International Journal of Cooperative Information Systems

CHANGE PROPAGATION AND CONFLICT RESOLUTION FOR THE CO-EVOLUTION OF BUSINESS PROCESSES

GEORG GROSSMANN
SHAMILA MAFAZI
WOLFGANG MAYER
MICHAEL SCHREFL
MARKUS STUMPTNER

Advanced Computing Research Centre
School of IT and Mathematical Sciences
University of South Australia
Mawson Lakes, SA, 5095, Australia

Received (Day Month Year)
Revised (Day Month Year)

In large organizations multiple stakeholders may modify the same business process. This paper addresses the problem when stakeholders perform changes on process views which become inconsistent with the business process and other views. Related work addressing this problem is based on execution trace analysis which is performed in a post-analysis phase and can be complex when dealing with large business process models. In this paper we propose a design-based approach that can efficiently check consistency criteria and propagate changes on-the-fly from a process view to its reference process and related process views. The technique is based on consistent specialization of business processes and supports the control flow aspect of processes. Consistency checks can be performed during the design time by checking simple rules which support an efficient change propagation between views and reference process.

Keywords: Business process evolution; process views; process change propagation; process consistency

1. Introduction

It is apparent that business processes evolve over time and have to be aligned with business rules, legislations, and law. Cause of change are factors such as continuing change in regulations, policies, business compliance requirements, and user demand. Since the economic success of any business relies on its ability to respond to changes quickly, it is important to be able to update processes in an efficient and correct way [1].

A usual scenario in business process management is that complex process models are abstracted to simplify understanding and complexity, and the creation of "personalized views" of a process is a much needed feature to support customized

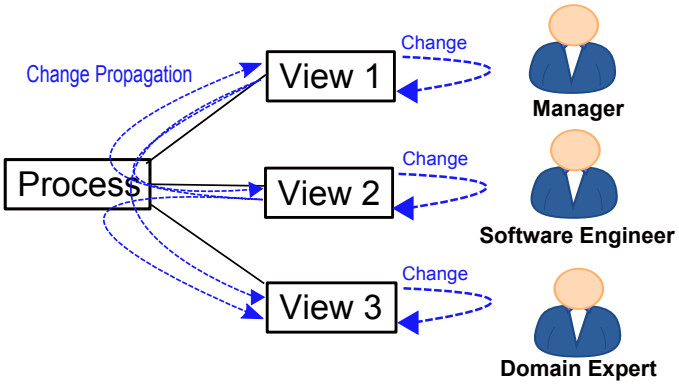


Fig. 1: Typical scenario: Three stakeholders change their personal views on a common *reference process*.

abstraction and visualization of processes for different stakeholders [2–5]. Designing and updating process models in large organizations is usually performed by multiple stakeholders in parallel. Each stakeholder has a particular view on a process and performs changes independently from other stakeholders leading to the challenge that views may become out of sync and inconsistent with its reference process and other views. In order to preserve consistency among views a change in one view needs to be propagated to the remaining views through their common *reference model*. The life-cycle of a process view can be described by three phases [6]: creation-, management- and elimination phase. This article addresses two major research questions: (1) How to propagate changes from a process view to its related reference process model and, thereafter, to other views of the reference model? (2) How to detect that two concurrent changes over two different views are in conflict and how to manage detected conflicts? The motivation behind this approach is that process views are simpler to understand for stakeholders because they are personalized, and stakeholders can perform changes more easily on views rather than on a complex reference process model.

1.1. *Typical Process Design Scenario*

Fig. 1 shows a typical process design scenario, where each stakeholder has his/her own personal view over a reference model and performs changes on it. To preserve the consistency between the models, changes in one view need to be propagated to related models.

Fig. 2 shows an example for the co-evolution of models specified in the Business Process Model and Notation language (BPMN). Process model M' depicts the reference model and models M_1 and M_2 show two views over M'. The dashed lines indicate the correspondence relations for activities in M' which are represented in M_1. The cross in M_2 indicates an activity that has been removed during the

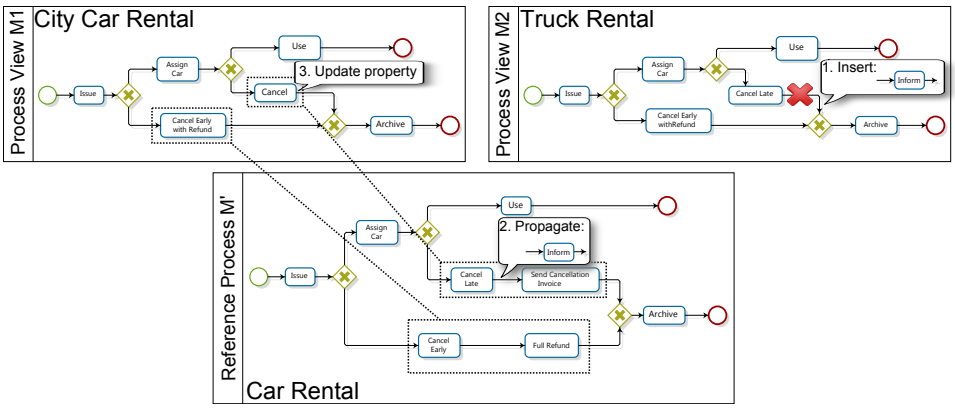


Fig. 2: A change propagation example from view M_1 to view M_2 via the reference process M'.

abstraction. Since the views look different, the mappings between elements in the reference process and elements in the two views are different. For example activities *Cancel Late* and *Send Cancellation Invoice* in M' are mapped to *Cancel* in M_1, whereas activity *Send Cancellation Invoice* has been removed in M_2.

In the example, a new activity *Inform* is inserted in M_2. This change is then propagated to M' by inserting *Inform* between *Cancel Late* and *Send Cancellation Invoice*. In a next step, the change is propagated to M_1 which consists only of an update of the properties in *Cancel* because it is an abstraction of the sequence *Cancel Late* and *Send Cancellation Invoice* in M'.

The consistent representation of data flows in related process models must be considered, as otherwise invalid views could be obtained where data flows are not well-formed or do not adequately reflect a consistent abstraction from the reference process. Reichert and Weber [1] investigated the correctness of data flow for pre-specified processes and discussed properties such as *no missing data*, *no unnecessary data* and *no lost updates*. Such properties are relevant in the presented example: if one decided to hide activity *Assign Car* from view M_1 in Figure 2, the resulting view would feature a *missing data anomaly* [1] as activity *Use* still requires the *Car Registration* as data input; however, this information is no longer supplied as the *Assign Car* activity has been omitted in the view.

In this article we address the consistent representation of data input and output in the context of behavioral consistency, such as observation- and invocation consistency [7]. For checking data flow correctness properties as identified by Reichert and Weber we refer to related work only. For example, activities in a view that are to be invocation consistent with the reference process activities must feature the same data inputs at the same level of granularity as their counterpart activities in the reference process. Similarly, for aggregate data elements, all aggregated components must be present in the corresponding process region in the reference process. These

examples illustrate that the consistency of data flows in related process models can be non-trivial and should therefore be an integral part of any process evolution approach. In particular, mechanisms to validate the consistency of data flows in a view relative to the reference process must be provided and integrated with other behavioral consistency notions. The following section presents our framework for maintaining data consistency in process models related by the behavioral consistency criteria of [7].

1.2. *Contribution*

This article focuses on the *view creation* and *view management phase* of a process view life cycle [6]. For the creation of views several methods have been proposed [8–10]. In previous work [11], we introduced a knowledge based framework to provide different views on a process model considering user-specific *conditions* which specify the purpose of the abstraction and identify elements of interest. We have applied our approach to a real-world business process repository from the "Product and System Engineering" domain which contained software development processes with up to 260 activities in a single process. In some use cases, a view could be generated that reduced the complexity by up to 90% [12]. For the view management phase, Gerth et al. [13] identified three steps: detection of differences, identification of change conflicts, and resolution of conflicts. For the detection and identification of change conflicts existing work is mainly based on global trace analysis which can be costly when dealing with large processes.

For the view creation phase we present an abstraction framework in this article which supports control flow and data input/output of activities in a business process model. The process view is generated from a process model based on an *abstraction condition*. An abstraction condition includes constraints on the process structure or properties of process elements, e.g., select all activities which take longer than 10 minutes.

For the view management phase we propose generic methods and techniques for the efficient change propagation from a stakeholder to reference process and its process views on the process model level. Given a well-formed reference process (i.e., safe; no missing, unnecessary or lost data), change propagation and conflict resolution between views and reference process is based on simple design rules that can be checked without the need to generate all possible execution paths and checking control flow consistency involves at most checking some "local" execution paths (i.e., execution paths of activities in the refinement of an activity in a view). These techniques supports change propagation on-the-fly so stakeholders become immediately aware of changes made by other stakeholders which affect their view. On-the-fly change propagation refers to propagating changes during design time from one view to its reference process and any other views which contains elements that have been affected by change. By providing these capabilities, a separate step for conflict identification and resolution can be kept to a minimum or in optimal cases

be eliminated. However, if changes are performed in two different views affecting the same process region, conflicts may arise and need to be resolved.

We apply existing techniques based on behavior inheritance and use this technique to propagate changes efficiently [7]. We allow users to perform changes that violate the refinement relations and then propagate changes so consistency is restored. In particular, we are applying consistent specialization of business processes to ensure consistency between views and reference process. By checking simple rules on the structure of process models and keeping relationships between elements in different processes, we are able to achieve co-evolution more efficiently.

The limitations of the current framework are that it supports a limited number of change operators which are *insert* and *elimination* similar to those proposed by Kolb et al. [5]. Although we propose a library of change operators that can be extended, some change operators, e.g., operators that insert new complex structures or the refinement of already abstracted elements are currently not supported.

This article extends previous work [14] published in the International Conference on Cooperative Information Systems (CoopIS 2013) and includes a new section on resolving change conflicts for business processes and an extended literature review. The main contributions in comparison to our previous work are: (1) Section 2.3 contains a detailed check for consistency of control- and dataflow using observation and invocation consistency, (2) Section 3 presents a framework for conflict detection in process co-evolution and describes a conflict resolution technique based on type discrimination, and (3) Section 5 provides a literature review and comparison criteria of state-of-the-art work in process (co-)evolution, change management and conflict resolution.

In the next section we explain our approach and the conflict resolution approach followed by a comparison of related work. The last section provides an outlook on future work and conclusion.

2. A Framework for the Consistent Co-Evolution of Business Process Models

A typical software development life-cycle consists of several phases. Köhler et al. [15] proposed a cycle for the alignment of business and IT which consists of model-, develop-, deploy-, monitor- and analyze & adapt phases. The presented framework focuses on the modeling phase and supports the consistent co-evolution of business processes in three steps: (1) in the *process modeling* step, a business process is created that represents the reference process, (2) in the *process view generation* step, views are generated from the reference process based on *abstraction conditions* which are provided by stakeholders, and (3) in the *change and change propagation* step, different stakeholders perform changes on their views and their changes are propagated on-the-fly to reference processes and other views. The first step is performed only once whereas Steps 2 and 3 are usually repeated for each development cycle.

Over the past decade significant effort has been spent on developing process models based on Semantic Web technologies, in particular in the context of service-oriented architectures. However, they have found little traction in the process modeling domain, as the focus of ontology languages is on class level descriptions, resulting in significant extra effort required to produce workable service/process specifications in these languages [16], and the requirement to use families of complex languages such as OWL [17], SPARQL [18], or SWRL [19] to enable the required level of expressiveness to describe even simple process instances, an effect that has also been found in other domains [20]. Instead, standard process notations allow a natural and concise expression of process structure.

Our framework makes use of the *model driven engineering (MDE)* paradigm. The advantages of MDE include increasing the productivity and compatibility by simplifying the process of design, and maximizing reusability by lifting platform specific code to a platform independent model level. Our approach is set on the level of models which are the central artifact in MDE. In the following we explain each step in detail and provide definitions for the formal notation and consistency constraints.

2.1. *Process Modeling*

There exist different languages for modeling business processes. BPMN has become the de-facto standard in recent years and we use this notation to represent the models. However, a more formal representation is required for specifying the relationship between reference process and process views, and verification of consistency. We use a Petri net representation of BPMN models, in particular labeled Petri nets, which are helpful to express the consistent specialization of processes [7]. Different approaches for mapping between the BPMN and Petri net notation exist. Since the discussion of such mappings are out of the scope of this paper we refer the interested reader to [21, 22].

