State Propagation in Abstracted Business Processes

Sergey Smirnov, Armin Zamani Farahani, Mathias Weske

Technische Berichte Nr. 47

des Hasso-Plattner-Instituts für Softwaresystemtechnik an der Universität Potsdam

Technische Berichte des Hasso-Plattner-Instituts für
Softwaresystemtechnik an der Universität Potsdam

Technische Berichte des Hasso-Plattner-Instituts für Softwaresystemtechnik an der Universität Potsdam | 47

Sergey Smirnov | Armin Zamani Farahani | Mathias Weske

State Propagation in Abstracted Business Processes

Universitätsverlag Potsdam

Bibliografische Information der Deutschen Nationalbibliothek
Die Deutsche Nationalbibliothek verzeichnet diese Publikation in der Deutschen Nationalbibliografie; detaillierte bibliografische Daten sind im Internet über http://dnb.de/ abrufbar.

Universitätsverlag Potsdam 2011
http://info.ub.uni-potsdam.de/verlag.htm

Am Neuen Palais 10, 14469 Potsdam
Tel.: +49 (0)331 977 4623 / Fax: 3474
E-Mail: verlag@uni-potsdam.de

Die Schriftenreihe **Technische Berichte des Hasso-Plattner-Instituts für Softwaresystemtechnik an der Universität Potsdam** wird herausgegeben von den Professoren des Hasso-Plattner-Instituts für Softwaresystemtechnik an der Universität Potsdam.

ISSN (print) 1613-5652
ISSN (online) 2191-1665

Online veröffentlicht auf dem Publikationsserver der Universität Potsdam
URL http://pub.ub.uni-potsdam.de/volltexte/2011/5148/
URN urn:nbn:de:kobv:517-opus-51480
http://nbn-resolving.org/urn:nbn:de:kobv:517-opus-51480

Zugleich gedruckt erschienen im Universitätsverlag Potsdam:
ISBN 978-3-86956-130-1

State Propagation in Abstracted Business Processes

Sergey Smirnov, Armin Zamani Farahani, and Mathias Weske

Hasso Plattner Institute, Potsdam, Germany
sergey.smirnov@hpi.uni-potsdam.de,
armin.zamanifarahani@student.hpi.uni-potsdam.de,
mathias.weske@hpi.uni-potsdam.de

Abstract. Business process models are abstractions of concrete operational procedures that occur in the daily business of organizations. To cope with the complexity of these models, business process model abstraction has been introduced recently. Its goal is to derive from a detailed process model several abstract models that provide a high-level understanding of the process. While techniques for constructing abstract models are reported in the literature, little is known about the relationships between process instances and abstract models.
In this paper we show how the state of an abstract activity can be calculated from the states of related, detailed process activities as they happen. The approach uses activity state propagation. With state uniqueness and state transition correctness we introduce formal properties that improve the understanding of state propagation. Algorithms to check these properties are devised. Finally, we use behavioral profiles to identify and classify behavioral inconsistencies in abstract process models that might occur, once activity state propagation is used.

1 Introduction

Recent years have seen increasing interest in modeling business processes to better understand and improve working procedures in organizations and to provide a blue print for process implementation. With increasing complexity of the processes and their IT implementations, process models tend to become complex as well. Often, business users can hardly grasp and analyze the process, due to its complexity. To cope with this problem, business process model abstraction (BPMA) has been introduced. In particular, techniques have been devised to generate from a detailed process model an abstract model that disregards aspects that are considered not important for the abstraction purpose.

While techniques and use cases of process model abstraction are well understood, e.g., [4, 5, 11, 10, 13, 16], little is known about the relationships between process instances and abstract process models. Only a small share of the named BPMA approaches discusses the role of process instances [4, 13]. However, even these research endeavors have gaps and limitations motivating this paper. In this paper we assume that abstract process models are delivered by the abstraction algorithm presented in [16]. The algorithm's input is a process model along with information about activity groups in this model. The output is an abstract model, where each activity corresponds to an activity group in the initial model. Assuming that each activity of the initial model belongs to some group, [16] is liberal in terms of activity group definition, since it enables non-hierarchical activity aggregation. Groups may share activities, while activities of one group can be arbitrary spread over the model.

