for expressing general process knowledge of some domain. Process type independent process knowledge requires more abstract kinds of models. For this purpose, we introduced graphical process knowledge models in our approach. These models specify the behavioural aspect of processes of some domain, i.e., they define which activities have to be mandatorily executed, have to be executed in a certain order or depend on each other.

Figure 6 exemplifies a process knowledge model. The ellipses denote *activity types* which pose abstractions of activities in process definitions or process instances. In the following, we abbreviate ‘activity of activity type X ’ with ‘ X -activity’. Activity types can be related to each other with *relational constraint* edges (solid arrows) in order to model structural knowledge about processes in a certain domain. Each edge is directed and defines a constraint for its source activity type. Conversely, a target activity type can satisfy the respective relational constraint. This knowledge particularly constrains processes with regard to the presence and order of activities as described in the next subsection.

Figure 6 Process knowledge model example



3.2 Model constraints

Process model changes and foremost dynamic changes via process instance models might result in problems and therefore need to follow constraints. In the following, we give examples for professional and technical problems with respect to the process instance model of Figure 5.

3.2.1 Professional compliance constraints

With (*professional*) *compliance constraints* we subsume all constraints that stem from laws, company specific regulations or just common sense. Using activity types A, B and relation constraint edges of a certain *kind* we can model a compliance constraints by patterns like $A \xrightarrow[n..m]{kind} B$ and $(n, m) \in \mathbb{N}_0 \times \mathbb{N} \cup \{*\}$, where $*$ denotes infinity. The *kinds* are actually the following which will be described by means of the example in Figure 6:

- *Inclusion.* *Claim handling* processes need to have exactly one *notification*-activity. No notification would leave the policy holder uninformed, twice or more (contradictory) notifications might confuse him. This is expressed by the includes-edge between the respective activity types and the multiplicity 1..1. Similarly, a claim handling process – considered just as a very complex activity – should have at most one *disbursement*-activity to avoid double compensation. The amount of *check*-activities in a claim handling process is unbounded according to the rightmost inclusion constraint. Thus, this *includes*-edge just stresses this unboundedness but could also be removed without side effects.

In general, $A \xrightarrow[n..m]{includes} B$ denotes that an A -process needs to include at least n and at most m B -activities which are executed in any case.

- *Existence.* Disregarding a particular process, occurrence of certain activities requires occurrence of others. For example, a *manual check*-activity requires the existence of 1..*, i.e., at least one *automatic check*-activity in order to prevent human mistakes. This constraint is modelled via the *req_existence* edge from *manual check* to *automatic check*. Generally,

$A \xrightarrow[n..m]{req_existence} B$ denotes that the existence of an A -

activity implies the existence of n to m B -activities in the same process.

- *Succedence/precedence.* Some activities depend on each other and can only be reasonably executed in a certain order. For example, results of a third cannot be incorporated before the third is commissioned. By the *succeeded_by*-constraint in Figure 6, we require that a *commission third*-activity always has to be succeeded by at least one *incorporate results* in the control flow yet not necessarily directly. In general,

$A \xrightarrow[n..m]{succeeded_by} B$ implies that an A -activity is

succeeded by at least n and at most m B -activities in the process. There is also an analogous constraint kind for precedence.

- *Direct succedence/precedence.* A direct-succedence-constraint $A \xrightarrow[n..m]{dir_succeeded_by} B$ is just a tightened succedence-constraint which requires that two activities are executed right after another in a certain order. For brevity, we left out an example for this constraint. Again, there is an analogous constraint kind for direct precedence.

Obviously,

$A \xrightarrow[n..m]{req_existence} B$

holds true if

$A \xrightarrow[n..m]{succeeded_by} B$

applies, which holds true if

$A \xrightarrow[n..m]{dir_succeeded_by} B$

holds true. These implications are analogous for precedence.

Comparable to inheritance between classes in object-oriented programming, one activity type (subtype) can be a specialisation of another one (super type). This is modelled by a *specialisation* edge (dashed arrow). A specialised activity type transitively inherits all constraints of its super types, i.e., all outgoing relational constraints. Furthermore, an activity type also inherits incoming relational constraints and thus may substitute a super type to satisfy a relational constraint.

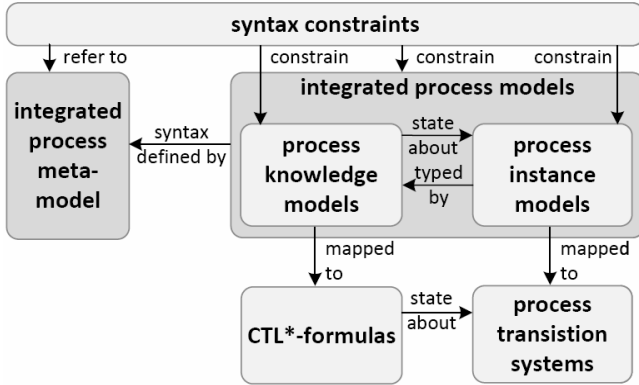
In our approach, activity types can be referenced by activities or entire processes in process definition or instance models in order to give activities (processes) a meaning. The activities of Figure 5 are typed in this manner. From this information, it can be derived that two of the dynamic changes done via the instance model of the claim handling process of Figure 5 violate professional compliance constraints modelled in the process knowledge model of Figure 6. First, the removal of *incorporate results* after *commission appraiser* violates the *succedence*-constraint from *commission third* to *incorporate results*. Second, the removal of the *notification*-activity

contradicts the inclusion constraint from *claim handling* to *notification*.

3.2.2 Formal semantics of compliance constraints

Since the expressions in process knowledge models only refer to the existence and ordering of activities, their semantics can be formally defined. The lower right side of Figure 7 shows how this is done conceptually. We use the superset CTL* of linear temporal logic (LTL) and computational tree logic (CTL).

Figure 7 Conceptual architecture for process model checks



The semantics of CTL* formulas can be defined with respect to a formal structure. We define CTL*-semantics with regard to process transition system (PTS), i.e., CTL*-formulas make statements about PTS. A PTS is a $\{\mathcal{R}, \mathcal{L}\}$ -structure with universe \mathcal{S}_p . Here, \mathcal{S}_p denotes the set of all process states, where a process state is just a composite state containing all activity states. $\mathcal{R} \subseteq \mathcal{S}_p \times \mathcal{S}_p$ is the binary and total transition relation between process states. $\mathcal{L}: \mathcal{S}_p \rightarrow 2^{\mathcal{AP}}$ maps process states to sets of atomic propositions \mathcal{AP} . In our case, atomic propositions are pairs, i.e., $\mathcal{AP} = \mathcal{AT} \times \mathcal{S}_a$ where \mathcal{AT} denotes the set of activity types and $\mathcal{S}_a = \{\text{Waiting}, \text{Active}, \text{Finished}\}$ possible activity states. With respect to a PTS, we can apply the usual CTL*-semantics definitions as provided (e.g., Clarke et al., 1999).

As depicted in Figure 7 process instance models are mapped to PTS. The semantics of process knowledge models can then be formally defined by mapping the language elements to CTL*-formulas.

For instance, we can formally define the semantics of:

$$A \xrightarrow[1..*]{req_existence} B$$

by the CTL*-formula:

$$\mathbf{A}(\mathbf{F}(A, \text{Active}) \rightarrow \mathbf{F}(B, \text{Finished}))$$

where boldface letters are path quantifiers and temporal operators. With regard to a process the CTL*-formula must be read as follows: “For all (A) process execution paths: if there is a state somewhere on the path (F) where an

A-activity is executed, i.e. Active, then this implies that there must also be a state on the path (F) where a B-activity has been executed, i.e., is Finished”. Analogously, we can formally define the semantics of the other process model knowledge elements. Please note, that it is generally not possible to do this independently of a certain multiplicity like 1..* in the example above, i.e., one cannot define the semantics of $A \xrightarrow[n..m]{req_existence} B$, in just one CTL*-formula independent of $n..m$.