A labeled Petri net is a Petri net that has labels attached to its arcs. The idea of labeling arcs corresponds to the mechanism by which different copies of a form are handled by each activity in business processes guided by paperwork, where different copies of a form have a different label [7]. Each activity deals with a different aspect of the same form where it can collect copies of the form from different input boxes and deliver them to different output boxes. The labels determine which copies have been used by which activities. When related to business processes, the form can be seen as a process instance and the labels as aspects of a process instance.

Definition 2.1. (Labeled Petri Net)

A tuple: $(N, F, D, \delta_{in}, \delta_{out}, L, l)$ is a labeled Petri net model, where N is a finite set of nodes partitioned into disjoint sets of activities (transitions) N_t and states (places) N_s, $F \subseteq (N_s \times N_t) \cup (N_t \times N_s)$ is the flow relation such that (N, F) is a connected graph, D is a finite set of data variables, and $\delta_{in} : N \mapsto 2^D$ and $\delta_{out} : N \mapsto 2^D$ are functions mapping each node to its set of input and output variables,

respectively. We write $n.D_{in}$ and $n.D_{out}$ for $\delta_{in}(n)$ and $\delta_{out}(n)$, respectively. L is a finite set of labels, and $l : F \mapsto 2^L \setminus \emptyset$ is a function that assigns a set of labels to each control flow.

Definition 2.2. (Immediate Pre-Node and Post-Node) The set of (immediate) pre-nodes of node $n \in N$ is defined as $\bullet n = \{p \in N | (p, n) \in F\}$. Likewise the set of the (immediate) post-nodes of node $n \in N$ is defined as $n\bullet = \{p \in N | (n, p) \in F\}$. We leave out the "immediate" if it is understood.

Different operations such as initializing, updating, and verifying, performed by each activity in a process model, can be divided into read and write operations from an implementation point of view. In our framework, when an activity reads or writes a data object, that data object is considered to be the input or the output of that activity. The sets of all inputs and outputs of an activity x are denoted as $x.D_{in}$ and $x.D_{out}$ respectively.

The execution semantics of a labeled Petri net is the same as that of a Petri net where an activity consumes a token from each of its pre-states and produces a token to each of its immediate post states.

Well-formed labeled Petri nets: For the reference process model and all its views, we assume a subset of labeled Petri net which satisfies the following structural properties and data flow properties. These properties must be observed before consistency checks can be performed, i.e., they must be observed during the creation of a view and after each change of a process view:

- *Single initial state per label*: Each label is assigned to exactly one state with no inflow.
- *Safe*: A labeled Petri net is safe iff there is no execution trace in the model such that some activities in that trace can be completed while there exist some post states of those activities on that trace, i.e., there can be only one token in all reachable markings where a marking is the distribution of tokens over activities. This is a necessary property as an unsafe Petri net contradicts the intention of processes to identify by a state or an activity a single, specific processing state of an object.
- *Activity reduced*: This property is satisfied iff for every activity there is an execution trace that contains it. This property is useful to ensure that the process model contains only activities which may have an influence on the process execution.
- *Deadlock-free*: This property ensures that the execution must continue unless it reaches a final state. This is an important property as it prevents from blocking execution.
- *Label preservation*: This property requires that for every activity the incoming and the outgoing arcs carry the same set of labels. If the analogy for introducing labels above is followed then the label preservation property

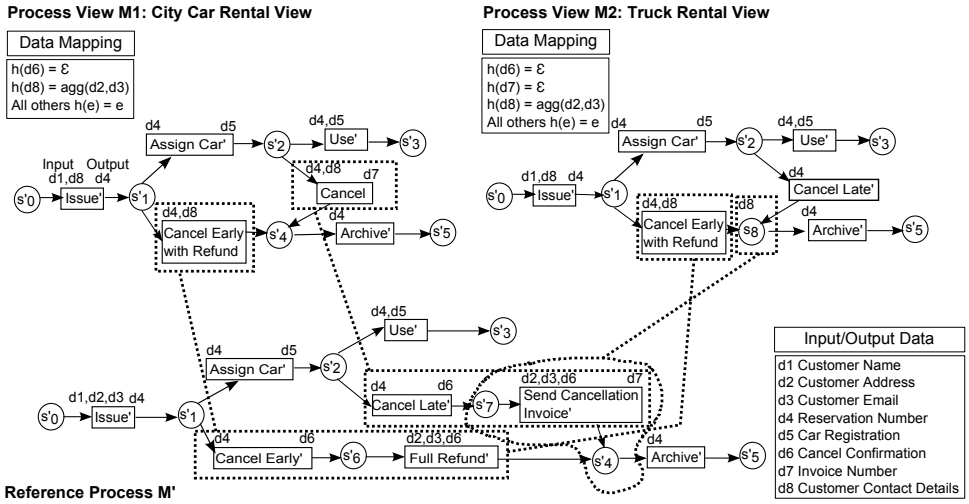


Fig. 3: Petri net model for Process Models M', M_1 and M_2 in Fig. 2. The flow labels are not shown.

ensures that if a copy of a process instance has been created, then this copy is neither overwritten nor are new copies created by other activities in the process model.

- *Unique label*: This property ensures that for every activity the label sets of all incoming arcs are disjoint and the label sets of all outgoing arcs are disjoint. In other words, it ensures that every activity can have some copies of a process instance from only one input box and must deliver them to only one output box.
- *Common label distribution*: This property ensures all the immediate arc(s) of a state must contain the same set of labels. This property ensures that each outgoing control flow holds the same copies of an instance.

Example 2.1. Fig. 3 shows the Petri net representation of the BPMN process model shown in Fig. 2. Data input and -output are indicated above each activity in the form of d_x where x indicates the type of data. Inputs (outputs) are shown on the left (right) hand side of an activity or state. For example, activity *Assign Car'* in Process View M_1 of Fig. 3 receives input d_4 (i.e. *Reservation Number*) which is produced by activity *Issue'*, and produces output d_5 (*Car Registration*) required by activity *Use'*.

After a reference process is modeled in BPMN and its equivalent representation as a labeled Petri net is generated, process views can be created from it.

2.2. *Creating Process Views*

In our previous work, we described a knowledge-based framework for the purposeful abstraction of process models based on abstraction conditions specified on the process structure [11]. The framework is based on configuration techniques and can provide multiple views on different abstraction levels. A stakeholder provides an *abstraction condition* for generating a view: the condition consists of a set of constraints over properties of the business process and its activities. Based on the provided constraints the framework identifies which activities in the process are significant and which ones are insignificant for the stakeholder. It then applies stepwise abstraction and elimination operators to the insignificant elements and creates a hierarchy of abstracted process models where each level in the hierarchy represents a more abstract model until the most abstract model is found [12].

Example 2.2. In Fig. 3, process M_1 is a view for *City Car Rental* over the reference process M'. Activity *Cancel* in M_1 aggregates *Cancel Late*$'$ and *Send Cancellation Invoice*$'$ in M'.

For abstracting insignificant activities, the operator *AggregateActivities()* is applied. It aggregates a group of activities which form a pattern and abstracts them to a single composite activity, therefore creating one-to-many relationship from the abstract to the more detailed elements. A single insignificant activity, i.e., an insignificant activity that has pre- and post-states that are connected to significant activities, cannot be aggregated or eliminated otherwise it would violate the *observation consistency* criteria which is described in Section 2.3 below. The name of a new composite activity created by the *AggregateActivities()* is extracted from a meronymy tree of activities which represents the domain knowledge within our abstraction framework [12].

The correspondences between a process model and its views are captured by inheritance using a *mapping function* $h()$ while constructing the views. This allows us to check for consistency and change propagation during the design phase. The function $h()$ is a *total and surjective mapping function* such that:

(1) h maps each state, activity, data, and label of the reference process model to the elements of the view.
(2) h maps labels of the reference process to labels in the view. Different labels in the reference process model can be mapped to a single label in the view.
(3) h maps initial state of the reference process to the initial state of the process view.

Definition 2.3. (Process View)

A labeled Petri net diagram
$M = (N, F, D, \delta_{in}, \delta_{out}, L, l)$ is defined as a view of another labeled Petri net diagram $M' = (N', F', D', \delta'_{in}, \delta'_{out}, L', l')$ by the function $h_{M' \mapsto M}$ which has the following properties:

$h : N' \cup D' \cup L' \mapsto N \cup D \cup L \cup \{\epsilon\}$ (h is total and surjective),

(i) $e' \in N' \Rightarrow h(e') \in N \cup \{\epsilon\}$,
(ii) $e' \in D' \Rightarrow h(e') \in D \cup \{\epsilon\}$,
(iii) $l' \in L' \Rightarrow h(l') \in L \cup \{\epsilon\}$,
(iv) (1) $a' \in N'_s, |\bullet a'| = 0 \wedge h(a') = a \Rightarrow |\bullet a| = 0$ and (2) $l \in l(a') \Rightarrow h(l) \in l(h(a'))$,

(v) $h(e') = e', h(e'') = e' \Rightarrow e' = e''$,
(vi) $h(e) \in N' \cup L' \cup D' \Leftrightarrow h(e) = e$

Property (i) ensures that each element in the reference process is either mapped to an element in the view or eliminated, property (ii) specifies that each data variable in the reference process is either mapped to a data variable in the view or eliminated, property (iii) specifies that each label in the reference process is either mapped to a label in the view or eliminated, property (iv) maps (1) each initial state of the reference process to an initial state of the view and (2) each label associated with an initial state to its corresponding label in the view, property (v) ensures that if an element is preserved identically in a view then no other element can be mapped to it, and property (vi) specifies that elements that appear unchanged in a view are mapped to their identical elements in the reference process.

Inheritance of elements without change, abstraction and elimination of elements are expressed by h as follows:

(i) *Inheritance without change*: For an unchanged activity, state, or label e the mapping $h(e) = e$ holds.
(ii) *Abstraction*: For a set of activities and states E abstracted to an activity or state e in M, $\forall e' \in E : h(e') = e$ holds. Similarly, for a set of labels E abstracted to label l in M, $\forall l' \in E : h(l') = l$ holds.
(iii) *Elimination*: For a set of activities, states, or labels E that are eliminated in the view, $\forall e \in E : h(e) = \epsilon$ holds.

For data flow abstraction, we distinguish two kinds of data abstraction: *aggregation* and *generalization*, where *agg(e)* denotes data aggregated from e and *gen(e)* denotes data generalized from e where *agg(e)* with $e \in D$ expresses that e represents an abstraction that consists of a composition of each element e' such that $h(e') = e$, and where *gen(e)* with $e \in D$ expresses that e represents an abstraction that represents exactly one element e' such that $h(e') = e$.

Example 2.3. Fig. 3 shows two process views M_1 and M_2 that have been generated from the reference process M'. For M_1 an abstraction condition was specified by a stakeholder who focuses on the car rental aspect of M', and for M_2 an abstraction condition was specified for a stakeholder who is interested in the truck rental aspect of M. The resulting views are slightly different, e.g., the sequence (*Cancel Late'*,s'_7,*Send Cancellation Confirmation'*) in M' is aggregated to *Cancel* in M_1 whereas the sequence (s'_7, *Send Cancellation Confirmation'*, s'_4) is aggre-

gated to state s_8 in M_2. Data flow is aggregated in the views such that d_2 and d_3 are aggregated to d_8 in the views.

Later in Section 3 we discuss the co-evolution of process views where different stakeholders may change their own views. There are two possible changes to a view: (1) Changes of the *viewpoint* over a given reference process. This is handled by changing the abstraction condition. (2) Changes of the reference process via changes of the process view. This kind of change is motivated in that process views are simpler to understand for stakeholders because they are personalized and they can perform changes more easily on views rather than on a complex reference process model. We consider the latter type of change.

If elements in two different views which map to the same elements in the reference are changed simultaneously, then a conflict of changes may arise. In order to identify conflicts and resolve them we require the definition of a *region*. To compute a region in a reference process M', we invert function h. In other words, with respect to a view M a region in M' is the set of elements of M' which map to the same process element in M.