This paper clarifies the relations between process instances and abstract process models. To achieve this we introduce an approach to derive the state of an activity in the abstract model from the states of concrete process activities, as they happen. The approach is based on activity instance state propagation that determines the state of an abstract activity by the states of their detailed counterparts. We identify two formal properties for state propagation approaches—*state uniqueness* and *state transition correctness*. Further, we develop methods for validation of these properties. We argue that the properties should be considered during the design of any state propagation approach and can be validated by the developed algorithms. Finally, we investigate behavioral inconsistencies that might result from state propagation.

The paper is structured as follows. Section 2 motivates the work and identifies the main challenges. Section 3 introduces the auxiliary formal concepts. In Section 4 we elaborate on the state propagation, its properties and property validation methods, while in section 5 the observed behavioral inconsistencies are explained. We position the contribution of this paper against the related work in Section 6 and conclude with Section 7.

2 Motivating Example and Research Challenges

This section provides further insights into the problem addressed by the current study. We start with a motivating example. Further, we informally outline our approach and identify the main research challenges emerging on the path towards the solution.

Various stakeholders use models with different amount of details about a given business process. In this setting several models are created for one process. Consider the example in Fig. 1. Model m describes a business process, where a forecast request is processed. Once an email with a forecast request is received, a request to collect the required data is sent. The forecast request is registered and the collected data is awaited. Then, there are two options: either to perform a full data analysis, or its quick version. The process concludes with a forecast report creation. Model m contains semantically related activities that are aggregated together into more coarse-grained ones. The groups of related activities are marked by areas with a dashed border, e.g., group g_1 includes *Receive email* and *Record request*. Model m_a is a more abstract specification of the forecast business process. Each activity group in m corresponds to a high-level activity in m_a, e.g., g_1 corresponds to *Receive forecast request*. Meanwhile, m'_a is even more abstract: its activities are refined by the activities of model m_a and are further

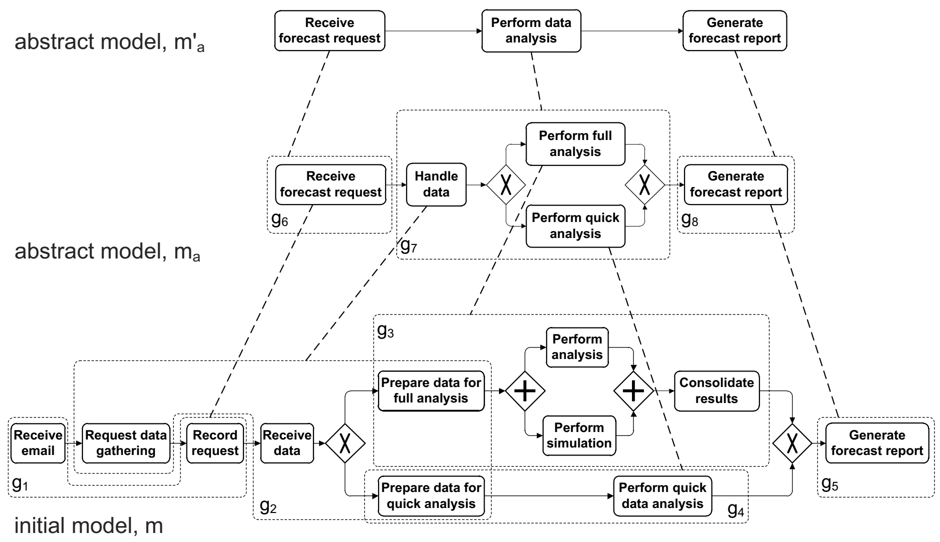


Fig. 1. Models capturing business process "Forecast request handling" at different levels of abstraction

refined by activities of m. While the forecast process can be enacted using model m, abstract models m_a and m'_a are suitable for monitoring the state of process instances. For example, a process participant might leverage model m_a, while the process owner may monitor states of instances by means of m'_a. Traditionally, models capturing the same business process at different levels of abstraction were created independently. However, this method turns out to be pricey and error-prone: keeping these models in sync is a challenge. Hence, in the context of this paper we adhere to the idea that an abstract model m_a is derived from m using an abstraction technique, in particular, the one developed in [16].