3.2.3 Technical correctness constraints

Process models of a certain kind have to adhere to complicated syntactic correctness constraints which are independent of other models. Violations of these constraints usually lead to technical problems within the respective PMS like premature termination of a process instance or deadlocks. Hence, we call these constraints (*technical correctness constraints*).

Correctness constraints require, e.g., that process variables have to be initialised by some activity before they can be read (cf. Wörzberger et al., 2008b) or that process variables should not be written by concurrent activities in order to avoid lost updates. Moreover, dynamic changes in process instance models should not introduce unreachable activities. This is the case in the process instance model of Figure 5. Here, check deceit has been added during execution of incorporate documents. Obviously, in this process instance model check deceit will never become Active. Since this problem stems from the control flow semantics of the process definition language but not from external process knowledge, we consider this an (internal) incorrectness of the model. Also, process knowledge models have to obey to correctness constraints, e.g., cyclic precedence or specialisation relationships have to be excluded to avoid unsatisfiable compliance constraints.

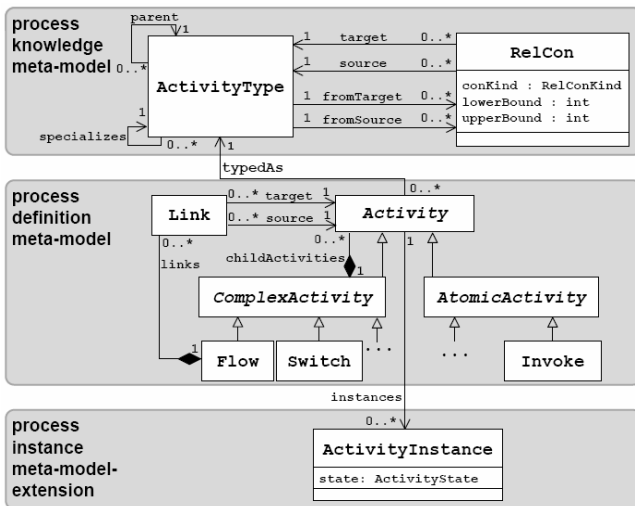
3.3 Syntax checks of integrated process models

Although we use temporal logics in order to formally define the semantics of process knowledge models we do not use the formulas in some model checker to detect the violation of process model constraints. Instead, violations of constraints are detected by checks of the process models’ static syntax. This is feasible since process definition models and process instance models specify the behaviour of a process (of a certain type). Hence, they cover all information necessary to do the constraint checks. The upper half of Figure 7 summarises the conceptual connections which are relevant in the following.

Before we can formally define complex syntactic rules on process models, we need to formally define the basic syntax, i.e., language elements and their possible relations and compositions. Since we are dealing with graphical languages, we use simplified graphical class diagrams as meta-models for this task. Figure 8 depicts the relevant parts of these meta-models. The process knowledge meta-model basically consists of two meta-classes. It defines that

ActivityTypes can be connected by relational constraints (RelCon), whose attribute conKind ranges over the kinds described in Subsection 3.2 and the attributes lowerBound and upperBound constitute the multiplicity of a relational constraint. The process definition meta-model defines Activity, which can be either an AtomicActivity like Invoke or a ComplexActivity like a Flow or a Switch. In a Flow, two activities can be connected by a Link to restrict possible execution sequences. The process instance meta-model is not a meta-model on its own but extends the process definition meta-model by a meta-class ActivityInstance for incorporating activity states ranging over {Waiting, Active, Finished} in process instance models. Please note that an Activity can be associated with arbitrarily many ActivityInstances. This is reasonable because activities can be nested in Switch-or While-elements and thus be executed never, once or arbitrarily many times.

Figure 8 Integrated process meta-model



As depicted in Figure 8, the meta-models are not isolated but refer to each other. Process instance models are instances of the combined meta-model consisting of the process definition meta-model and the process instance meta-model. Furthermore, an Activity can be typed with an ActivityType by the reference typedAs. Consequently, a process instance model together with a process knowledge model containing its activity types form an integrated syntactic structure.

Meta-models define the basic syntax, e.g., relations and compositions, of a graphical model. However, they are to inexpressive to capture more sophisticated syntactic constraints. Therefore, we cannot cover the model constraints of Subsection 3.2 just by means of the meta-models. Instead, we augment the meta-models by textual object constraint language (OCL v2.0)-expressions, which tighten the syntactic constraints on process models. In this way, correctness constraints can be formalised in OCL-expressions that refer to a single process meta-model. Compliance constraints can be expressed in OCL by expressions, which refer to the integrated meta-model.

Hence, both sorts of constraints are handled by the same method.

Listing 1 exemplifies a correctness constraint for a process knowledge model. This constraint evaluates to false, if an ActivityType is an ancestor of itself with regard to specialisation. In Wörzberger et al. (2008b), we exemplify another correctness constraint in OCL which constrains process definition and instance models.

Listing 2 provides an example of an OCL-expression that defines an invariant which holds true for an Activity if and only if the activity complies with the precedence relationship of its activity type. This expression uses the elsewhere defined OCL-definitions allAdjPrec, matchBounds and allSuccs. Briefly speaking, the expression compliesPrecedence navigates from an activity (self) to the activity type, collects via allAdjPrec all precedence relationships. Then it checks for all these relationships via matchBounds and allSuccs if the precedence relationships are satisfied in the process instance model of the checked activity. Wörzberger et al. (2008b) provides a more detailed description of this and other OCL-expressions.

In summary, for the specification of correctness constraints as well as compliance constraints OCL-expressions are used. However, while the OCL-expressions for correctness constraints each explicitly define a specific constraint, the OCL-expressions for compliance merely specify how process definition or instance models are to be checked against process knowledge models.

Listing 1 Cycle-freeness correctness constraint for specialisation in process knowledge models

```
context ActivityType
inv cyclefreeSpecialization:
not self->closure(parent)->includes(self)
```

Listing 2 Precedence compliance constraint for process instance and definition models

```
context Activity
inv compliesSuccedence:
self.typedAs.allAdjPrec->forall(rc1 |
rc1.matchBounds(self.allSuccs->select(
typedAs = rc1.target)->size() ))
```

3.4 Implementation remarks

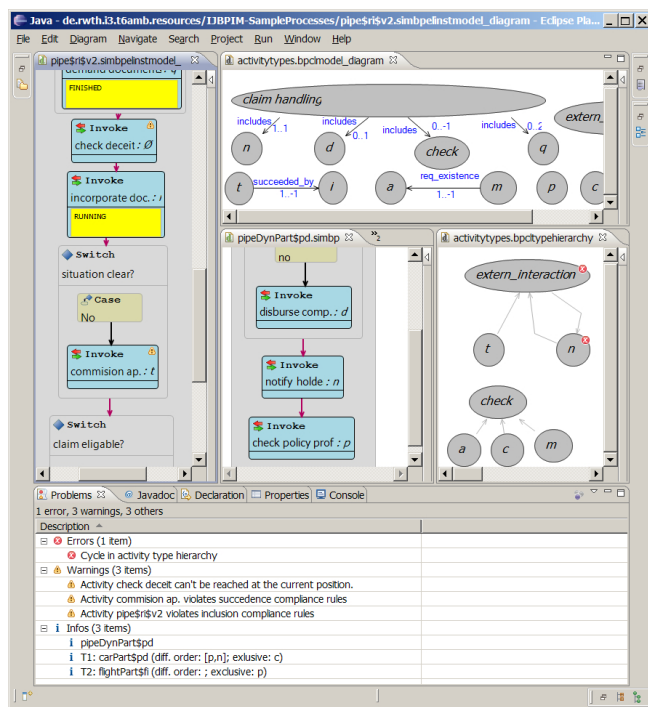
Changes to process models are not directly propagated to the dynamics component (above WPS) but are at first local to the respective instance of the PME. After a model change, correctness and compliance checks are manually triggered by a process participant via an according menu item in the PME.