Definition 2.4. (Region)
Given a reference process $M' = (N', F', D', \delta'_{in}, \delta'_{out}, L', l')$, a view $M = (N, F, D, \delta_{in}, \delta_{out}, L, l)$ and its mapping $h : M' \mapsto M$, the region $R(e)$ in M' corresponding to an element $e \in N \cup D \cup L$ is defined as $R(e) = \{e' \in N' \cup D' \cup L' \mid h(e') = e\}$.

Example 2.4. The region $R(Cancel)$ in reference process M' with respect to view M_1 shown in Fig. 3 is $\{CancelLate', s'_7, SendCancellationInvoice'\}$.

Within a region we have two special types of activities (states): *source* and *sink activities (states)*. We call an activity (state) e' a source activity (state) of a region $R(e)$, if e' is in the refinement of activity (state) e and has an incoming arc from a state (activity) outside the refinement of e, and we call an activity (state) e' a sink activity (state) of e if e' is in the refinement of e and has an outgoing arc into a state (activity) outside of e. Sink- and source activities and states are required for linking to newly inserted activities (states) in a view as discussed in Section 2.4.

A process view must be consistent with its reference process. In the following section we describe the consistency criteria between a reference process and views, and how they can be checked using simply design rules.

2.3. *Checking Consistency of Views*

The problem of inconsistency between multiple views of a reference model is well-studied [23]. Consistency criteria help to identify contradictions between views and their reference process model once different stakeholders change the views. In our framework we apply *consistent specialization of business processes* to ensure the consistency between process views and their respective reference process model.

Specialization consists of *refinement* and *extension* where refinement refers to refining an activity into more detail and extension refers to adding new elements to a process model. A process model P' is an extension of process model P if it possesses new features relative to P. Likewise, process model P' is a refinement of a process model P if the inherited features are specified in more detail. In this sections we first discuss consistency criteria, then apply the criteria on the control flow aspect of processes and finally discuss data flow consistency in the context of the criteria.

2.3.1. *Consistency Criteria*

Schrefl and Stumptner [7] investigated two criteria for the consistent refinement and extension of business processes: *observation-* and *invocation consistency*. Informally, observation consistent specialization between a process model M and M' guarantees that if features added in process M' are ignored and features refined in M' are considered unrefined, any instance of M' can be observed as correct from the point of view of M.

Invocation consistency is divided into *weak-* and *strong invocation consistency*. Weak invocation consistency ensures that instance of a view M can be used in the same way as instances of a reference process M'. An extended version of this property, strong invocation consistency, guarantees that one can continue to use instances of M in the same way as instances of M', even after activities that have been added in M' have been executed.

In our framework we apply observation consistency and strong invocation consistency. We apply observation consistency to those activities and states in a process view which have been abstracted from a reference process and considered as insignificant according to the abstraction condition (see also Section 2.2). The reason for this is that stakeholders can observe that something is happening in their view which is the responsibility of another stakeholder. The same holds for data flow.

We apply strong invocation consistency to those activities that are significant to the stakeholder. We assume that by specifying an abstraction condition and identifying elements as significant means that the stakeholder has the interest or responsibility in invoking these activities. From now on we refer to *invocation consistency* when referring to *strong invocation consistency*.

Similar consistency criteria have been investigated in the past. Protocol- and projection inheritance introduced by Basten et al. [24] corresponds to weak invocation consistency and observation consistency respectively. Yongchareon et al. [25] studied consistent specialization in particular for artifact-centric business processes and multiple artifacts interacting with each other. The main advantage of the work by Schrefl and Stumptner [7] is a set of rules that allow us to efficiently check for consistency without analyzing all possible execution traces. In the following we introduce and apply those rules for checking control flow consistency.

Related work mentioned above has addressed consistency criteria on control flows so far. In this article we extend existing work to data flows and check the

data input and output for observation and invocation consistency. Below, we first address the control flow consistency between reference process and views and then address the data flow consistency.

2.3.2. *Checking Control Flow Consistency*

We check for observation consistency (OC) for elements that are abstracted and insignificant in a view and for invocation consistency (IC) for activities that are significant. Schrefl and Stumptner [7] introduced simple rules that are necessary and sufficient for checking OC and IC between two processes. The rules can be applied during design time on processes that have been refined or extended. These rules have been adapted for the context of this article which is a reference process and related process views: Rules R1 (Fig. 4) and R2 (Fig. 5 and 6) allow for checking OC, whereby R2.1 considers abstraction and R2.2 elimination of activities. Rules R3 (Fig. 7) allow for checking IC. Note that OC is necessary for IC as discussed in [7]. We explain the rules in the context of process views in the following.

Fig. 4 shows the pre- and post-state satisfaction rules R1. The pre- and post-state satisfaction rules verify that the behavior of a view is correctly reflected in the detailed model: (a1) each sequence flow entering an activity in the view arises from a corresponding inbound flow in the detailed model; (a2) there is a corresponding sequence flow in the detailed model for each sub-label of an abstract label. The first constraint ensures that sequence flows in the abstract and detailed model are consistent with each other, whereas the second ensures that the refinement of states in the detailed model is complete with respect to the view. Rules (b1) and (b2) are analogous for the outbound flows of an activity.

Example 2.5. Consider reference process M' and view M_1 in Fig. 3. Transition *Cancel Late*$'$ in M' consumes from state s'_2 and *Send Cancellation Invoice*$'$ produces into s'_4. To satisfy the pre- and post- satisfaction rule from Fig. 4 transition *Cancel* in M_1 must consume from at least one state and produce in at least one state as in M'. This is the case as transition *Cancel* consumes from s'_2 where $h(s'_2) = s'_2$ holds and *Cancel* produces into s'_4 where $h(s'_4) = s'_4$.

For checking OC also rules R2 shown in Fig. 5 and 6 need to be checked. The rules for pre- and post-state refinement R2.1 verify that all the behavior of the detailed model is considered in an abstraction. Specifically, behavior consistency requires that control flows in the detailed model that are not hidden in an abstraction must be reflected in the abstract model as flow with consistent label abstraction. Rule R2.1(a1) and (a2) verify conditions for sequence flows inbound to an activity, and R2.1(b1) and (b2) verify the corresponding conditions for outbound sequence flows.

Example 2.6. The rules R2.1 (b1) and (b2) are satisfied for post state s_8 of activities *Cancel Late*$'$ and *Cancel Early with Invoice* in M_2 shown in Fig. 3.

If elements are eliminated in a process view, rules R2.2 shown in Fig. 6 need to

R1. Pre and post-state satisfaction
(a1) $t \in N_t, s \in N_s, \hat{s} \in N_s, t' \in N'_t, s' \in N'_s :$
$h(t') = t \wedge h(s') = s \wedge (s,t) \in F \wedge (s',t') \in F' \wedge (\hat{s},t) \in F$
$\Rightarrow \exists \hat{s'} \in N'_s : h(\hat{s'}) = \hat{s} \wedge (\hat{s'},t') \in F'$
(a2) $t \in N_t, s \in N_s, t' \in N'_t, s' \in N'_s :$
$h(t') = t \wedge h(s') = s \wedge (s,t) \in F \wedge (s',t') \in F'$
$\wedge x \in l(s,t) \wedge h(x'') = x$
$\Rightarrow \exists s'' \in N'_s : h(s'') = s \wedge (s'',t') \in F' \wedge x'' \in l'(s'',t')$
(b1) $t \in N_t, s \in N_s, \hat{s} \in N_s, t' \in N'_t, s' \in N'_s :$
$h(t') = t \wedge h(s') = s \wedge (t,s) \in F \wedge (t',s') \in F' \wedge (t,\hat{s}) \in F$
$\Rightarrow \exists \hat{s'} \in N'_s : h(\hat{s'}) = \hat{s} \wedge (t',\hat{s'}) \in F'$
(b2) $t \in N_t, s \in N_s, t' \in N'_t, s' \in N'_s :$
$h(t') = t \wedge h(s') = s \wedge (t,s) \in F \wedge (t',s') \in F'$
$\wedge x \in l(t,s) \wedge h(x'') = x$
$\Rightarrow \exists s'' \in N'_s : h(s'') = s \wedge (t',s'') \in F' \wedge x'' \in l'(t',s'')$

Fig. 4: Rules for checking pre- and post-state satisfaction of an activity t in a view. Rules are adapted from Fig.7 in [7].

R2.1 Pre and post-state refinement:
$s' \in N'_s, t' \in N'_t, h(t') \neq \epsilon$:
(a1) $(s',t') \in F' \wedge h(s') \neq h(t') \wedge h(s') \neq \epsilon$
$\Rightarrow (h(s'),h(t')) \in F$
(a2) $(s',t') \in F' \wedge h(s') \neq h(t') \wedge x' \in l(s',t') \wedge h(x') \neq \epsilon$
$\Rightarrow h(s') \neq \epsilon \wedge h(x') \in l(h(s'),h(t'))$
(b1) $(t',s') \in F' \wedge h(s') \neq h(t') \wedge h(s') \neq \epsilon$
$\Rightarrow (h(t'),h(s')) \in F$
(b2) $(t',s') \in F' \wedge h(s') \neq h(t') \wedge x' \in l(t',s') \wedge h(x') \neq \epsilon$
$\Rightarrow h(s') \neq \epsilon \wedge h(x') \in l(h(t'),h(s'))$

Fig. 5: Rules for checking pre- and post-state refinement of activity t' and state s' from the reference process in the process view. Rules are adapted from Fig.7 in [7].

be checked. Note that *parallel extension* is from the point of view of the reference processes meaning that the reference processes contains some elements (an extension to the model) which are omitted in the view.

Invocation consistency (IC) is checked only for those activities in a view which are identified as *invocable*. Given OC, rules R3 shown in Fig. 7 allow to check for IC. If rules are obeyed for an activity t in a view then t is invocation consistent. The

R2.2 Parallel extension:
$s' \in N'_s, t' \in N'_t, h(t') = \epsilon$:
(a1) $(s', t') \in F' \Rightarrow h(s') = \epsilon$
(a2) $(s', t') \in F' \wedge x' \in l(s', t') \Rightarrow h(x') = \epsilon$
(b1) $(t', s') \in F' \Rightarrow h(s') = \epsilon$
(b2) $(t', s') \in F' \wedge x' \in l(t', s') \Rightarrow h(x') = \epsilon$

Fig. 6: Rules for checking parallel extension of activity t' in the reference process which are eliminated in the process view. Rules are adapted from Fig.9 in [7].

R3 Invocation consistency:
(a) $t \in N_t : \mathit{invocable}(t) \Rightarrow R(t) = \{t\}$
(b) $s' \in N'_s, t' \in N'_t : \mathit{invocable}(h(t')) \wedge (s', t') \in F' \Rightarrow h(s') = s' \wedge (s', t') \in F$

Fig. 7: Rules for checking invocation consistency (IC) of invocable activity t in a process view.

rules ensure that (a) the activity t' from a reference process has not been changed in the view and (b) all pre-states of t' in the reference process are also in the view and connected to it.

Example 2.7. The activities *Assign Car'* and *Use'* are invocable in view M_1 and have the same pre- and post-states s'_1, s'_2, and s'_3 as in the reference process M'.

In the following we address consistency of data flows in process views.

2.3.3. *Checking Data Flow Consistency*

To check data flow consistency, we can apply a similar technique as described before. We assume that data input and output of an activity can also be abstracted by applying an extended version of the abstraction function $h()$ to data definitions in the reference process model to produce the data definitions in the views. We distinguish two kinds of abstraction: *aggregation* and *generalization*, where *agg(e)* denotes data aggregated from e and *gen(e)* denotes data generalized from e. Which one is applied depends on the domain knowledge specified in a part-of hierarchy (meronymy tree) [12]. A meronymy tree could be given for activity abstraction where a hierarchy of activities including data input and output is specified, or there could be an additional meronymy tree only for data types.

It is important to note that activities in a reference process can be abstracted into states in a view. Therefore states in a view can have input and output data.

As in the previous section where we introduced consistency criteria for control flow, we now describe corresponding criteria for data flow as represented by data input and data output of activities. The difference to the consistency rules for behaviors lies in the fact that we do not provide a full calculus for a complete data type system (of which there are many that are described in the literature). Rather, we assume minimal criteria of abstraction (aggregation and generalization) and the rules are defined in terms of the *agg* and *gen* abstractions. A choice of a particular data type framework will then determine the specific implementation of these operators but is beyond the scope of this paper.