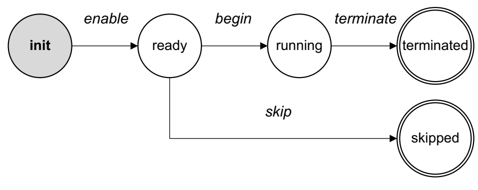


Fig. 2. Activity instance life cycle

We assume that the state of a process instance is defined by the states of its activity instances. The paper adheres the activity instance life cycle presented in Fig. 2. When an activity is created, it is in the *init* state. We consider process models to be acyclic. Hence, once a process is instantiated, all of its activity instances are created in state *init*. The *enable* state transition brings the activity into state *ready*. If an instance is not required, *skip* transition brings it to state *skipped*. The *skipped* state has to be spread among activities that are not required. This can be realized by a well established approach of dead path elimination [12]. From the *ready* state the activity instance may evolve to *running* state by means of transition *begin*. When the instance completes its work, *terminate* transition brings it to the *terminated* state. The use of one activity instance life cycle implies that all activity instances behave according to this life cycle disregard of the abstraction level of the model an activity belongs to. Throughout this paper we frequently refer to *activity instance states*. As *activities* at the model level do not have states, we interchangeably and unambiguously use terms *activity state* and *activity instance state*.

To monitor process instance state by means of an abstract model, while a detailed model is executed, one needs a mechanism establishing the relation between the states of activities in the abstract model and activities of the detailed model. We reference this mechanism as activity instance state propagation. Consider a group of activities g in model m and activity x of the abstract model, such that x is refined by activities of g. State propagation maps the states of instances of activities in g to the state of x. One can design various state propagation mechanisms depending on the use case at hand. However, we identify two formal criteria to be fulfilled by any state propagation. The first criterion, *activity instance state uniqueness*, is motivated by the observation that each activity instance at every point in time is exactly in one state. Hence, this criterion requires state propagation to be a surjective mapping: each constellation of instance states in group g must result exactly one state for x. Second criterion, *activity instance state transition correctness* requires state propagation to assure that every activity instance behaves according to the declared life cycle, neither adding, nor ignoring predefined state transitions. Section 4 not only presents the state propagation, but shows how its properties can be validated.

We design state propagation approach that considers the activity grouping information along with information about the states of activity instances in the groups. This state propagation is simple and can be efficiently implemented. However, this approach does not consider control flow information. Hence, one may observe *behavioral inconsistencies* taking place in the abstract model: while the model control flow prescribes one order of activity execution, state propagation results contradicting activity instance states. Section 5 elaborates on this phenomenon.

3 Preliminaries

This section introduces the basic notions the paper builds on. We start formalizing the concepts of a process model and process instance and link them together. The section postulates behavioral profiles concept and concludes providing insights into the abstraction principles assumed in this paper.

Definition 1 (Process Model). A tuple $m = (A, G, F, s, e, t)$ is a *process model* , where:
- A is a finite nonempty set of activities;
- G is a finite set of gateways;
- $N = A \cup G$ is a set of nodes with $A \cap G = \emptyset$;
- $F \subseteq N \times N$ is a flow relation, such that (N, F) is an acyclic connected graph;
- $\bullet n = \{n' \in N | (n', n) \in F\}$ and $n \bullet = \{n' \in N | (n, n') \in F\}$ denote, respectively, the direct predecessors and successors of a node $n \in N$;

- $\forall\ a \in A : |\bullet a| \leq 1 \wedge |a \bullet| \leq 1$
- $s \in A$ is the only start activity, such that $\bullet s = \emptyset \wedge \forall a \in A \backslash \{s\} : |\bullet a| > 0$;
- $e \in A$ is the only end activity, such that $e \bullet = \emptyset \wedge \forall a \in A \backslash \{e\} : |a \bullet| > 0$;
- $t : G \rightarrow \{and, xor\}$ is a mapping that associates each gateway with a type.