Correctness violations are considered severe as they can cause technical failures in the run time environment. Thus, a change in a process instance model cannot be committed to the dynamics component a correctness violation have been

found. Detected compliance violations are tolerated since their violation might be intentional. Thus, changes in process instance models that just cause compliance violations can be committed to the dynamics component.

Figure 9 depicts a screenshot of the PME. The entry in the Errors-section hints to a correctness violation in the process knowledge model, which is a cycle in the specialisation-hierarchy in this case. Please note that specialisation hierarchies are modelled in dedicated views (diagram on the right) separated from relational constraints (diagram on the top). The entries in the Warnings-section are compliance violations as discussed in Subsection 3.2. They refer to the diagram on the left, which depicts the process instance model of Figure 5. The entries in the Infos-section are discussed in 4.8.

Figure 9 Screenshot of the PME (see online version for colours)



Due to the syntax-based constraint checking approach, the PME, which actually checks compliance and correctness of process models, could be implemented in a model driven fashion. The PME is basically generated from the abstract syntax definition specified in eclipse modelling framework (EMF) meta-models and concrete syntax definition specified in graphical modelling framework (GMF)-models. Furthermore, we made use of the Eclipse OCL framework which provides an interpreter for our OCL-expressions. Details about the generation process are given by Wörzberger and Heer (2008).

3.5 Discussion

The purely syntax-based constraint checking approach exhibits some advantages and drawbacks. A drawback surely is the need for typing activities before they are properly recognised in compliance checks, i.e., compliance checks fail if activities are not properly typed or not typed at

all. For example, if the activities with type *automatic check* in Figure 5 were untyped, a false compliance violation would be detected due to the *req_existence*-edge in the process knowledge model in Figure 6. Conversely, if activity commission appraiser was untyped, the compliance violation stemming from the *succeeded_by*-edge in the process knowledge model could not be found.

The dilemma can be described as follows. On the one hand, the usage of domain specific activity types for process modelling still meets many obstacles in practice. These are among others the establishment and maintenance of a well understood and organisation-wide process knowledge model, and the possible over specification of process models where almost every activity has its own activity type. The problems even grow when process knowledge models not only cover the behavioural aspects but also organisational or data-related aspects. On the other hand, the definition of compliance rules for process models seems to be infeasible without relying on activity types. To the best knowledge of the authors there is no feasible approach for defining general compliance rules which apply to many different process models where the activities are only distinguished by their names. Relying on the comparison of activity names is a bad idea as there might be many synonyms (e.g., 'invoice', 'invoicing', and 'billing') or homonyms and the like. Since the obstacles for the application of activity types for process modelling are hard but resolvable, but the abandonment of activity types makes compliance constraint modelling practically impossible, we decided to build our approach on typed activities.

One of the main advantages is that the implementation can be done model-driven with frameworks mentioned in Subsection 3.4. First, handcrafted Java-code does not exceed one single Java-class that initialises the OCL-interpreter. Second, although the process definition and instance meta-models and therefore the OCL-expressions are aligned to WS-BPEL, they can be easily adapted to other languages like BPMN without further Java-programming effort. Third, the process definition model meta-model also covers data flow, which is neglected in Figure 8 for sake of brevity. Wörzberger et al. (2008b) show how data flow can be utilised to formalise constraints like the one that process variables have to be initialised before being read. Fourth, OCL-expressions are always evaluated on a certain model element. Therefore, constraint checks yield not just a Boolean value but point to the respective element which is likely to be blamed. This is indicated by an error marker on the element in the PME. Admittedly, this does not work optimal in all situations, e.g., the violation of the correctness constraint in Listing 1 leads to an error marker on every *ActivityType* in the cycle. Fifth, OCL-expressions can be efficiently evaluated compared to other techniques like model checking. This is crucial for process instance models since time consuming checks due to changes to these models would impede efficient process execution. Sixth, the approach uniformly treats technical (intra-model) and professional (inter-model) constraints.

3.6 Related work

Works that are related to correctness and compliance checks versus explicit process knowledge as presented in this section are from diverse fields. In the following, we group these works by terms presented in this section: correctness and compliance.

3.6.1 Correctness

There are plenty of works aiming at preserving correctness of process models. Most of them deal with correctness of process definition models, i.e., with correctness at design time. For example, Mendling and van der Aalst (2007) use graph reduction techniques to verify correctness of event-driven process chains-models (EPC) for generic process models. Since the reduction works just on the syntactic structure of the process model, the computational complexity surely is comparable with our checks.

Verbeek et al. (2001) describe the process definition analyser Woflan. Woflan works on the Petri net class WF-nets. Thus, other languages have to be mapped on this language in order to be analysed. This can be done for WS-BPEL as sketched by Verbeek and van der Aalst (2005). WF-nets are then analysed for certain correctness properties by Woflan based on a certain representation of their state space. This approach is exhaustive yet computationally expensive.

The verification of correctness of distributed WS-BPEL processes is presented by Bianculli et al. (2007) and Gravel et al. (2007). In these works, WS-BPEL processes are translated to an input language for a model checker. Similarly, Koehler et al. (2002) map process models to finite state automata (FTAs) and use a model checker to verify, e.g., the termination of a process. We neither apply model checking nor consider correctness of distributed process models so far.

Reichert and Dadam (1998) formalise numerous transformations by calculi on ADEPT control flow graphs – comparable to our process instance models – which realise dynamics patterns. These transformations are correctness preserving, i.e., applying a transformation like dynamic adding to a control flow graph again yields a correct graph. In contrast to that, our PME allows for flaws in the process models but marks them and prohibits their propagation to the WPS if necessary.

3.6.2 Compliance

Preceding works of our group in the AHEAD-project also dealt with compliance of process models (Schleicher, 2002). We continued some of the basic ideas, e.g., the interrelation of different process model layers. Yet, we had to significantly adapt and extend the concepts in order to account for industry standards like WS-BPEL and to avoid some limitations like the necessity for recompilation of the AHEAD-prototype after changes in the process knowledge models.

Ly et al. (2008) provide a set of major requirements for compliance checking process models. Parts of these are actually satisfied by our approach: We provide a user friendly yet formal language (req. 1). ‘Constraint organisation’ (req. 2) is at best partially fulfilled as process knowledge models are stored and versioned in flat files. We just support implementation independent constraints (half of req. 3). ‘Support for life time compliance’ (req. 4) is partially given since during checks we treat process definition models just like process instance models being at the very beginning of execution. Conflicting changes in definitions and instances are not treated (req. 5). The check feedback is intelligible (req. 6) to some extent as it contains violation details. Compliance constraints are naively overridable (req. 7) by ignoring found violations. Traceability (req. 8) of compliance checks (persistent logging) is not implemented.

Process models in the DECLARE project by Pesic et al. (2007) resemble our process knowledge models with regard to the level of abstraction. However, DECLARE pursues a completely different approach since the ‘declarative’ process models are interpreted and directly used for process execution support whereas our process knowledge models just impose compliance constraints onto executable and executing models. Although the DECLARE-approach is appealing inasmuch as DECLARE-models provide much flexibility by design, we had to account for the rather imperative paradigm of WS-BPEL due to our project constraints.

Governatori et al. (2006) demonstrate a way to formalise the compliance relationship between BPMN process models and natural language business contracts. The latter are mapped to derivation rules of formal contract logic (FCL) the former to FCL-sequences. Then, compliance is equivalent to derivability of sequences in the FCL-calculus. This approach focuses multilateral process models with time-related escalations, which we have not dealt with so far.