- Rule D-R0 in Fig. 8 states that if data is abstracted in a view, then this must result in either an aggregation or a generalization of the original data element(s).
- Rule D-R1(a1) and (b1) shown in Fig. 8 ensure that each input (output) of a view node that is obtained by (possibly trivial) generalization is an abstraction of an input (output) of an activity in the detailed reference process. Rules D-R1(a2) and (b2) ensure that for each input (output) data formed by aggregation, all aggregated data components are indeed present on each execution path in the corresponding region in the reference process.
- Rule D-R2.1 shown in Fig. 9 ensures that input and output data of an activity in the reference process are also represented in the corresponding activity in the view.
- Rule D-R2.2 shown in Fig. 10 checks that if activities in a reference process have been removed in a view, their output is also removed in the view (D-R2.2 (b)) and that their input is removed unless it is also used as an input by another activity that is not removed (D-R2.2 (a)).

For invocable activities in a view, we need to check additional rules for invocation consistency: Rule D-R3 shown in Fig. 11 is applied only to invocable activities in a view and ensures that all data inputs are present and have not been changed in a view.

Note that we have not imposed a restriction on when exactly data elements are written during an activity. If activity t has input x (or output y), then this means that *sometime* during the execution of activity t, t reads x (or writes y). If n is a node in a view with input x (or output y), then sometime during execution of an activity t in the region of n, input x' with $h(x') = x$ is read (or, output y' with $h(y') = y$ is written).

Having defined the different consistency dimensions, in the following we discuss the application of changes to views in general and then how the consistency criteria are checked in the change propagation step.

D-R0. Data Abstraction
$\forall e' \in \delta'_{in}(x') \cup \delta'_{out}(x'), x' \in N'_t : h(e') = e \wedge e \neq e' \Rightarrow agg(e) \oplus gen(e)$,

D-R1. Rules for data input and output satisfaction
(a1) $n \in N, e \in D : e \in \delta_{in}(n) \wedge (gen(e) \vee h(e) = e) \Rightarrow$
in at least one possible execution trace of region $R(n)$ there exists a t' s.t. $\exists e' \in D' : h(e') = e \wedge e' \in \delta'_{in}(t')$.
(and there is only one such t' in that trace)
(a2) $n \in N, e \in D, e' \in D' : h(e') = e \wedge e \in \delta_{in}(n) \wedge agg(e) \Rightarrow$
in each possible execution trace of region $R(n)$ there exists t' s.t. $e' \in \delta'_{in}(t')$.
(b1) $n \in N, e \in D : e \in \delta_{out}(n) \wedge (gen(e) \vee h(e) = e) \Rightarrow$
in at least one possible execution trace of region $R(n)$ there exists a t' s.t. $\exists e' \in D' : h(e') = e \wedge e' \in \delta'_{out}(t')$.
(and there is only one such t' in that trace)
(b2) $n \in N, e \in D, e' \in D' : h(e') = e \wedge e \in \delta_{out}(n) \wedge agg(e) \Rightarrow$
in each possible execution trace of region $R(n)$ there exists t' s.t. $e' \in \delta'_{out}(t')$.

Fig. 8: Rules for checking data input and output satisfaction.

D-R2.1 Rules for checking data input and output of refined activity:
(a) $n' \in N', e' \in D' : e' \in \delta'_{in}(n') \wedge h(n') \neq \epsilon \wedge h(e') \neq \epsilon \Rightarrow h(e') \in \delta_{in}(h(n'))$.
(b) $n' \in N', e' \in D' : e' \in \delta'_{out}(n') \wedge h(n') \neq \epsilon \wedge h(e') \neq \epsilon \Rightarrow h(e') \in \delta_{out}(h(n'))$.

Fig. 9: Rules for checking data flow of non-eliminated activities.

D-R2.2 Rules for checking consistent data deletion:
(a) $t' \in N'_t, e' \in D' : e' \in \delta'_{in}(t') \wedge h(t') = \epsilon \wedge$
$(\nexists t'' \in N' : e' \in \delta'_{in}(t'') \wedge h(t'') \neq \epsilon) \Rightarrow h(e') = \epsilon$
(b) $t' \in N'_t, e' \in D' : e' \in \delta'_{out}(t') \wedge h(t') = \epsilon \Rightarrow h(e') = \epsilon$

Fig. 10: Rules for checking data flow of eliminated activities.

2.4. *Performing Changes and Change Propagation*

Since we deal with the co-evolution of views, a stakeholder can perform certain updates on a view. Two types of changes are presented here: (1) inserting and

D-R3. Invocation consistency:
$t' \in N'_t, t \in N_t, e \in D, e' \in D'$:
(a) $invocable(t) \wedge e \in \delta_{in}(t) \wedge h(e') = e \Rightarrow e = e'$
(b) $invocable(h(t')) \wedge e' \in \delta'_{in}(t') \Rightarrow h(e') = e' \wedge e' \in \delta_{in}(h(t'))$

Fig. 11: Rules for checking data input of invocable activities.

(2) eliminating an activity. Inserting means that a new activity is added to the view and eliminating means that an activity is removed from the view. Additional change operations can be added to the framework. In order to perform an update, a list of operators are provided which consist of basic and composite operators [12]. As indicated in [26], a majority of change operators can be composed out of basic operators. Some examples of operators are presented below and shown in Fig. 12–16.

Let $P = (N, F, D, \delta_{in}, \delta_{out}, L, l)$ be the labeled Petri net representing the original process model, and let $P' = (N', F', D', \delta'_{in}, \delta'_{out}, L', l')$ denote the resulting model after a change operation has been applied. We assume that N (N') is partitioned into N_t and N_s (N'_t and N'_s).

Adding an activity $a \notin N_t$ with input data variables D_{in} and output variables D_{out} in place of a flow in F can be formalized as follows. We distinguish three cases: adding an activity between a state s and an existing activity t (operator *AddActivityAtInflow* shown in Fig. 12), between an existing activity t and a state s (operator *AddActivityAtOutflow* shown in Fig. 13), and creating a new alternative or parallel branch (operators *AddAlternativeBranch* and *AddParallelBranch* shown in Fig. 14 and 15).

The labeling invariants and absence of deadlocks must be checked for operators *AddAlternativeBranch()* and *AddParallelBranch()* to ensure a well-formed model is obtained. Otherwise, the model trivially satisfies the requirements for well-formed labeled Petri nets.

Currently, the operator *RemoveActivity()* is defined only for an activity a that has a single pre-state and a single post-state as shown in Fig. 16.

Operators for adding, removing and re-labeling flow arcs can be defined similarly by updating the F, L and l components accordingly. However, additional checks that verify safeness, absence of deadlocks, and labeling properties must be applied in order to ensure the resulting Petri net is a well-formed process model. Operators for adding and removing data inputs and outputs can be defined such that the resulting labeled Petri net differs only in the components D, δ_{in} and δ_{out}. These trivial operators have been omitted for brevity.

After an operator has been applied, OC and IC between the reference process and the view are checked as described above. If an inconsistency is detected then the user has to resolves the conflicts. If no inconsistency is detected then the changes are propagated to the reference process and other views.

$$
\begin{aligned}
&AddActivityAtInflow(a, D_{in}, D_{out}, (s,t), s') : \\
&\text{Precondition: } a \notin N_t \wedge (s,t) \in F \wedge s' \notin N_s \\
&\quad N'_t = N_t \cup \{a\} \\
&\quad N'_s = N_s \cup \{s'\} \\
&\quad F^+ = \{(s,a), (a,s'), (s',t)\} \\
&\quad F' = F \setminus \{(s,t)\} \cup F^+ \\
&\quad D' = D \cup D_{in} \cup D_{out} \\
&\delta'_{in}(a) = D_{in} \text{ and } \delta'_{in}(a') = \delta_{in}(a') \text{ for } a' \in N_t \\
&\delta'_{out}(a) = D_{out} \text{ and } \delta'_{out}(a') = \delta_{out}(a') \text{ for } a' \in N_t \\
&\quad L' = L \\
&\quad l'(f) = l((s,t)) \text{ for } f \in F^+ \text{ and } l'(f) = l(f) \text{ for } f \in F' \setminus F^+
\end{aligned}
$$

Fig. 12: Change operator *AddActivityAtInflow()* for inserting a new activity a between an existing state s and activity t in a process model.

$$
\begin{aligned}
&AddActivityAtOutflow(a, D_{in}, D_{out}, (t,s), s') : \\
&\text{Precondition: } a \notin N_t \wedge (t,s) \in F \wedge s' \notin N_s \\
&\quad N'_t = N_t \cup \{a\} \\
&\quad N'_s = N_s \cup \{s'\} \\
&\quad F^+ = \{(t,s'), (s',a), (a,s)\} \\
&\quad F' = F \setminus \{(t,s)\} \cup F^+ \\
&\quad D' = D \cup D_{in} \cup D_{out} \\
&\delta'_{in}(a) = D_{in} \text{ and } \delta'_{in}(a') = \delta_{in}(a') \text{ for } a' \in N_t \\
&\delta'_{out}(a) = D_{out} \text{ and } \delta'_{out}(a') = \delta_{out}(a') \text{ for } a' \in N_t \\
&\quad L' = L \\
&\quad l'(f) = l((t,s)) \text{ for } f \in F^+ \text{ and } l'(f) = l(f) \text{ for } f \in F' \setminus F^+
\end{aligned}
$$

Fig. 13: Change operator *AddActivityAtOutflow()* for inserting a new activity a between an existing activity t and state s in a process model.

Fig. 17 illustrates a change propagation scenario where activity *Inform* is inserted in M_2 after activity *Cancel Late'*:

(1) Apply changes: A stakeholder applies changes to a view using one of the

$AddAlternativeBranch(a, D_{in}, D_{out}, s, s') :$
Precondition: $a \notin N_t \wedge s \in N_s \wedge s' \in N_s \wedge \{s\} \triangleleft s' \wedge L_s = L_{s'}$
where $L_s \subseteq L$ and $L_{s'} \subseteq L$ are the set of labels associated with s and s' :

$$
\begin{aligned}
N'_t &= N_t \cup \{a\} \\
F^+ &= \{(s,a),(a,s')\} \\
F' &= F \cup F^+ \\
D' &= D \cup D_{in} \cup D_{out} \\
\delta'_{in}(a) &= D_{in} \text{ and } \delta'_{in}(a') = \delta_{in}(a') \text{ for } a' \in N_t, a' \neq a \\
\delta'_{out}(a) &= D_{out} \text{ and } \delta'_{out}(a') = \delta_{out}(a') \text{ for } a' \in N_t, a' \neq a \\
L' &= L \\
l'(f) &= L_s \text{ for } f \in F^+ \text{ and } l'(f) = l(f) \text{ for } f \in F,
\end{aligned}
$$

Fig. 14: Change operator *AddAlternativeBranch()* for inserting a new alternative branch represented by activity a between existing states s and s' in a process model.

$AddParallelBranch(s_1, s_2, s_3, s_4, l, t_1, t_2, t_3) :$
Precondition: $s_1, s_2, s_3, s_4 \notin N_s \wedge l \notin L \wedge t_1, t_3 \in N_t \wedge t_2 \notin N_t$

$$
\begin{aligned}
N'_t &= N_t \cup \{t_2\} \\
N'_s &= N_s \cup \{s_1, s_2, s_3, s_4\} \\
F^+ &= \{(s_1,t_1),(t_1,s_2),(s_2,t_2),(t_2,s_3),(s_3,t_3),(t_3,s_4)\} \\
F' &= F \cup F^+ \\
L' &= L \cup \{l\} \\
l'(f) &= \{l\} \text{ for } f \in F^+ \text{ and } l'(f) = l(f) \text{ for } f \in F
\end{aligned}
$$

Fig. 15: Change operator *AddParallelBranch()* for inserting a new parallel branch represented by t_2 between two activities t_1 and t_3 in a process model.

available operators.

Example 2.8. Activity *Inform* is inserted in view M_2 in Fig. 17.

(2) Check well-formedness of process view: A first correctness check includes checking for properties (well-formedness and data consistency). The properties can be checked efficiently using techniques such as SESE fragmentation [27]. If a violation of the properties is detected then the process model must be changed by the user so they are satisfied.