The execution semantics of a process model is given by a translation to a Petri net [1, 8]. Since the defined process model has exactly one start activity and end activity the corresponding Petri net is a WF-net. We consider models that can be mapped to *free-choice* WF-nets [1]. Finally, we assume that a process model is *sound* [2].

To describe the process instance level, we formalize the activity instance life cycle shown in Fig. 2 as a tuple $(\mathcal{S}, \mathcal{T}, tran, \{init\}, \mathcal{S}')$. $\mathcal{S} = \{init, ready, running, terminated, skipped\}$ is a set of activity instance states, where *init* is the initial state and $\mathcal{S}' = \{skipped, terminated\}$ is a set of final states. $\mathcal{T} = \{enable, begin, skip, terminate\}$ is a set of state transition labels. The state transition mapping $tran : \mathcal{S} \times \mathcal{T} \rightarrow \mathcal{S}$, is defined as *tran(init, enable) = ready*, *tran(ready, begin) = running*, *tran(running, terminate) = terminated*, *tran(ready, skip) = skipped*. Against this background, a *process instance* is defined as follows.

Definition 2 (Process Instance). Let $\mathcal{S}$ be the set of activity instance states. A tuple $i = (m, I, inst, stat)$ is a *process instance,* where:

- $m = (A, G, F, s, e, t)$ is a process model;
- I is the set of activity instances;
- $inst : A \rightarrow I$ is a bijective mapping that associates an activity with an activity instance;
- $stat : I \rightarrow \mathcal{S}$ is a mapping establishing the relation between an activity instance and its state.

As Definition 1 claims the process model to be acyclic, there is exactly one activity instance per process model activity, i.e., $|I| = |A|$. To introduce behavioral profiles, see [22], we inspect the set of all traces from s to e for a process model $m = (A, G, F, s, e, t)$. The set of *complete process traces* $\mathcal{T}_m$ for m contains lists of the form $s \cdot A^* \cdot e$, where a list captures the activity execution order. To denote that an activity a is a part of a complete process trace we write $a \in \sigma$ with $\sigma \in \mathcal{T}_m$. Within this set of traces the auxiliary *weak order* relation for activities is defined.

Definition 3 (Weak Order Relation). Let $m = (A, G, F, s, e, t)$ be a process model, and $\mathcal{T}_m$—its set of traces. The *weak order relation* $\succ_m \subseteq (A \times A)$ contains all pairs (x, y), where there is a trace $\sigma = n_1, \ldots, n_l$ in $\mathcal{T}_m$ with $j \in \{1, \ldots, l-1\}$ and $j < k \leq l$ for which holds $n_j = x$ and $n_k = y$.

Two activities of a process model are in weak order, if there exists a trace in which one activity occurs after the other. Depending on how weak order relates two process model activities, we define three relations forming the behavioral profile.

Definition 4 (Behavioral Profile). Let $m = (A, G, F, s, e, t)$ be a process model. A pair $(a, b) \in (A \times A)$ is in one of the following relations:
– *strict order relation* $\rightsquigarrow_m$, *if* $a \succ_m b$ *and* $b \not\succ_m a$;
– *exclusiveness relation* $+_m$, *if* $a \not\succ_m b$ *and* $b \not\succ_m a$;
– *interleaving order relation* $||_m$, *if* $a \succ_m b$ *and* $b \succ_m a$.
The set of all three relations is the *behavioral profile* of m.