Rozinat and van der Aalst (2008) show that there is also a compliance relationship between process instance data and a process definition model. This sort of compliance runs the risk of being violated in situations where a process definition model does not control a process from the very beginning. However, this does not apply for our project setting.

Interestingly, there are several groups that use model checking methods in order to check compliance. Förster et al. (2007) model process knowledge in a model similar to ours. These models are translated to LTL-formulas. LTL formulas are then evaluated on transition systems which represent the state space of a process definition model. Work of Awad et al. (2008) is similar to that, yet, they target BPMN-models instead of UML activity diagrams and use a variant of LTL. Compliance checking of WS-BPEL process definitions via model checking is presented by Liu et al. (2007). Again, the process knowledge models are translated to LTL and a transition system is generated from a WS-BPEL process definition.

In summary, our approach differs from related approaches insofar as we uniformly address correctness and compliance checks with one syntax-based method. Certainly, generating and querying a process model's state space allows for detection of even very subtle errors which is beyond the capabilities of our approach. However, we pay this price to gain computational efficiency which is crucial particularly when checking process instance models during process execution. Adaptation to other process modelling languages causes effort either way: one has to adapt the meta-models and OCL-constraints in our case or the transformers to LTL and generators of transition system in the approaches mentioned above.

4 Implicit process knowledge

Process knowledge models allow for explicit modelling of professional process constraints which can be utilised for compliance checking process definition and instance models. It is a downside of these explicit models that they cause extra work and might be incomplete. At the same time, process definition models and process instance models of running or finished processes themselves implicitly contain process knowledge and are available anyway. Therefore, we complement the checks of a *testee*, i.e., a modified process instance model versus explicit process knowledge (as described in Section 3) by checks versus implicit knowledge derived from other process definition and instance models. Since checks versus implicit knowledge refer to potential contradictions, i.e., inconsistencies of models of the same kind, we name these checks *consistency checks*.

In this section, we describe how differences between a testee process model and a certain set of other process models can be detected and compactly presented to a participant working on the former. Here, differences between process models refer to (dis-)similar behaviour rather than to syntactic, i.e., structural similarity since even structurally dissimilar process models might exhibit similar behaviour as recognised (e.g., by van der Aalst et al., 2006). First, we describe the structure of the knowledge base M_{kb} , i.e., the set of available process models excluding process knowledge models, which are not regarded in this section. Second, we define for a testee a subset of the available process models containing those models (testers) which are sufficiently similar to the testee. This is necessary since a detailed analysis of two process models cannot yield meaningful results if the process models are not similar to some degree, i.e., are 'apples and oranges'. Third, we describe how differences between the testee and the models of this set are detected and compactly presented to the process participant/modeller who is working on the testee.

4.1 Structure of the implicit-knowledge base

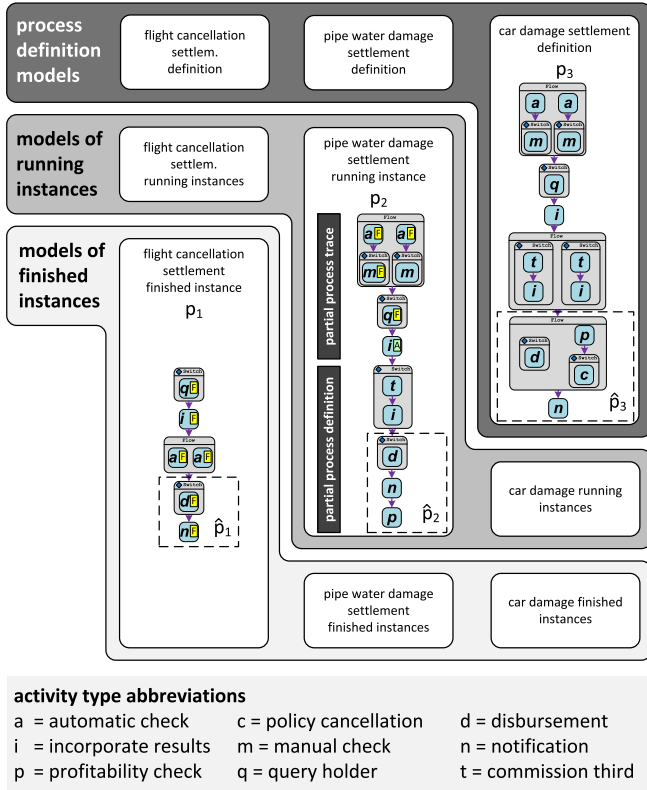
The knowledge base for compliance checks is explicitly modelled, i.e., it consists of process knowledge models with

dedicated syntax and semantics (cf. Subsection 3.1). In contrast, the knowledge base for consistency checks M_{kb} consists of different kinds of models. As depicted in Figure 10, it is composed of the three technical model classes explained below. Please note that in this figure the activity labels just contain abbreviations of the respective activity type in order to keep the figure clear. Furthermore, the regions bounded by the dotted lines in the example models p_1 to p_3 just denote process model parts \hat{p}_1 to \hat{p}_3 . We will use these less complex models in the next subsections for sake of simplicity.

- *Process definition models* carry implicit knowledge about which activities are to be carried out for a certain process type. Normally, they do not narrow processes to a single possible sequence of activities but contain information about which activities are to be executed alternatively, sequentially or in parallel.
- *Models of finished process instances* represent process traces. We consider a process trace a time-related succession of activity finishing events in the past. One process trace reflects in which sequence steps of a single process case have been carried out. For example, in Figure 10 the depicted instance of flight cancellation settlement represents a past process. Hence, all activities are in state Finished.
- *Models of running process instances* constitute a mixture of the above mentioned model kinds. On the one hand, they have a determined past like finished process instances, i.e., a partial process trace of past activity events. Since process instance models essentially are process definition models plus activity states, on the other hand, the running process instance models have not a fully determined future which is modelled by a partial process definition within the instances. For example, the instance of pipe water damage settlement running instance has proceeded to \hat{p} -activity. For this running instance (a certain order of), the finished activities constitute the partial process trace. The structure in the lower half contains a partial process definition representing the remaining future of this instance for which it is not determined if, e.g., a disbursement takes place or not. Many domains – in particular the insurance domain – have many long running processes; this mixture class can therefore not be neglected.

Figure 10 suggests that each process instance is in line with a process definition. However, please recall that models of process instances might be structurally different from any of the process definition models due to dynamic changes. This is the case for the running instance of Figure 10 where a p -activity has been dynamically added. Neither the past process trace of a process instance model nor the potential futures might therefore be covered by a process definition model.

Figure 10 Process model classes with implicit knowledge (see online version for colours)



4.2 Process trace sets

Each process model is associated with a *process trace set* containing exactly those process traces which the respective process model can produce. Like for compliance checking we refer to activity types instead of names. Furthermore, we refer to activity type names by one letter abbreviations in the following, e.g., *query holder* is abbreviated with *q* (cf. Figure 10). So, a process trace of a process model is represented by an activity type name sequence.

Definition 1: (Process trace set). Let M be the set of all possible process models, T the set of all activity types and T^* the set of all finite activity type name sequences. The process trace set is a mapping:

$$c : M \rightarrow 2^{T^*}$$

which maps a process model to the set of all process traces possible in that model.

Due to the complex semantics of the process models the formal definition of the mapping c is also complex and thus omitted here.

For example, with regard to the process models of Figure 10 the following holds: $c(p_1) = \{ \langle qiaadn \rangle \}$, i.e., this process model trivially just has exactly one process trace. Furthermore, $c(p_2) = \{ \langle aamqinp \rangle, \langle aamqitnp \rangle, \langle aamqitidnp \rangle, \langle aamqidnp \rangle \}$ where the common prefix $\langle aamqi \rangle$ is the determined past of the respective process. We cannot write out the process traces of p_3 since $|c(p_3)| = 280$. Note that the examples are quite simple in as

much the process models' Flows do neither contain complicated link structures nor While-elements.