$RemoveActivity(a)$:

Precondition: $a \in N_t \wedge |\bullet a| = 1 \wedge |a \bullet| = 1$

$$
\begin{aligned}
N'_t &= N_t \setminus \{a\} \\
N'_s &= N_s \setminus a\bullet \\
F^- &= \{(s,a) \mid s \in \bullet a\} \cup \{(a,s) \mid s \in a\bullet\} \cup \{(s,t) \in F \mid s \in a\bullet\} \\
F^+ &= \{(t,s_0) \mid s_0 \in \bullet a, s_1 \in a\bullet, (t,s_1) \in F\} \\
F' &= F \setminus F^- \cup F^+ \\
D' &= D \\
\delta'_{in}(a') &= \delta_{in}(a') \text{ for } a' \in N'_t \\
\delta'_{out}(a') &= \delta_{out}(a') \text{ for } a' \in N'_t \\
L' &= L \\
l'(f) &= L_a \text{ for } f \in F^+ \text{ and } l'(f) = l(f) \text{ for } f \in F' \setminus F^+, \\
&\quad \text{where } L_a \subseteq L \text{ is the set of labels associated with the state in } \bullet a
\end{aligned}
$$

Fig. 16: Change operator *RemoveActivity()* for removing an existing activity from a process model.

Example 2.9. After activity *Inform* has been inserted, the process view M_2 in Fig. 17 remains safe, activity-reduced and deadlock-free.

(3) Check consistency of intended changes to reference process: It is checked whether the changes can be propagated to the reference process such that the reference process is well-formed and a consistent specialization of the view (checking OC and IC for control flow consistency and corresponding criteria for data flow consistency). For this, elements added to the view are added in a shadow copy $\hat{M}'$ of the reference process and elements removed from the view are removed from $\hat{M}'$. The new abstraction function $h_{new}()$ is altered for elements in $\hat{M}'$ as follows: $h_{new}(e) = h_{old}(e)$ if the element e has not been added and $h_{new}(e) = e$ if element e has been added. Flows between view elements have to be propagated to $\hat{M}'$ in a way to meet well-formedness and consistency criteria. For flow $(s,t) \in F$ with $s \in N_s$ and $t \in N_t$ and $x \in l(s,t)$, the flow can be uniquely propagated to F' of the shadow copy reference process if there exists for each $x' \in R_{new}(x)$ exactly one sink state $s' \in N'_s$ in region $R(s)$ such that $x' \in l'(s')$ [a]. In such a case this state s' is connected to any source activity t' in $R(t)$ to gain a flow in F'. If several such states exist designer input is required to make a choice. Correspondingly, the same procedure is applied to flows $(t,s) \in F$ with $s \in N_s$ and $t \in N_t$.

[a] l' is extended from flows to states; this extension is unique due to the property of common label distribution, see Section 2.1.

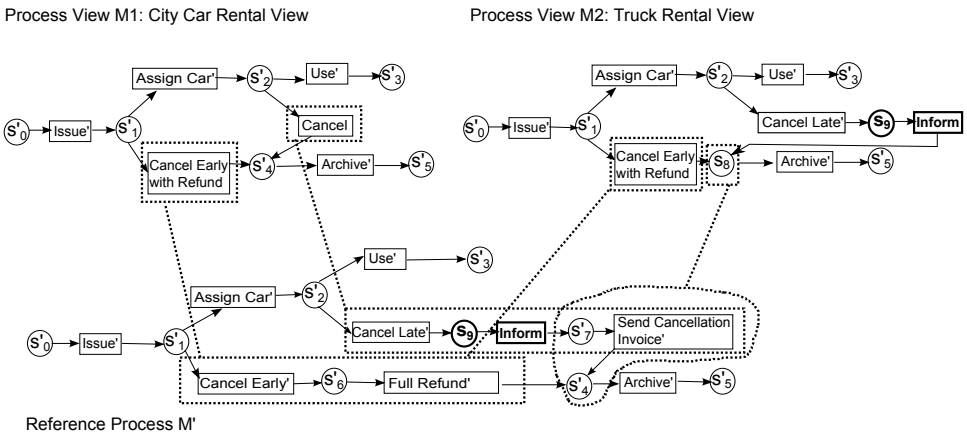


Fig. 17: Change propagation steps following the insert of s_9 and *Inform* in view *M2*, for clarity the control flow labels and data labels are omitted.

If resulting shadow reference process $\hat{M}'$ is well-formed and a consistent specialization of the view, the planned change is correct and can now be propagated from the shadow reference process to the reference process.

Example 2.10. State s_9 and activity *Inform* are added to a shadow reference process $\hat{M}'$ which is a copy of M'. In $\hat{M}'$ the activity *CancelLate'* is connected to s_9 and activity *Inform* is connected to s'_7 which is the source state within region $R(s_8)$. The mapping between elements in $\hat{M}'$ and M_2 is updated with $h_{new}(s_9) = s_9$ and $h_{new}(Inform) =$*Inform*. Process $\hat{M}'$ is well-formed and observation consistent with the changed M_2 process.

The shadow process $\hat{M}'$ is well-formed and observation consistent with the changed M_2 process, and changes can therefore be propagated to M' as shown in Fig. 17.

(4) Propagate changes to the reference process: The reference process is replaced by the shadow reference process. A shadow copy of the view is generated from the reference process based on the existing abstraction condition. If the existing view and its shadow copy differ, additions may not manifest in the view as they are considered insignificant by the abstraction condition and do not meet the abstraction condition. Moreover, as a result of deletions, fewer elements can be represented in the view [12]. If added elements are considered insignificant, the stakeholder is given the opportunity to modify the abstraction condition.

(5) Propagate changes to other views:

The other views of the reference process are regenerated from the modified reference process and their abstraction functions are updated accordingly [12].

3. Conflict Detection and Resolution

In this section we consider conflicting changes where changes in multiple views affect common elements in the reference process in incompatible ways. We first discuss conflict detection and subsequently cover propagation and conflict resolution strategies in our framework. Conflicting changes are identified by *change regions*.

Definition 3.1. (Change Region)

A change region $C(o)$ of an applied change operator o is the union of regions $\hat{R}(e)$ of all elements $e \in params(o)$ that have been affected by o ($params(o)$ denotes the set of actual parameters passed to operator o): $C(o) = \bigcup_{e \in params(o)} \hat{R}(e)$ where
$R_{old} = R(e)$ with regard to h_{old},
$R_{new} = R(e)$ with regard to h_{new},

$$\hat{R}(e) = \begin{cases} R_{old}(e), & \text{if } e \text{ has been removed from view} \\ R_{new}(e), & \text{if } e \text{ has been added to view} \\ R_{new}(e) \cup R_{old}(e), & \text{if } e \text{ is in the view before and after the change} \end{cases}$$

The definition of *change region* is based on *region* (see Definition 2.4) where a region is the preimage of a view element with respect to function $h()$. If a change operator is applied then $h()$ is updated as described in Step 3 above. Depending on the type of change operator, the $h()$ function before ($h_{old}()$) or the $h()$ function after the change ($h_{new}()$) or both are used to determine a change region. If an element e is eliminated from a process view then the mapping function $h_{old}()$ before the change will be used, if a new element e is inserted in a process view then the mapping function $h_{new}()$ after the change is used, and if e is in the view before and after the change then the union of the regions before and after the change is used.

Example 3.1. If the change operator o =*RemoveActivity(Cancel)* with $params(o) = \{Cancel\}$ is applied to M_1 in Fig. 3 then $C(o) = \{CancelLate', s_7', SendCancellationInvoice'\}$.

Example 3.2. If the change operator
o=AddActivityAtOutFlow(Inform,{reservationNumber},{confirmationNo},
(CancelLate',s_8*)},*s_9*)* with $params(o) = \{CancelLate', s_9, Inform, s_8\}$ is applied to M_2 in Fig. 3 then $C(o) = \{CancelLate', s_7', SendCancellationInvoice', s_4'\}$. Newly added elements are shown in Fig. 17.

Definition 3.2. (Conflicting Changes)

The changes of two operators o_1 and o_2 in different views are in conflict if their respective change regions $C(o_1)$ and $C(o_2)$ are overlapping, i.e., $C(o_1) \cap C(o_2) \neq \emptyset$.

Example 3.3. Imagine the stakeholder of view M_1 in Fig. 3 adds a new parallel branch from activity *Use* and at the same time the stakeholder for view M_2 removes *Use*. This would lead to a conflict because the elements *Use* in both views are in conflicting change regions.

For conflict resolution we assume a *view condition* associated to each process

view which specifies the subset of possible instances of the reference processes that can run through the process view. The view condition is a constraint over the property values of a process instance.

Example 3.4. The *view condition* of view M_1*: City Car Rental View* is *CarType='City Car'* and for view M_2*: Truck Rental View* it is *CarType='Truck'*.

If two views are in conflict then there exist two situations depending on the relation between their respective *view condition*: (1) The view conditions are disjoint or (2) they are overlapping. We assume that the view condition of two different views are not identical, i.e., if two stakeholders created a view with the same abstraction condition then they are sharing the same view with the same view condition. Below some principles of the approach are described first and then the two situations are addressed. In the remaining section we refer to *view condition* with the term *condition*.

In this section we discuss conflict resolution based on *type discrimination* on the process instances. The idea is to create a subtype of the reference process model for each view and then propagate changes to a shadow copy of the reference process which becomes the process of a subtype. Each subtype is associated with the view condition of the view for which the subtype was created and the association is called *membership condition*. During run time the reference process checks the properties of an instance and by type discrimination based on the membership condition directs the instances to the proper process of a subtype.

We first describe how two conflicting views can be resolved and then how the same technique can be applied to multiple views.

3.1. *Process Subtyping*

Fig. 18 shows the conceptual model in the state where a conflict has been detected and changes have not yet been propagated to the reference process M'. For simplicity, the figure shows only one view M_1. For each view M_1 and M_2, subtypes M_1' and M_2' of M' are created and associated with membership conditions identical to the respective view conditions of M_1 and M_2.

Depending on the view conditions there are two possibilities: (1) two views hold disjoint view conditions or (2) two views hold overlapping view conditions. Change conflicts in the two situation need to be handled differently which is discussed in the following subsections.

3.2. *Conflicting Views with Disjoint View Conditions*

The conceptual model of two views with disjoint conditions is shown in Fig. 19.

Example 3.5. The example contains two conflicting views *City Car View* and *Truck View* as shown in Fig. 19. Imagine that the stakeholder responsible for the City Car view process refines an activity "Offer Insurance" and at the same time the

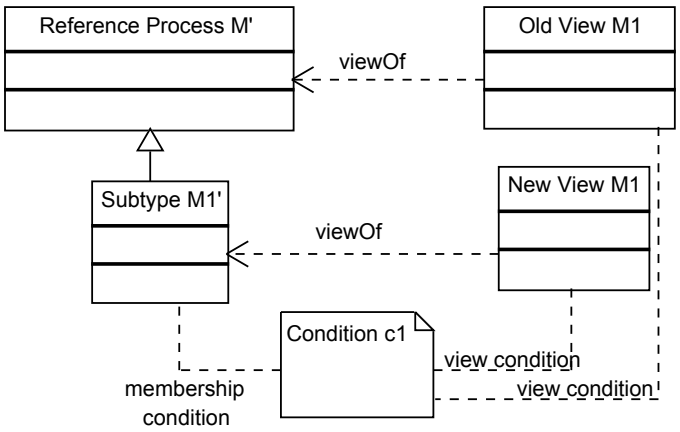


Fig. 18: Reference process M' with a new added subtype M_1'. Changes made in *Old View* M_1 are propagated to a shadow copy of M' which becomes the process of M_1'. The subtype M_1' holds the membership condition of *New View* M_1 which is the same as the view condition.