The relations of the behavioral profile, along with the inverse strict order $\rightsquigarrow^{-1} = \{(x, y) \in (A \times A) \mid (y, x) \in \rightsquigarrow\}$, partition the Cartesian product of activities in one process model.

The abstraction approach assumed in this paper takes as the input a process model and activity grouping information. The algorithm's output is the abstract process model. We formalize the activity grouping by means of function *aggregate*.

Definition 5 (Function Aggregate). Let $m = (A, G, F, s, e, t)$ be a process model and $m_a = (A_a, G_a, F_a, s_a, e_a, t_a)$—the abstract model derived from m. Function $aggregate : A_a \to (\mathcal{P}(A)\backslash\emptyset)$ sets a correspondence between one activity in m_a and the set of activities in m. We postulate function $st_{agg} : A_a \to (\mathcal{P}(\mathcal{S})\backslash\emptyset)$ defined as $st_{agg}(x) = \bigcup_{\forall a \in aggregate(x)} \{stat(inst(a))\}$, where $x \in A_a$.

The example model in Fig. 1 illustrates function *aggregate* as follows *aggregate(Receive forecast request)* = {*Receive email, Record request*}. In the subsequent examples we denote the coarse-grained activities as x and y, where $x, y \in A_a$, while a and b are such activities of the model m, i.e., $a, b \in A$ that $a \in aggregate(x), b \in aggregate(y)$. The main steps of the employed abstraction approach are as follows.

I derive the behavioral profile BP_m for model m
II construct the behavioral profile BP_{m_a} for model m_a
III **if** a model consistent with profile BP_{m_a} exists
IV **then** create m_a, **else** report an inconsistency.

Within the scope of this paper we are interested in construction of the behavioral profile BP_{m_a} described by Algorithm 1 in [16]. Behavioral profile construction discovers the behavioral profile relations between each pair of activities in m_a. To achieve this we analyze the behavioral profile relations among activities of m and activity groups defined by function *aggregate*. For each pair of coarse-grained activities $x, y \in A_a$, we study the behavioral profile relations of all pairs (a, b), where $(a, b) \in aggregate(x) \times aggregate(y)$. For each group pair we count the number of occurrences of $a \succ_m b$, $b \succ_m a$, $a \not\succ_m b$, and $b \not\succ_m a$. According to Definition 4, each of these relations contributes to the behavioral profile relations. Hence, the evaluation of the weak order relation occurrences along with the prioritization of behavioral profile relations, enables the choice of the ordering constraint for x and y.

4 Activity Instance State Propagation

This section formalizes state propagation. We start designing the state propagation method. Further, Section 4.2 proposes the algorithm validating activity instance state uniqueness, while Section 4.3 elaborates on the algorithm for activity instance state transition correctness validation. The role of the algorithms is twofold. First, they validate the developed state propagation. Second, the algorithms can be reused for validation of other state propagation methods.

4.1 State Propagation

State propagation implies that the state of an activity x in the abstract model m_a is defined by the states of activities $aggregate(x)$ in model m. Consider a process instance example presented in Fig. 3, where *aggregate(Receive forecast request)={Receive email, Record request}*. The instances of *Receive email* and *Record request* define the state of *Receive forecast request* instance. To formalize state propagation we introduce five auxiliary predicates. Each predicate corresponds to one activity instance state and is "responsible" for propagation of this state to an abstract activity. An argument of a predicate is a nonempty set of activity instance states $S \subseteq \mathcal{S}$. Set S is populated by the states of activity instances observed in the activity group $aggregate(x)$, i.e., $S = st_{agg}(x)$. If a predicate evaluates to true, it propagates the respective state to the instance of x. For example, predicate p_{ru} corresponds to the state *running*. Given an instance of *Receive forecast request* and instances of *Receive email* and *Record request*, we evaluate predicate p_{ru} against set {*init, terminated*}. If p_{ru} evaluates to true, we claim that the instance of *Receive forecast request* is in state *running*. Fig. 3 illustrates exactly this constellation. The predicates are defined as follows.