4.3 Bilateral process model dissimilarity

The bilateral process model dissimilarity between process models is pivotal for our approach.

Definition 2: (Bilateral process model dissimilarity). The bilateral process model dissimilarity is a function:

$$d : M \times M \rightarrow \mathbb{N}_0; (p, q) \mapsto \min_{\substack{t_p \in c(p) \\ t_q \in c(q)}} \{ l(t_p, t_q) \},$$

where l is the Levenshtein-distance (Levenshtein, 1966) between two process traces.

Particularly, $d(p, q) = 0 \Leftrightarrow c(p) \cap c(q) \neq \emptyset$, i.e., iff there is at least one common process trace covered by both process models p and q .

The bilateral process model dissimilarity d is vital for the definition of the t -similar process model set of a process model p .

Definition 3: (t -similar process model set). The t -similar process model set $M_t(p)$ is defined by:

$$M_t(p) := \{ q \in M_{kb} \mid d(p, q) \leq t \}.$$

This set contains all process models q whose dissimilarity to p is not higher than a threshold t . Regarding a testee p and an appropriately low t , the set $M_t(p)$ contains sufficiently similar process models which are relevant for further analysis.

4.4 Graph grammars and transition systems

The previous subsections raise the question how d can be computed. A naive implementation of $d(p, q)$ might look like this:

- 1 generate $c(p)$ and $c(q)$
- 2 find a $(t_p, t_q) \in c(p) \times c(q)$ with minimal $l(t_p, t_q)$.

This is inappropriate for three reasons: first, $c(p)$ and $c(q)$ have to be generated, which is particularly difficult for process models containing complex link structures and impossible for process models containing loops since these have possibly infinite process traces sets. Second, a direct implementation is computationally inefficient because there are $|c(p)| \cdot |c(q)|$ Levenshtein-distances to be computed. Third, we need to analyse similar process traces later on in order to provide the user with constructive hints.

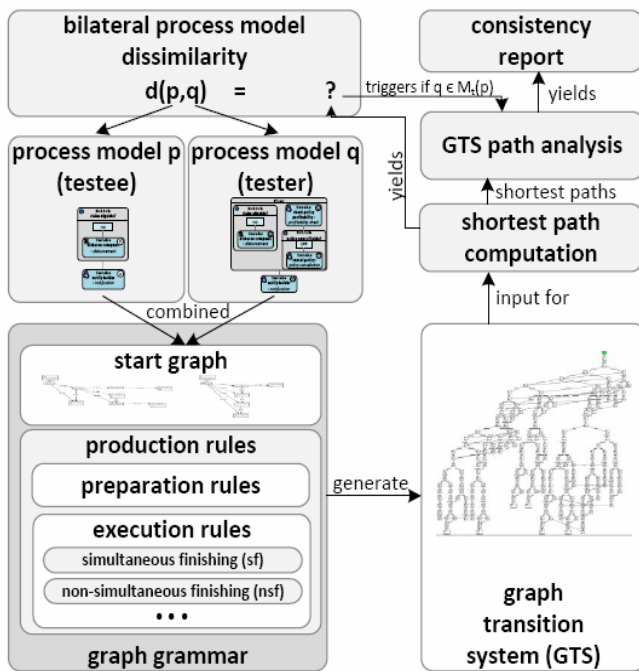
In the sequel, we present a graph grammar-based computation of d which is efficient and whose intermediate results can be used for later analysis. The coarse procedure is depicted in Figure 11. First, two process models p (testee) and q (tester) are combined to a single graph which contains the abstract syntax of both process models. This combined graph is repeatedly and automatically modified according to

possible execution steps in the process models. From that we obtain a *transition system*, wherein states are (modified) graphs and transitions are applied modifications. Since certain modifications hint to dissimilarity, we assign cost to these transitions and search for the cost of the *shortest path* in the transitions system. Then, this cost is the value of $d(p, q)$ and we can decide if $q \in M(p)$. If this is the case, the shortest paths are further analysed in order to generate a *consistency report* for p .

In the following, we give a short and informal description of graph grammars. This description is aligned with the formal definition of Rensink (2008).

A *graph grammar* is a pair $\mathcal{G} = (\mathcal{R}, \mathcal{I})$, where \mathcal{R} is a set of production rules and \mathcal{I} is an initial graph. Graph grammars are similar to common formal string grammars yet they operate on graphs instead of sequential structures. Thus, a production rule $r \in \mathcal{R}$ is similar to a rewriting rule in string grammars. It has a left hand side and a right hand side, which are both graphs. The *left hand side* constitutes a graph pattern that has been found in a graph s in order to apply r . If r is applicable, the matched subgraph in s is replaced by the graph defined by the right hand side of r yielding a graph s' . Like the start symbol in a string grammar, the initial graph \mathcal{I} can be successively rewritten as long as there are applicable production rules.

Figure 11 Computation of bilateral process model dissimilarity d (see online version for colours)



A graph grammar has an associated graph transition system (GTS). A GTS is similar to derivation structures of string grammars. Its states are graphs; particularly its initial state s_1 is the initial graph \mathcal{I} . Its transitions correspond to production rule applications insofar as a state s has a transition for each rule application. In GTS transitions are therefore labelled with the name of the respective

production rule. Please note that a certain rule r can have several matches in a certain graph. Thus, a state s might have several transitions labelled with the name of r each of which leads to a different state s' .

4.5 GTS for synchronous process executions

Due to space limitations we cannot provide a tutorial for the precise syntax and semantics of graph production rules as provided (e.g., Heckel, 2006). Instead, we exemplify applications of graph production rules by following paths in a GTS as depicted in Figure 12.

In this figure, the state graphs are drawn in the same way as the process models in the figures before. Please note that this is just a matter of concrete syntax representation. For brevity, the example is restricted to the comparison of the process models \hat{p}_2 and \hat{p}_3 delimited by the dotted bounding boxes in Figure 10. Furthermore, Figure 12 just shows some of those paths which lead to an end state. Even these paths are not complete as we leave out some intermediate states.

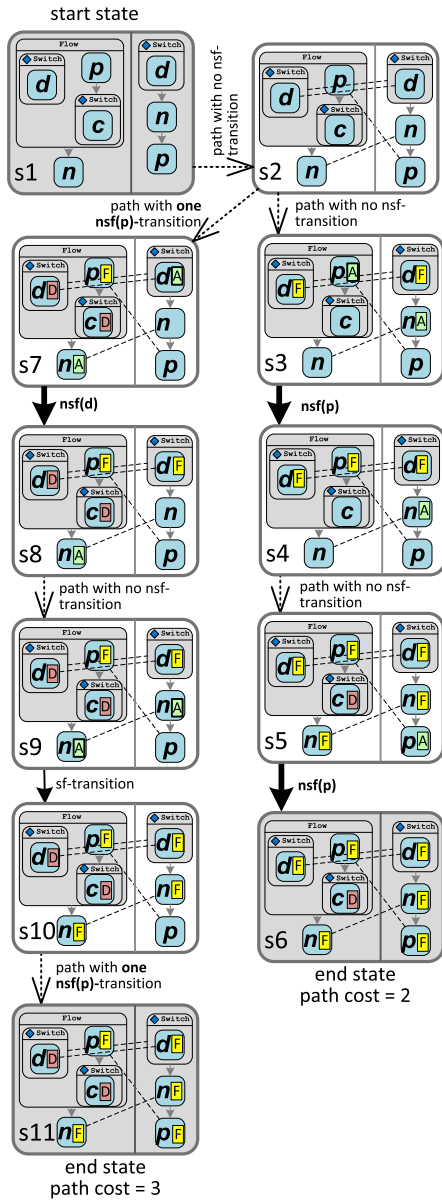
The transition system is divided into two phases. States s_1 to s_2 are part of the *preparation phase*. On the path from s_1 to s_2 activities with the same type in different processes are ‘paired’. Newly introduced pair-edges (dashed lines) mark the activity pairs. Pairing is not just done for atomic Invoke-activities but also for composite activities, which have exactly the same substructure, e.g., the Switch containing the activity d . Furthermore, if the process models have n and m activities of the same type, the pairing yields $n \cdot m$ pairs for this type.