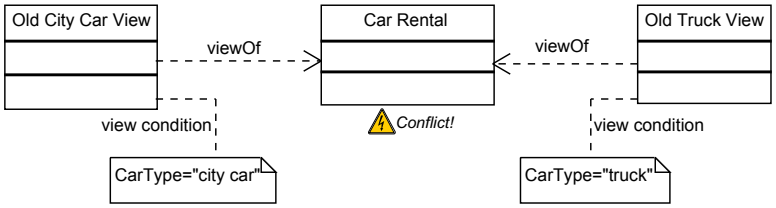
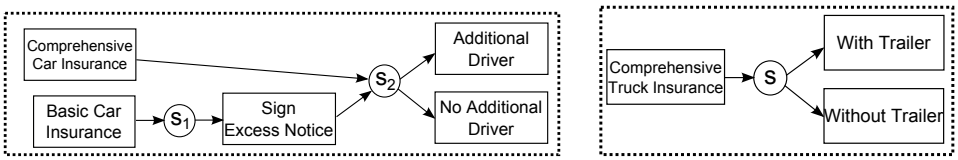


Fig. 19: Example of two conflicting views *City Car View* and *Truck View* with two disjoint conditions.

stakeholder of the truck view process refines the same activity differently. Fig. 20a and 20b show the different refinements. In case of a city car, the customer can choose either comprehensive or basic insurance. If basic insurance is chosen, then an excess notice needs to be signed. At the end, the client can choose to add an additional driver. In the truck view, a customer has to take comprehensive insurance and can then choose to also purchase insurance for a trailer, or not.

If two views M_1 and M_2 have disjoint conditions c_1 and c_2 then processes for subtypes M_1' and M_2' are generated. They are generated in the same way as explained in Step 3 for change propagation (see Section 2.4): Changes that were applied in M_1 are propagated to a shadow copy of reference process M'. This shadow copy becomes then the process of subtype M_1' which holds the membership condition identical to view condition c_1 of M_1. The same procedure is used for generating the process of subtype M_2' but with another shadow copy of M' to which changes



(a) Refinement in *City Car View*. (b) Refinement in *Truck View*.

Fig. 20: Activity "Offer Insurance" refined differently in *City Car* and *Truck View*.

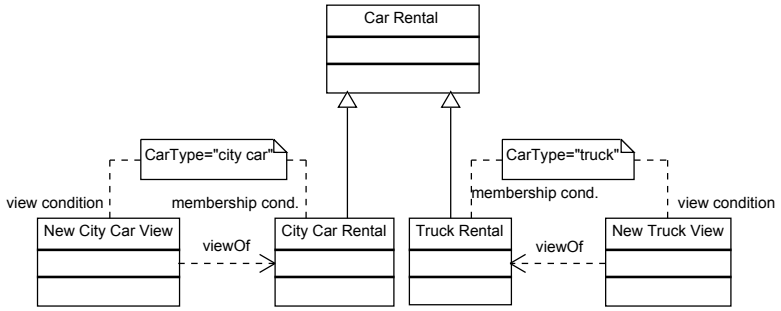


Fig. 21: Resulting conceptual model after conflicts indicated in Fig. 19 are propagated.

applied in M_2 are propagated. Views M_1 and M_2 are subsequently re-generated from M_1' and M_2' respectively.

Example 3.6. To resolve the conflict between *City Car* and *Truck View*, two subtypes *City Car Rental* and *Truck Rental* of the reference process *Car Rental* are created. The membership condition *CarType="city car"* is associated with the subtype *City Car Rental* and condition *CarType="truck"* with *Truck Rental*. The refinement of activity "Offer Insurance" in view *City Car* shown in Fig. 20a is propagated to a shadow copy of process *Car Rental*. This shadow copy becomes then the process of subtype *City Car Rental*. Likewise for the *Truck View*: Its refinement shown in Fig. 20b is propagated to a new shadow copy of *Car Rental* which becomes the process of subtype *Truck Rental*. The new views are regenerated from the processes of *City Car Rental* and *Truck Rental* as shown in the resulting conceptual model in Fig. 21.

At run time, process *Car Rental* can use the membership conditions associated with its subtypes to direct the instances to the appropriate subtype process by type discrimination based on the values of the process instance properties.

In order to use type discrimination for resolving conflicts, it is important to note that we assume the *abstract superclass rule*: each process instance that is executed on M' is an instance of a most specific process type. This ensures that the type

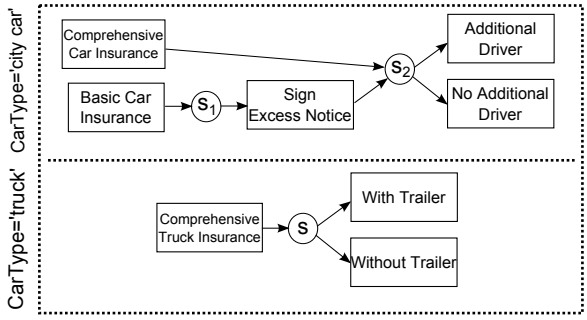


Fig. 22: Activity "Offer Insurance" in reference process *Car Rental* with expanded refinements shown in Fig.20a and 20b.

discriminator can assign the correct process region to an instance at run-time.

The main reference process M' is rebuilt as observation consistent generalization of the different process subtypes. More details on how to create an observation-consistent generalization of two processes can be found in work by Preuner et al. [28]. The relationship between a process subtype M'_i and the main reference process M' is captured by a mapping function h_i. An element e of the main reference process may, considering the mapping functions h_1 and h_2 of two subprocesses, have different regions R_{h1} and R_{h2}, if the region covered by e is handled differently in each subprocess. The reference process M' may expand e into lanes to show the different regions. An example is given below.

Example 3.7. Fig. 22 shows the two refinements made for "Offer Insurance" in *City Car View* and *Truck View*, one in each lane. The membership condition is shown on the left hand side. If *CarType="city car"* of a process instance is satisfied then the activities in the upper lane are executed and if *CarType="truck"* is satisfied then refinement in the lower lane is executed.

3.3. *Conflicting Views with Overlapping View Conditions*

If views M_1 and M_2 have overlapping conditions c_1 and c_2 then a more advanced approach needs to be taken compared to disjoint conditions. In addition to the two subtypes M'_1 and M'_2 previously created three new subtypes that are subtypes of M'_1 and M'_2 are generated. The following membership conditions are associated with them:

- M'_1 is a subtype of M' and associated with condition c_1,
- M'_2 is a subtype of M' and associated with condition c_2,
- M'_3 is a subtype of M'_1 and associated with condition $c_1 \setminus c_2$,
- M'_4 is a subtype of M'_2 and associated with condition $c_2 \setminus c_1$, and
- M'_5 is a subtype of M'_1 and M'_2 and associated with condition $c_1 \wedge c_2$.

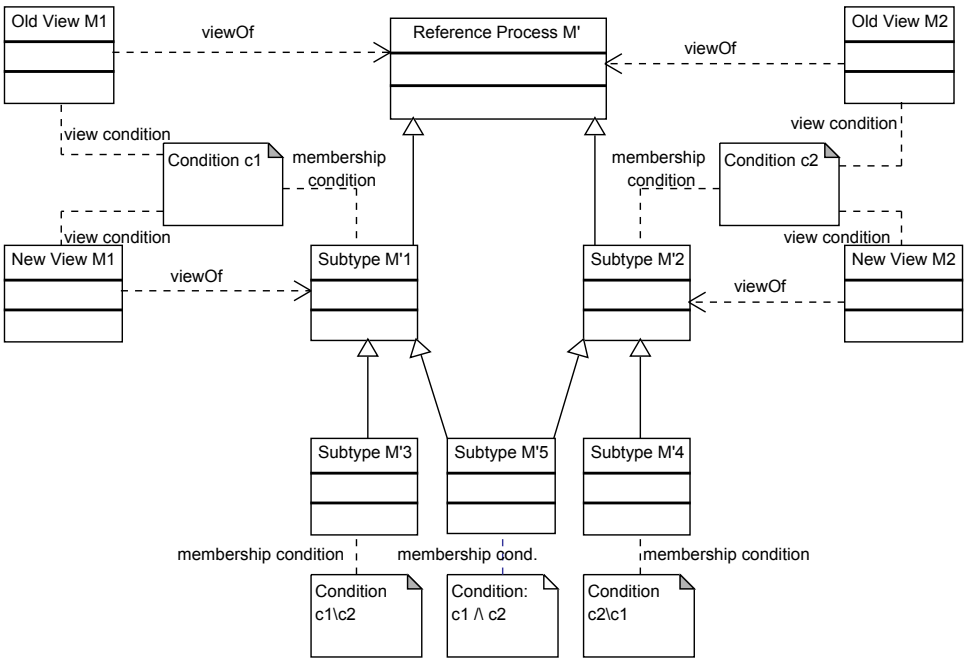


Fig. 23: Conceptual model of how to resolve conflicting changes in views with overlapping view conditions. Five subtypes $M'_1 - M'_5$ are created where M'_1 is a most specific observation consistent generalization of M'_3 and M'_5, and M'_2 is a most specific observation consistent generalization of M'_4 and M'_5.

The situation is illustrated in Fig. 23. Changes applied in M_1 are propagated to a shadow copy of reference process M' which becomes the process of subtype M'_3. Likewise for changes applied in M_2, a shadow copy of M' with propagated changes is created which becomes the process of subtype M_4. How the processes for M'_1, M'_2 and M'_5 are determined is explained below.

Example 3.8. Imagine two views as shown in Fig. 23 where M_1 is a "City Car View" and M_2 is a "VIP Customer View" which handles VIP customers. A stakeholder of M_1 refines activity "Offer Insurance" as shown in Fig. 20a whereas stakeholder of M_2 refines the same activity by adding "Special VIP Insurance". In case of a process instance where a regular customer rents a city car the process of subtype M'_3 is followed. In case of a VIP customer who does not rent a city car the behavior of subtype M'_4 is followed. In case a VIP guest rents a city car the behavior of subtype M'_5 is followed. This process can be specified in different ways.

Subtype M'_5 covers instances that hold condition $c_1 \wedge c_2$. The process can be specified in three different ways:

(1) Select one subtype as the default process, i.e., either the process of M'_3 or M'_4.

(2) Select the process prior to the change, i.e., the process of reference process M' before changes are propagated.
(3) Create a new process for subtype M'_5.

The processes of remaining subtypes M'_1 and M'_2 are created by the *most specific observation-consistent generalizations* [28]. M'_1 is a most specific observation-consistent generalizations of M'_3 and M'_5, and M'_2 is a most specific observation-consistent generalizations of M'_4 and M'_5. The new views M_1 and M_2 are generated from M'_1 and M'_2 respectively.

Above we have discussed how to resolve two conflicting views by type discrimination. If there are more than two views with conflicts and their conditions are disjoint then the same approach as described in Section 3.2 is applied. If conditions are overlapping then all possible combination of overlapping conditions need to be considered.

4. Case Study

The presented framework has been applied to real-world processes from the *Product and System Engineering* domain [12]. In the use case five reference process models have been considered: (1) The Product Management process described the product life cycle management and included activities such as planning, defining, forecasting or marketing of a product within a company, (2) the System Integration process described the development and execution of an integration plan for a product life cycle, (3) the Requirement Engineering process covered activities related to analysis, definition, validation, and updating of requirements of a product during its life cycle, (4) the Architecture and Design process described how to create and design an architecture and interfaces, and (5) the Software Development process described the implementation and testing of a software product.

The size and complexity of the five reference process models varied. The largest model was the reference process for Product Management with 258 activities and 654 control- and data flows and the smallest was Software Development with 146 activities and 532 control- and data flows. By generating views the size and complexity could be reduced significantly depending on the abstraction condition. For the five processes we selected an abstraction condition on the roles that execute activities, i.e., those activities executed by a specific role were significant: For Product Management and System Integration the role was "project manager", for Requirement Engineering it was the "requirements manager", for the Architecture and Design the role was "architecture", and for Software Development the role was "architect". After applying the abstraction, the size of the process views were between 8–34% of the reference process in respect to numbers of activities. This helped to focus on the relevant activities and made it easier to apply changes to them.

All reference processes were modeled in the Event-driven Process Chains (EPC) notation within the ARIS Framework. For applying presented framework on the processes, the EPC diagrams were first transformed into labeled Petri nets based

on related work that investigated the mapping of EPC to Petri nets [29]. The labels were added manually following the well formedness properties described in Section 2.1.

A prototype of the modeling environment for process models, abstraction, change operators and consistency checking is based on the *Domain Modeling Environment (DoME)* [b] implemented in Cincom's VisualWorks Smalltalk system. The architecture of the implementation is explained in more detail in [6]. Our implementation provides graphical editors for modeling business processes and properties of activities, and an implementation of the abstraction computation mechanism based on abstraction constraints. The user provides a detailed model as input, specifies the constraints and, after executing the configurator, obtains a hierarchy of abstractions providing different views. We do not currently provide a graphical user interface for constraint acquisition. Once the views are generated, changes can be performed on them and consistency checks can be executed.

5. Related Work

The following subsections discuss related work directly addressing change propagation between process models as well as the approaches which are potentially useful in the co-evolution of process models such as finding corresponding process model elements. All the mentioned approaches complement one aspect presented in this paper. The following subsections discuss the three main aspects in change management: *process model alignment*, *co-evolution* and *variability*. The results of comparison of related work have been captured in Table 1 where rows specify the papers and indicate their support with respect to the following criteria:

Goal: This column classifies the related work based on their goal.

Data Flow (Consistency): The data flow column indicates whether the related work considers data flow consistency in a change propagation scenario. This criterion is important as the robustness of a process model is fundamentally depended on the correctness of the data flow [1]. The possible values include supported, not supported, the criterion is not in the scope of the paper, and the criterion is in the scope of the paper but it has not been discussed.

Bidirectional Change Propagation: This column indicates the support for top down and bottom up change propagation where different levels of the abstraction hierarchy have been considered by the approach. This is an important aspect since changes cannot be restricted to any particular abstraction level.

Support for Process Views: This column indicates the support for different process views.

Consistency Preservation: The column discusses whether consistency between different models is maintained during change propagation. This is an important criterion as in practice there is no single process model but a group of process

[b] http://sml.cis.unisa.edu.au

models. In order to preserve the uniformity between such models, the consistency between them must be evaluated against formal criteria.

Change Operators: This column indicates whether the change propagation technique is based on change operators or not. Using an operator based approach has advantages such as identifying and enabling the traceability of the changed region.

Abstraction Hierarchy: This column discusses whether process models at different levels of abstraction are considered, as in practice process models are usually at different levels.

Conflict Detection and Resolution: These columns specify whether the proposed approach supports conflict detection and resolution in the change propagation scenario.

Support for Behavioral Model: This column specifies whether related work supports process models such as BPMN and Petri net or if it focuses on static models such as UML class diagrams. This is an important criterion as the semantics of processes is more complex than the semantics of static models [30].

5.1. *Process Model Alignment*

One important aspect of propagating changes from one process model to another is finding corresponding elements in the two process models. Finding corresponding elements in two structures means that for each entity in one structure, a corresponding entity in the other structure, with the same intended meaning can be found. To find the correspondences between models at different abstraction levels, Dijkman et al. [31] use lexical matching of element pairs and graph matching.

Brockmans et al. [32] propose a technique for semantic alignment of Petri net models at different levels of abstraction by translating the process model to OWL. For this purpose they consider the syntactic and semantic similarity between model elements.

Branco et al. [33] deal with management of consistency between models at different levels of abstraction. They propose an algorithm which establishes corresponding nodes between the models belong to different abstraction levels based on the type and name of the nodes.

Weidlich et al. [34] represent a framework for matching process model elements. The focus is mainly on the computational complexity of maintaining correspondences between elements in different models. Their approach focuses on correspondences between two models, whereas our work is applicable in propagation scenarios involving a set of related process models.

Finally, several approaches deal with producing different process views considering different requirements [9, 35–37]. However their focus is not on the change propagation between these views.

5.2. *Process Co-Evolution*

Kurniawan et al. [38] propose two algorithms for identifying changed region as well as propagating the changes. In this framework each process model is mapped to a node, and relationships between models (such as part-whole, generalization-specialization, and inter-operation) are expressed as edges between nodes. When one of the processes changes, a process model is revised to correct the relationship violation using a repair algorithm searching for a variant which violates consistency the least. The repair procedure aims to ensure that the relationship constraints which were satisfied before the change must be satisfied after the change and the chosen variant must have the minimum number of conflicts with the remaining processes.

Weidlich et al. [39] use *behavioral profiles* for identifying the changed regions in a model and propagate the changes to other models at different levels of abstraction. The changed region can be identified since any changes in the structure of a process model impact the relation of nodes (i.e. behavioral profile). In this approach data properties are not considered.

Kolb et al. [4,5,40] present the *proView* framework by which personalized process views over a central process model can be created. The framework supports control- and data flow, provides a library of change operators for the evolution of views, and updates associated process models accordingly. In comparison to our framework, conflicting changes are not considered and this work does not discuss consistency criteria of related process models.

Fdhila et al. [41] propose a generic approach to propagate changes in collaborative process maintenance scenarios. In such scenarios different business partners have their own private processes from which a public view is provided for other partners. The proposed approach deals with propagating the changes from a changed view to others in order to preserve invocation consistency. In comparison to our work, if the consistency of a changed view with other view cannot be preserved, the change is abandoned. There is no repair plan by which the identified inconsistency can be resolved.

Weidmann et al. [42, 43] propose a synchronization method to align changed process models with IT. The changes are propagated based on model correspondences between models at different abstraction levels, using a change queue. When a change occurs, it is logged in the change queue of the relevant model and is propagated to all synchronized models with the changed model. In comparison to our approach the work does not address a solution for conflicting changes, focuses on alignment of activities only, and does not consider gateways or activities properties including data flows.

Dam et al. [44] propose an agent-oriented framework for fixing inconsistencies resulting from change propagation. For this purpose they create a library of plans for fixing the constraint violations at run time. Although the completeness and correctness of generated plans are guaranteed, since all the possible repair plan

choices are considered, the approach is not scalable. Our approach relies on heuristic repairs and local consistency checks in order to provide scalable change propagation.

Ekanayake et al. [45] present a semi-automated technique for change propagation across process model versions. To keep the versions synchronized, changed fragments are locked until the change is propagated to all the other versions containing that changed fragment. In comparison to our framework, the approach does not deal with abstraction of processes on different levels of abstraction but focuses on a storage model for process fragments to facilitate their management.

Gerth et al. [46] propose a language independent framework for change management. The framework uses workflow graphs as an intermediate representation for the process model, by which the list of differences, dependencies, and conflicts are computed. This framework is applicable in versioning scenarios where different versions must be refinements of a source model. By merging different versions, the changes in the views are identified and propagated to the source model. The possibility that changes in one version may impact other versions is not considered and also the changes are always propagated from the views to the source model, not vice versa.

Favre et al. [47] present a prototype for the co-evolution of process views in an Eclipse-based graphical BPMN 2.0 editor. It provides a limited set of operators and focuses on the control flow of process models. It provides the option of user interaction, e.g., a propagated change can be accepted or neglected by a stakeholder. Main difference to the presented framework is missing discussion on consistency criteria and abstraction techniques applied.

5.3. *Conflict Detection and Resolution*

This section provides an overview over the approaches in conflict detection and resolution.

Brosch et al. [48] deal with resolving conflicts between multiple versions of UML Class Diagrams while merging different modifications into a consolidated version. For this purpose they introduce a versioning system called *AMOR* which provides an exact conflict report in model merging as well as semi-automatic executable resolution strategies. The conflicts are resolved based on a set of predefined resolution patterns, and consistency preservation is not discussed.

Gerth et al. [49] outline the importance of conflict detection in version management and provide a definition for semantic and syntactic conflicts that resulted from applying conflicting change operators. They capture the process structure in process model terms used for conflict detection, and the applied change operators are gathered from the change logs. The focus of the paper is on providing a mechanism for detecting conflicts and not on resolving them.

Küster et al. [50] introduce the concept of conflicting and dependent change operations and represents a method for identifying them based on applied changes derived from change logs. In contrast to our approach preserving consistency is not

discussed. A disadvantage of the approaches which depend on change logs is that the tools may not provide change logs or process models that can be exchanged across tool boundaries [51].

Küster et al. [51] propose a systematic framework which detects and resolves conflicts by comparing the differences in the structure of a process model before and after change based on derived change operations. The proposed approach is not automated. Instead it assists process designers in resolving conflicts manually by classifying changes.

Taentzer et al. [52, 53] introduce two kinds of conflict detection for operation-based and state-based conflicts in model versioning based on graph theory. Operation based conflicts refer to conflicts resulting from changing the structure of a process model such as adding a node in one version and removing it in another version. The operation based conflicts are resolved by giving priority over one of the conflicting operators. For instance, an *add* operation has priority over a *delete* operation. A state-based conflict occurs if the final state of the process model after a change contradicts the consistency rules of the related modeling notation. These distinctions have been elaborated in [52] for *EMF-based* models. The modeling features of the models such as controlling, multiplicity, and ordered features are formalized as graph constraints. Conflict patterns are presented which illustrate the conflicting operations, such as *delete-use, delete-move, delete-update, update-update, move-move,* and *insert-insert*. For resolving any of these conflicts a strategy is presented. However it is not clear how changes are set to a pre-defined change operation. This can cause difficulties in the identification of the potential operations especially where the investigated versions differ significantly.

Dam et al. [54] propose an enterprise architectural (EA) description language for change propagation within an EA model. To resolve conflicts and inconsistency while propagating the changes, the framework devises repair plans based on consistency and well-formedness rules formulated in Alloy. A significant difference to our work is that it does not support business process models and does not deal with the abstraction of models.

Cicchetti et al. [55] propose a domain-specific language to control the conflicts resulting from cooperative changes over the same process model elements. This semi-automated approach is based on a meta model representation which enables detection of semantic and syntactic conflicts. Changed models are merged and suitable conflict resolutions are recommended. Similar to previous approach, it does not focus on process models and abstraction of models.

Rinderle et al. [56] present a formal framework for dealing with overlapping and non-overlapping process changes at instance and type level, as well as migration strategies for running process instances. By considering a behavioral equivalence relation, i.e. trace equivalence, disjoint and overlapping changes are determined. In [57] the framework has been extended to deal with structural conflicts between concurrent changes in both running instances in the process model before the change and instances which have been modified individually. In contrast to this work, the

presented framework does not deal with the adaptation of process instances to changed process models.

Table 1: Comparison of Process Model Change Propagation Approaches

	Goal	Data Flow	Bi-directional	Process View	Consistency Preservation	Operators	Abstraction Hierarchy	Conflict Detection	Conflict Resolution	Behavioral models
Process Model Alignment										
Dijkman [31]	Process Alignment	n/a	n/a	+	n/a	n/a	+	n/a	n/a	+
Brockmans [32]	Inter-operability	-	n/a	n/a	n/a	n/a	+	n.d.	n.d.	+
Branco [33]	Matching	-	n/a	+	n.d.	-	+	n/a	n/a	+
Weidlich [34]	Compatibility	-	n/a	n/a	+	n/a	-	n/a	n/a	+
Liu [35]	Create View	-	n/a	+	+	n/a	+	n/a	n/a	+
Eshuis [9]	Create View	-	n/a	+	+	n/a	+	n/a	n/a	+
Bobrik [36]	Create View	-	n/a	+	+	n/a	+	n/a	n/a	+
Zhao [37]	Create View	-	n/a	+	+	n/a	+	n/a	n/a	+
Process Co-Evolution										
Kurniawan [38]	Refinement Preservation	-	+	-	+	+	+	+	+	+
Weidlich [39]	Change Propagation	-	+	n/a	+	+	+	n.d.	n.d.	+
Kolb [4, 5, 40]	View Management	+	+	+	n.d.	+	-	n.d.	n.d.	+
Fdhila [41]	Compatibility	-	-	+	+	+	+	-	-	+
Weidmann [42, 43]	Change Propagation	-	+	-	+	+	+	-	-	+
Dam [44]	Change Propagation	-	+	n/a	+	+	-	-	-	-
Ekanayake [45]	Change Management	n.d.	+	-	-	n.d.	-	n.a.	n.a.	+
Gerth [46]	Change Management	-	-	-	n.d.	+	+	+	+	+
Favre [47]	Change Propagation	-	+	+	n.d.	+	-	-	-	+
Conflict Detection and Resolution										
Brosch [48]	Change Management	-	-	-	n.d.	+	+	+	+	-
Gerth [49]	Change Management	-	-	-	n.d.	+	+	+	+	+
Küster [50]	Conflict resolution	-	+	n/a	+	+	+	+	+	+
Küster [51]	Change Management	-	-	-	n.d.	+	+	+	+	+
Taentzer [52]	Change Management	-	-	-	n.d.	+	+	+	+	-
Dam [54]	EA Change Management	+	+	-	+	n.d.	-	-	-	-
Cicchetti [55]	Model Management	-	+	-	+	n.d.	-	+	+	-
Rinderle-Ma [56, 57]	Instance Adaptation	+	n.a.	n.a.	+	-	n.a.	+	+	+
Our Approach	Change Propagation	+	+	+	+	+	+	+	+	+

Legend: + *supported*, − *not supported*, n/a *not applicable*, n.d. *not discussed*.