Beginning with state s_2 *simultaneous execution* of the processes is simulated (*execution phase*). In the figure, finished activities are flagged with an F, active activities with an A and waiting activities not at all. In order to better distinguish unreachable activities, e.g., those that are on a dead path in a switch, we have also a mark D for that state.

Each execution step is a transition in the GTS and therefore conducted via application of a certain graph production rule. These rules take care of the control flow definition of the process models. For example, in the transition from s_7 to s_8 the activity d is set from active to finished.

Processes with similarities allow for *simultaneous finishing* of paired active activities. For example, there is one active n -activity in each of both processes in state s_9 and both are finished in the transition to state s_{10} . The corresponding production rule is called sf (simultaneous finishing). Being member of a pair is necessary but not sufficient for an activity for being simultaneously finished. For example, in the left hand process the p -activity must be finished before n can be set to active due to the process’ control flow definition. In the transition from s_3 to s_4 , p is therefore finished *non-simultaneously*, and again in the transition from s_5 to s_6 . The corresponding production rule is named nfs (non-simultaneous finishing).

Figure 12 Cutout of GTS (see online version for colours)



4.6 Shortest path computation

The *end states* of the GTS are those states, where both processes are finished, i.e., all activities are either finished or dead. The purpose of the simultaneous process executions is to compute how many nsf-transitions are visited to reach an end state in the GTS.

Each end state can usually be reached on several different paths. Many of them contain unnecessary nsf-transitions. Thus, we assign cost 1 to each nsf-transition and then do a *shortest path search* from the start state to one of the end states that is the path with the lowest cost. In the example, we have to traverse at least two bad transitions – one for the finishing of the left hand *p*-activity and one for the right hand.

The shortest paths in the GTS are the source for further analysis. The accumulated cost of a shortest path is the bilateral process model dissimilarity d of the respective process models, e.g., $d(\hat{p}_2, \hat{p}_3) = 2$ as well as $d(p_2, p_3) = 2$.

Comparing p_2 with p_1 yields $d(p_2, p_1) = 3$. For sake of simplicity, let the knowledge base be $M_{kb} = \{p_1, p_2, p_3\}$ and p_2 be the testee; then $M_2(p_2) = \{p_2, p_3\}$ (the testee is trivially included).

4.7 GTS path analysis

In order to provide detailed information concerning the differences between a testee p and relevant testers in $M_i(p)$ the shortest paths in the respective GTSs are to be analysed. In a GTS, labels of nsf-transitions contain information about the match of the respective production rule, i.e., about what is changed in the transition. In particular, a bad nsf-transition contains the activity type of the activity that had to be finished non-simultaneously. In the shortest path of Figure 12, both bad transitions are labelled with $nfs(p)$, i.e., a p -activity has to be finished non-simultaneously twice – once in each process. On another shortest path which is not depicted in the figure, there are two $nfs(n)$ transitions. Patterns in shortest paths like the one described indicate that activities are badly positioned. For example, in p_2 the p -activity is badly positioned with regard to the n -activity compared to process model p_3 . Such patterns can be automatically detected and enlisted in a *consistency report*.

The size of a consistency report depends on the size of $M_i(p)$ and on the number of shortest paths in each bilateral process model dissimilarity check. A process participant might therefore be overburdened with details about differences between the testee and the (numerous) testers. For that reason we follow a simple strategy to consolidate the consistency report. In dynamically modified process instance models, the consistency report can be aligned with the changed process part. For example, in the GTS of \hat{p}_2 and \hat{p}_3 there is also a shortest path with two $nfs(n)$ -transitions instead of $nfs(p)$ -transitions. This path is omitted in the figure for sake of readability. The (depicted) shortest path with two $nfs(p)$ and the (omitted) shortest path with two $nfs(n)$ are complementary since the n -activity and the p -activity are badly positioned against each other. Assume that p has been dynamically added. Then, it is reasonable to put the blame on p and leave out the complementary shortest paths containing $nfs(n)$ in the consistency report.

4.8 Implementation remarks

Graph grammars gave rise to an entire research field. There are several mature tools for the definition of production rules and their execution [cf. comparison in Fuss et al. (2007)]. From these tools we chose GROOVE (Rensink, 2003) for the following reasons: first, GROOVE particularly targets checking properties of systems with graph-structured states which suits our need as we can graphically represent a process state (process instance model). Second, GROOVE provides functions for explicit GTS-generation. Third, quantifiers in production rules are available which helps to reduce the state space of the GTS. Forth, there a few

technical dependencies coming along with GROOVE; thus, it can be easily integrated with our PME.

Essentially, GROOVE is a model checking tool. Unfortunately, it therefore also suffers from the well-known state space explosion problem, which is intrinsic to all model checking approaches [Baier and Katoen (2008), Section 2.3]. This problem has to be accounted for at design time of the production rules. Luckily, the state space can be drastically reduced by a control automaton defined in GROOVE which imposes an additional scheduling of production rules and the application of quantified subrules. Despite focusing the behaviour (process trace set) of process models, we can therefore optimise by exploiting the structural identities of composite activities. For instance, composite paired activities with equal substructures can be simultaneously finished in one step without executing their internal substructure. This is legitimate since identical (sub-) structures always exhibit the same behaviour. For instance, this is applicable for the Flow-activity at the beginning of process models p_2 and p_3 during their simultaneous execution.

In the screenshot of the PME (cf. Figure 9), the list entries in the Infos-section at the very bottom constitute the consistency report for the process model \hat{p}_2 (diagram in the centre). For example, it states that \hat{p}_2 differs from \hat{p}_1 in the execution order of activity types p and n and exclusively has a p -activity.

4.9 Discussion

The *strengths* of the graph grammar approach are the following: first, we can avoid the explicit computation of the process trace set $c(p)$ for a process model p . Instead, process traces are indirectly generated during simultaneous execution of the process models. Due to the mentioned optimisation, unnecessary dissimilar process traces are avoided, which drastically reduces time complexity compared to the naive implementation. Second, the production rules take into account the (partial) process traces of running or finished process instances. This is achieved, e.g., by ruling out a simultaneous finishing if one activity in the respective pair is already determined to be unreachable (dead).

The approach presented in this section bears a *problem* which is still to be solved. In spite of the optimisations in the graph grammar, combinations of processes which have many activity types in common but exhibit a completely different behaviour due to different control flow definitions still lead to too big GTS. For the time being, we cancel the GTS-generation at a certain threshold for generated states. Fortunately, these cases just imply a decline of the consistency check result quality yet do not render the entire result useless. Nonetheless, we plan to alleviate this problem by utilising a mixture of incomplete GTS-generation strategies and settle for locally shortest paths. Moreover, checking each model against each other model in the knowledge base requires $|M_{kb}|^2$ checks, which of course is too much for knowledge bases of realistic sizes. In future

work, we will deal with this problem by pre-filtering potential testers.

4.10 Related work

There are plenty of works dealing with similarity of process models. They differ in several regards from each other and our work: first, the notion of similarity is different, i.e., whether processes must have partial or full process traces in common, whether they have to have the same process traces, or if they even have to be bisimilar. Our similarity notion is deliberately weak since we consider processes similar ($d = 0$) if they just share at least one common process trace. Second, the evaluation of a similarity computing function is considered to be either qualitative (similar or not) or quantitative (degree of similarity). Our approach is qualitative as we use the degree of similarity for determining the testers which are further analysed. Third, the approaches differ much in how they actually compute a similarity function, even in what is taken as input for computation. Some approaches consider differences in the syntactic structure of the compared process models others regard the respective behaviour, i.e., the process traces. Particularly, this property contributes to the novelty of our approach. We mainly regard the behaviour but use structural identities for optimisation. To our best knowledge, there is no other approach pursuing the same strategy, particularly no one that employs graph grammars. Nonetheless, related literature provides many ideas which we consider worth for adoption.

Research on process similarity can profit from works on string language similarity since both processes and string languages can be defined by FTA. This is studied by Wombacher and Rozie (2006). The authors point out the limitations of approaches which merely consider the graph structure of FTAs.

This limitation is targeted in the approach of Li et al. (2008b), which is sensitive for semantics of control flow activities. Here, the edit distance between ADEPT process models of Reichert and Dadam (1998) is measured for quantitative similarity computation. This is done by determining the minimal number of correctness preserving high level edit operations (insert, delete, and move) which are necessary to transform process models into each other. In Li et al. (2008a), the authors show how this can be used to derive a reference model with an aggregated minimal distance to variants in a given set. We particularly consider ordering conflicts detection based on order matrices interesting as it indirectly resembles our GTS-path analyses yet refers to the process structure.

Küster et al. (2008) provide an approach for displaying and semi-automatically resolving differences in the syntactic structure of process definition model versions. This particularly includes computation of the hierarchical composition structure in (flat) graphically structured process models before searching for equal (composite) fragments between two process models. This part of their work is interesting for us as we could adopt it to find more complex pairs in the preparation phase described in Subsection 4.6.

In van der Aalst et al. (2006), the authors refrain from comparing process models directly but take (simulated) process traces of at least one model as input for quantitative similarity computation on Petri nets. Of course, the strengths of this approach can be applied if representative process traces are present or can be generated.

There is also a connection between our work and process (change) mining since consistency checks rely on information mined from existing process models. However, approaches and frameworks like ProM for process mining (van Dongen et al., 2005) rather use existing process traces for deriving process definition models from scratch instead of supporting the editing of existing process instance models by mining other process models. Process change mining as presented by Günther et al. (2006) mines logged dynamic change operations in order to improve the quality of process definition models which is not in our focus.

5 Conclusions

In this article, we presented results of the DYPROTO project, a three-year research cooperation between our group and the GDIS GmbH. The goal of this work was to provide support for dynamic business processes in the insurance domain.

It was clear from the very beginning that these dynamic processes require functionality we name ‘run time dynamics’, which supports dynamic modification of process instance models, i.e., models of running processes. Run time dynamics at least have to include dynamic adding of unpredicted activities, dynamic removing of unnecessary activities or dynamic reiteration of (previously executed) activities. We had to build upon our partner’s existing PMS WPS (along with the process definition language WS-BPEL) since our prototypes should not change let alone replace WPS but should non-invasively extend it. Driven by these requirements we learned that direct manipulation of a process instance model’s structure is dispensable for implementation of run time dynamics. Instead, one can simulate direct manipulation by an automatic augmentation of process definition models by additional control flow structures, e.g., additional *Invoke*-, *Switch*- and *While*-activities. These additional activities are hidden from process participants but used at process run time to carry out a dynamic change, e.g., by rerouting the control flow to another newly created process instance in order to simulate dynamic adding. Our approach does not rely on any WPS-specifics and is thus applicable for other WS-BPEL-engines. Furthermore, the approach is aligned with dynamics patterns, e.g., dynamic adding. Support for each pattern can be realised or disabled depending on the needs of process participants and on the dynamics patterns which are already natively implemented by direct process structure manipulation in the particular PMS. Essentially, we can state: Appropriate support for dynamic business processes can also be achieved with static PMS.

Process models are not supposed to be edited in an arbitrary way but have to adhere to correctness constraints, e.g., process variables have to be initialised before being read, and compliance constraints modelled in explicit graphical process knowledge models, e.g., commission of expertises requires its later incorporation. In the worst case, violations of these constraints lead to technical problems (exceptional process instance termination) or professional problems (conflicts with company specific or legal regulations). We discovered that many of these constraints can be efficiently checked by analysing the syntactic structure of the respective process models. This works in particular for a dynamically modified process instance model. We make use of the OCL, and define OCL-expressions which refer to integrated process meta-models. Through this, checks for correctness and compliance constraints can be uniformly realised since in both cases we check one (complex) syntactic structure for certain properties.

Explicit process knowledge models bear the immanent downside that their maintenance causes extra work and that they might be incomplete. However, process knowledge is also implicitly contained in existing process definition and instance models. In order to utilise this knowledge, we first have to filter out those process models (testers) which are sufficiently similar to a checked process model (testee). Then, consistency between the testee and testers can be reasonably analysed yielding differences between process models with regard to executed activities and their execution order. In order to abstract from the process model structure (control flow definition) and to avoid computational difficulties, we made use of the graph grammar tool GROOVE.

Acknowledgements

This research project was part of the Transfer Center 61 funded by the Deutsche Forschungsgemeinschaft. We thank Prof. Dr.-Ing. M. Nagl for his valuable academic input for our work, our project partners K. Wolf, Dr. S. Bühne, and H. Wessels for fruitful discussions about insurance processes and our students N. Ehses, T. Kurpick, A. Fischer, and T. Lake for their contributions to the prototypes.

References

- Atkinson, C. and Dadam, P. (2007) ‘AristaFlow: Komponentenbasierte Anwendungsentwicklung, Prozesskomposition mittels Plug & Play und adaptive Prozessausführung’, in *doIT-Forschungstag 2007*.
- Awad, A., Decker, G. and Weske, M. (2008) ‘Efficient compliance checking using BPMN-Q and temporal logic’, in Dumas, M., Reichert, M. and Shan, M.-C. (Eds.): *6th International Conference on Business Process Management (BPM) 2008, LNCS*, Vol. 5240, pp.326–341, Springer.
- Baier, C. and Katoen, J.-P. (2008) *Principles of Model Checking*, MIT Press.