6. Conclusion

We presented a framework for process co-evolution that applies the notion of consistent specialization for on-the-fly change propagation. We showed on an example how rules developed for checking consistent refinement and extension of business processes can be used to propagate changes efficiently and re-establish consistency among a reference process and its views. So far the framework supports change propagation on the model level, and the rules have been implemented in a process abstraction framework. Further we presented an extension for dealing with conflicts resulting from concurrent changes affecting the same process region. These conflicts are resolved by building a hierarchy of process types and apply type discrimination to assign process instances dynamically to view specific elements at run-time. As the next step in our research agenda we plan to deploy the framework to a distributed collaborative process design environment.

Acknowledgments

This research was partially funded by the Data to Decisions Cooperative Research Centre (D2D CRC).

References

1. M. Reichert and B. Weber, *Enabling Flexibility in Process-Aware Information Systems* (Springer, 2012).
2. M. Weidlich, M. Weske and J. Mendling, Change propagation in process models using behavioural profiles, in *Proc. of SCC*, (IEEE, 2009), pp. 33–40.
3. M. Weidlich, S. Smirnov, C. Wiggert and M. Weske, Flexab – Flexible Business Process Model Abstraction, in *CAiSE Forum*, (CEUR Workshops, 2011), pp. 17–24.
4. J. Kolb, K. Kammerer and M. Reichert, Updatable process views for user-centered adaption of large process models, in *Proc. ICSOC*, (7636), (Springer, 2012), pp. 484–498.
5. J. Kolb and M. Reichert, Data flow abstractions and adaptations through updatable process views, in *Proc. of SAC*, (ACM, 2013), pp. 1447–1453.
6. S. Mafazi, G. Grossmann, W. Mayer and M. Stumptner, Towards a reference architecture for the co-evolution of business processes, in *Proc. of the 7th IEEE EDOC Workshop on Evolutionary Business Processes (EVL-BP)*, (IEEE, 2014).
7. M. Schrefl and M. Stumptner, Behavior-consistent specialization of object life cycles, *ACM Trans. Softw. Eng. Methodol.* **11**(1) (2002) 92–148.
8. S. Smirnov, H. Reijers, M. Weske and T. Nugteren, Business process model abstraction: a definition, catalog, and survey, *Distributed and Parallel Databases* **30** (2012) 63–99.
9. R. Eshuis and P. Grefen, Constructing customized process views, *Data &; Knowledge Engineering* **64** (2008) 419–438.
10. M. Reichert, J. Kolb, R. Bobrik and T. Bauer, Enabling personalized visualization of large business processes through parameterizable views, in *Proc. of SAC*, (ACM Press, 2012), pp. 1653–1660.
11. S. Mafazi, W. Mayer, G. Grossmann and M. Stumptner, A knowledge-based approach to the configuration of business process model abstractions, in *Int. Workshop on Knowledge-Intensive Business Processes (KiBP 2012)*, *Vol.861*, (CEUR-WS, 2012).

12. S. Mafazi, G. Grossmann, W. Mayer, M. Schrefl and M. Stumptner, Consistent abstraction of business processes based on constraints, *Journal on Data Semantics* **3** (2014).
13. C. Gerth, J. Küster and G. Engels, Language-independent change management of process models, in *Proc. of MoDELS*, *LNCS 5795*, (Springer, 2009), pp. 152–166.
14. S. Mafazi, G. Grossmann, W. Mayer and M. Stumptner, On-the-Fly Change Propagation for the Co-evolution of Business Processes, in *Proc. of OTM*, *LNCS 8185*, (Springer, 2013), pp. 75–93.
15. J. M. Küster, J. Koehler and K. Ryndina, Improving Business Process Models with Reference Models in Business-Driven Development, in *Proc. of BPM 2006 Workshops*, *LNCS 4103*, (Springer, 2006), pp. 35–44.
16. M. Sabou, D. Richards and S. van Splunter, An experience report on using DAML-S, in *Proc. of WWW'03*, (ACM, 2003).
17. W. O. W. Group, Web Ontology Language (OWL), tech. rep., W3C (2002).
18. R. D. A. W. G. (DAWG), SPARQL Protocol and RDF Query Language, tech. rep., W3C (2008).
19. I. Horrocks, P. Patel-Schneider, H. Boley, S. Tabet, B. Grosof and M. Dean, SWRL: A Semantic Web Rule Language Combining OWL and RuleML, tech. rep., W3C (2004).
20. A. Felfernig, G. Friedrich, D. Jannach, M. Stumptner and M. Zanker, Configuration knowledge representation for semantic web applications, *AI EDAM* **17** (January 2003) 31–50, Special issue on Configuration.
21. R. M. Dijkman, M. Dumas and C. Ouyang, Semantics and analysis of business process models in BPMN, *Information & Software Technology* **50**(12) (2008) 1281–1294.
22. G. Decker, R. M. Dijkman, M. Dumas and L. García-Bañuelos, Transforming BPMN Diagrams into YAWL Nets, in *Proc. of BPM*, (Springer, 2008), pp. 386–389.
23. M. Weidlich and J. Mendling, Perceived consistency between process models, *Information Systems* **37**(2) (2012) 80 – 98.
24. T. Basten and W. M. van der Aalst, Inheritance of behavior, *The Journal of Logic and Algebraic Programming* **47**(2) (2001) 47 – 145.
25. S. Yongchareon, C. Liu and X. Zhao, A Framework for Behavior-Consistent Specialization of Artifact-Centric Business Processes, in *Proc. of BPM*, *LNCS 7481*, (Springer, 2012), pp. 285–301.
26. B. Weber, S. Rinderle and M. Reichert, Change patterns and change support features in process-aware information systems, in *Proc. of CAiSE*, *LNCS 4495*, (Springer, 2007), pp. 574–588.
27. D. Fahland, C. Favre, B. Jobstmann, J. Koehler, N. Lohmann, H. Völzer and K. Wolf, Instantaneous Soundness Checking of Industrial Business Process Models, in *Proc. of BPM 2009*, *LNCS 5701*, (Springer, 2009), pp. 278–293.
28. G. Preuner, S. Conrad and M. Schrefl, View Integration of Behavior in Object-Oriented Databases, *DKE* **36**(2) (2001) 153–183.
29. B. van Dongen, W. van der Aalst and H. Verbeek, Verification of EPCs: Using reduction rules and petri nets, in *Advanced Information Systems Engineering*, *LNCS 3520*, (Springer Berlin / Heidelberg, 2005), pp. 1–35.
30. C. Gerth, J. Küster, M. Luckey and G. Engels, Detection and resolution of conflicting change operations in version management of process models, *Software & Systems Modeling* **12**(3) (2013) 517–535.
31. R. Dijkman, M. Dumas, L. Garcia-Banuelos and R. Kaarik, Aligning business process models, in *Proc. of EDOC*, (IEEE, 2009), pp. 45–53.
32. S. Brockmans, M. Ehrig, A. Koschmider, A. Oberweis and R. Studer, Semantic alignment of business processes, in *Proc. of ICEIS*, (INSTICC Press, 2006), pp. 191–196.

33. M. Castelo Branco, J. Troya, K. Czarnecki, J. Küster and H. Volzer, Matching business process workflows across abstraction levels, in *Proc. MoDELS*, *LNCS 7590*, (Springer, 2012), pp. 626–641.
34. M. Weidlich, R. M. Dijkman and M. Weske, Deciding behaviour compatibility of complex correspondences between process models, in *Proc. BPM*, (Springer, 2010), pp. 78–94.
35. D.-R. Liu and M. Shen, Workflow modeling for virtual processes: an order-preserving process-view approach, *Information Systems* **28**(6) (2003) 505 – 532.
36. R. Bobrik, M. Reichert and T. Bauer, View-based process visualization, in *Proc. of BPM*, *LNCS 4714*, (Springer, 2007), pp. 88–95.
37. X. Zhao, C. Liu, W. Sadiq and M. Kowalkiewicz, Process view derivation and composition in a dynamic collaboration environment, in *Proc. of OTM*, *LNCS 5331*, (Springer, 2008), pp. 82–99.
38. T. Kurniawan, A. Ghose, H. Dam and L.-S. Le, Relationship-preserving change propagation in process ecosystems, in *Proc. of ICSOC*, *LNCS 7636*, (Springer, 2012), pp. 63–78.
39. M. Weidlich, J. Mendling and M. Weske, Propagating changes between aligned process models, *Journal of Systems and Software* **85**(8) (2012) 1885–1898.
40. J. Kolb and M. Reichert, A flexible approach for abstracting and personalizing large business process models, *ACM SIGAPP Applied Computing Review* **13**(1) (2013) 6–18.
41. W. Fdhila, S. Rinderle-Ma and M. Reichert, Change propagation in collaborative processes scenarios, in *Proc. of CollaborateCom 2012*, (IEEE, 2012), pp. 452–461.
42. M. Weidmann, M. Alvi, F. Koetter, F. Leymann, T. Renner and D. Schumm, Business process change management based on process model synchronization of multiple abstraction levels, in *Proc. SOCA*, (IEEE Computer Society, 2011).
43. M. Weidmann, F. Koetter, T. Renner, D. Schumm, F. Leymann and D. Schleicher, Synchronization of Adaptive Process Models Using Levels of Abstraction, in *Proc. of EDOC Workshops*, (IEEE, 2011), pp. 174–183.
44. K. H. Dam and M. Winikoff, Generation of repair plans for change propagation, in *Proc. of AOSE*, (Springer, 2007), pp. 30–44.
45. C. Ekanayake, M. Rosa, A. Hofstede and M.-C. Fauvet, Fragment-based version management for repositories of business process models, in *Proc. CoopIS*, *LNCS 7044*, (Springer, 2011), pp. 20–37.
46. C. Gerth, *Business Process Models. Change Management* (Springer, 2013).
47. C. Favre, J. Küster and H. Völzer, The shared process model, in *Demo Track 10th Int'l Conf on Business Process Management (BPM'12)*, (Springer, 2012).
48. P. Brosch, G. Kappel, M. Seidl, K. Wieland, M. Wimmer, H. Kargl and P. Langer, Adaptable model versioning in action, in *Modellierung*, *LNI 161*, (GI, 2010), pp. 221–236.
49. C. Gerth, J. Küster, M. Luckey and G. Engels, Precise detection of conflicting change operations using process model terms, in *Proc. MoDELS*, *LNCS 6395*, (Springer, 2010), pp. 93–107.
50. J. Küster, C. Gerth and G. Engels, Dependent and conflicting change operations of process models, in *Model Driven Architecture - Foundations and Applications*, *LNCS 5562*, (Springer, 2009), pp. 158–173.
51. J. Küster, C. Gerth, A. Frster and G. Engels, Detecting and resolving process model differences in the absence of a change log, in *Proc. of BPM*, *LNCS 5240*, (Springer, 2008), pp. 244–260.
52. G. Taentzer, C. Ermel, P. Langer and M. Wimmer, A fundamental approach to model

versioning based on graph modifications: from theory to implementation, *Software & Systems Modeling* **11**(2) (2012) 1–34.
53. G. Taentzer, C. Ermel, P. Langer and M. Wimmer, Conflict detection for model versioning based on graph modifications, in *Proc. of Graph Transformations*, *LNCS 6372*, (Springer, 2010), pp. 171–186.
54. H. K. Dam, L.-S. Le and A. Ghose, Supporting change propagation in the evolution of enterprise architectures, in *Proc. of EDOC*, (IEEE Press, 2010), pp. 24–33.
55. A. Cicchetti, D. Ruscio and A. Pierantonio, Managing model conflicts in distributed development, in *Proc. of MODELS*, *LNCS 5301*, (Springer, 2008), pp. 311–325.
56. S. Rinderle, M. Reichert and P. Dadam, Disjoint and overlapping process changes: Challenges, solutions, applications, in *Proc. of OTM*, *LNCS 3290*, (Springer, 2004), pp. 101–120.
57. S. Rinderle, M. Reichert and P. Dadam, On dealing with structural conflicts between process type and instance changes, in *Proc. of BPM*, *LNCS 3080*, (Springer, 2004), pp. 274–289.