- Bianculli, D., Ghezzi, C. and Spoletini, P. (2007) 'A model checking approach to verify BPEL4WS workflows', in *IEEE International Conference on Service-Oriented Computing and Applications (SOCA) 2007*, IEEE Computer Society, pp.13–20.
- Casati, F. (1998) 'Models, semantics, and formal methods for the design of workflows and their exceptions', PhD thesis, Politecnico di Milano.
- Clarke, E.M., Grumberg, O. and Peled, D.A. (1999) *Model Checking*, The MIT Press.
- Dadam, P. and Reichert, M. (2009) 'The adept project: a decade of research and development for robust and flexible process support', *Computer Science – Research and Development*, Vol. 23, No. 2, pp.81–97.
- Eversheim, W., Michaeli, W., Nagl, M., Spaniol, O., Weck, M. and Westfechtel, B. (1997) 'SUKITS: Management von Entwicklungsprozessen im Maschinenbau', *Softwaretechnik-Trends*, p.17.
- Förster, A., Engels, G., Schattkowsky, T. and Straeten, R.V.D. (2007) 'Verification of business process quality constraints based on visual process patterns', in *First Joint IEEE/IFIP Symposium on Theoretical Aspects of Software Engineering (TASE) 2007*, IEEE Computer Society, pp.197–208.
- Fuss, C., Mosler, C., Ranger, U. and Schultchen, E. (2007) 'The jury is still out: a comparison of Agg, Fujaba, and progress', *ECEASST*, Vol. 6.
- Governatori, G., Milosevic, Z. and Sadiq, S.W. (2006) 'Compliance checking between business processes and business contracts', in *Tenth IEEE International Enterprise Distributed Object Computing Conference (EDOC)*, IEEE Computer Society, pp.221–232.
- Gravel, A., Fu, X. and Su, J. (2007) 'An analysis tool for execution of BPEL services', in *CEC/EEE*, IEEE Computer Society, pp.429–432.
- Günther, C.W., Rinderle, S.B., Reichert, M.U. and van der Aalst, W.M.P. (2006) 'Change mining in adaptive process management systems', in *14th International Conference on Cooperative Information Systems (CoopIS'06)*, LNCS, Vol. 4275, pp.309–326, Springer, Montpellier, France.
- Heckel, R. (2006) 'Graph transformation in a nutshell', *Electr. Notes Theor. Comput. Sci.*, Vol. 148, No. 1, pp.187–198.
- Heer, T., Briem, C., and Wörzberger, R. (2008) 'Work-flows in dynamic development processes', in Ardagna, D., Mecella, M. and Yang, J. (Eds.): *Business Process Management Workshops, Lecture Notes in Business Information Processing*, Vol. 17, pp.266–277, Springer.
- Heller, M., Jäger, D., Krapp, C.-A., Nagl, M., Schleicher, A., Westfechtel, B. and Wörzberger, R. (2008) *An Adaptive and Reactive Management System for Project Coordination*, in Nagl, M. and Marquardt, W. (Eds.): pp.300–366.
- Koehler, J., Tirenni, G. and Kumaran, S. (2002) 'From business process model to consistent implementation: a case for formal verification methods', in *6th International Enterprise Distributed Object Computing Conference (EDOC)*, IEEE Computer Society.
- Küster, J.M., Gerth, C., Förster, A. and Engels, G. (2008) 'Detecting and resolving process model differences in the absence of a change log', in Dumas, M., Reichert, M. and Shan, M.-C. (Eds.): *BPM, Lecture Notes in Computer Science*, Vol. 5240, pp.244–260, Springer.
- Levenshtein, V.I. (1966) 'Binary codes capable of correcting deletions, insertions and reversals', *Soviet Physics Doklady*, Vol. 10, No. 8, pp.707–710.
- Li, C., Reichert, M. and Wombacher, A. (2008a) 'Discovering reference process models by mining process variants', in *ICWS*, IEEE Computer Society, pp.45–53.
- Li, C., Reichert, M. and Wombacher, A. (2008b) 'On measuring process model similarity based on high-level change operations', in Li, Q., Spaccapietra, S., Yu, E. and Olivé, A. (Eds.): *27th International Conference on Conceptual Modeling (ER)*, LNCS, Vol. 5231, pp.248–264, Springer.
- Liu, Y., Müller, S. and Xu, K. (2007) 'A static compliance-checking framework for business process models', *IBM Systems Journal*, Vol. 46, No. 2, pp.335–362.
- Ly, L.T., Göser, K., Rinderle-Ma, S. and Dadam, P. (2008) 'Compliance of semantic constraints – a requirements analysis for process management systems', in *Proc. 1st Int'l Workshop on Governance, Risk and Compliance – Applications in Inf. Sys. (GRCIS'08)*.
- Mendling, J. and van der Aalst, W.M.P. (2007) 'Formalization and verification of EPCs with or-joins based on state and context', in Krogstie, J., Opdahl, A.L. and Sindre, G. (Eds.): *19th International Conference on Advanced Information Systems Engineering (CAiSE)*, LNCS, Vol. 4495, pp.439–453, Springer.
- Nagl, M. and Marquardt, W. (Eds.) (2008) 'Collaborative and distributed chemical engineering', *From Understanding to Substantial Design Process Support – Results of the IMPROVE Project*, LNCS, Vol. 4970, Springer.
- Pesic, M., Schonenberg, M. and Aalst, W. (2007) 'Declare: full support for loosely-structured processes', in *EDOC '07: 11th IEEE International Enterprise Distributed Object Computing Conference (EDOC)*, IEEE Computer Society.
- Recker, J. and Mendling, J. (2006) 'On the translation between BPMN and BPEL: conceptual mismatch between process modeling languages', in *The 18th International Conference on Advanced Information Systems Engineering – Proceedings of Workshops and Doctoral Consortium*.
- Reichert, M. (2000) 'Dynamische ablaufänderungen in workflow-management-systemen', PhD thesis, University of Ulm.
- Reichert, M. and Dadam, P. (1998) 'ADEPTflex-supporting dynamic changes of workflows without losing control', *Journal of Intelligent Information Systems*, Vol. 10, No. 2, pp.93–129.
- Rensink, A. (2003) 'The GROOVE simulator: a tool for state space generation', in Pfaltz, J.L., Nagl, M. and Böhlen, B. (Eds.): *Second Workshop on Applications of Graph Transformations with Industrial Relevance (AGTIVE)*, LNCS, Vol. 3062, pp.479–485, Springer.
- Rensink, A. (2008) 'Explicit state model checking for graph grammars', in Degano, P., Nicola, R.D. and Meseguer, J. (Eds.): *Concurrency, Graphs and Models*, LNCS, Vol. 5065, pp.114–132, Springer.
- Rozinat, A. and van der Aalst, W.M.P. (2008) 'Conformance checking of processes based on monitoring real behavior', *Inf. Syst.*, Vol. 33, No. 1, pp.64–95.
- Sadiq, S.W., Sadiq, W. and Orłowska, M.E. (2001) 'Pockets of flexibility in workflow specification', in *20th Int. Conf. on Conceptual Modeling (ER) 2001, Lecture Notes in Computer Science*, Vol. 2224, pp.513–526, Springer.

- Schleicher, A. (2002) 'Management of development processes: an evolutionary approach', PhD thesis, RWTH Aachen University.
- van der Aalst, W.M.P., de Medeiros, A.K.A. and Weijters, A.J.M.M. (2006) 'Process equivalence: comparing two process models based on observed behavior', in Dustdar, S., Fiadeiro, J.L. and Sheth, A.P. (Eds.): *Business Process Management, LNCS*, Vol. 4102, pp.129–144, Springer.
- van Dongen, B.F., de Medeiros, A.K.A., Verbeek, H.M.W., Weijters, A.J.M.M. and van der Aalst, W.M.P. (2005) 'The ProM framework: a new era in process mining tool support', in *ICATPN, LNCS*, No. 3536, pp.444–454, Springer.
- Verbeek, H. and van der Aalst, W. (2005) 'Analyzing BPEL processes using Petri nets', in *Second International Workshop on Applications of Petri Nets to Coordination*, pp.59–78.
- Verbeek, H.M.W., Basten, T. and van der Aalst, W.M.P. (2001) 'Diagnosing workflow processes using Woflan', *The Computer Journal*, Vol. 44, No. 4, pp.246–279.
- Weber, B., Reichert, M. and Rinderle-Ma, S. (2008) 'Change patterns and change support features – enhancing flexibility in process-aware information systems', *Data Knowl. Eng.*, Vol. 66, No. 3, pp.438–466.
- Wombacher, A. and Rozie, M. (2006) 'Evaluation of work-flow similarity measures in service discovery', in Schoop, M., Huemer, C., Rebstock, M. and Bichler, M. (Eds.): *Conference on Service Oriented Electronic Commerce, LNI*, Vol. 80, pp.51–71, GI.
- Wörzberger, R. and Heer, T. (2008) 'Process model editing support using eclipse modeling project tools', in Friese, P., Zambrovski, S. and Zimmermann, F. (Eds.): *Second Workshop on MDS Today, Lecture Notes in Informatics*, Shaker Verlag.
- Wörzberger, R., Ehses, N. and Heer, T. (2008a) 'Adding support for dynamics patterns to static business process management systems', in Pautasso, C. and Tante, É. (Eds.): *Software Composition, LNCS*, Vol. 4954, pp.84–91, Springer.
- Wörzberger, R., Kurpick, T. and Heer, T. (2008b) 'Checking correctness and compliance of integrated process models', in Negru, V., Jebelean, T., Petcu, D. and Zaharie, D. (Eds.): *SYNASC 2008*, IEEE Computer Society, pp.576–583.