- $p_{in}(S) := \forall s \in S : s = init$
- $p_{re}(S) := (\exists s' \in S : s' = ready \wedge \forall s \in S : s \in \{init, ready, skipped\}) \vee (\exists s', s'' \in S : s' = init \wedge s'' = skipped \wedge \forall s \in S : s \in \{init, skipped\})$
- $p_{ru}(S) := \exists s \in S : s = running \vee (\exists s', s'' \in S : s' = terminated \wedge s'' \in \{init, ready\})$
- $p_{te}(S) := \exists s \in S : s = terminated \wedge \forall s' \in S : s' \in \{skipped, terminated\}$
- $p_{sk}(S) := \forall s \in S : s = skipped$

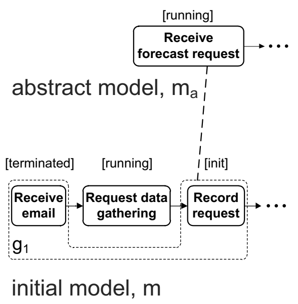


Fig. 3. State propagation example

We reference this set of predicates as *ps*. The predicate design results in exactly one predicate to evaluate to *true*, i.e., activity instance state uniqueness holds. While the predicates p_{in}, p_{te}, and p_{sk} are easy to interpret, p_{re} and p_{ru} require explanation. In particular, the second part of the disjunction in predicate p_{re} propagates state *ready*, if in S exists a *skipped* activity, i.e., this activity *was* in state *ready* and there exists an initialized activity, i.e., that activity *will be* in state *ready*. Similarly, the second part of the disjunction in p_{ru} propagates state *running*, if in S exists a *terminated* activity, i.e., this activity *was* in state *running* and there exists an initialized or *ready* activity, i.e., that activity *can be* in state *running*. These

Algorithm 1 Verification of activity instance state uniqueness

```
1:  checkStateUniqueness(Predicate[] $ps$, LifeCycle $lifeCycle$)
2:  for all $S \subseteq lifeCycle.\mathcal{S}$ do
3:    if $S \neq \emptyset$ then
4:      propagated = false;
5:      for all $p$ in $ps$ do
6:        if !propagated then
7:          if $p(S)$ then
8:            propagated = true;
9:        else
10:         if $p(S)$ then
11:           return false
12:     if !propagated then
13:       return false
14: return true
```

additional conditions assure that a high-level activity behaves according to the life cycle, i.e., correctness of activity instance state transition holds. The five predicates construct *activity instance state propagation function stp* which is used to define the state of activity x instance.

Definition 6 (Activity Instance State Propagation Function). *Activity instance state propagation function stp* : $\mathcal{P}(\mathcal{S}) \to \mathcal{S}$ maps a set of activity instance states to one activity instance state:

$$stp(S) = \begin{cases} init, & if\, p_{in}(S) \\ ready, & if\, p_{re}(S) \\ running, & if\, p_{ru}(S) \\ terminated, & if\, p_{te}(S) \\ skipped, & if\, p_{sk}(S). \end{cases}$$

Let $m = (A, G, F, s, e, t)$ be a process model with its process instance $i = (m, I, inst, stat)$ and $m_a = (A_a, G_a, F_a, s_a, e_a, t_a)$—the abstract model with process instance $i_a = (m_a, I_a, inst_a, stat_a)$. Then, function $stat_a : I_a \to \mathcal{S}$ is defined as $stat_a(inst_a(x)) = stp(st_{agg}(x))$.

4.2 Activity Instance State Uniqueness

State propagation mechanism maps the states of activity instances of $aggregate(x)$ to the state of $inst(x)$. Activity instance state uniqueness requires exactly one predicate of ps to evaluate to $true$ for any combination of states in activity group $aggregate(x)$. However, an activity group is defined by the user and is not known in advance. Hence, it is not efficient to reason about state uniqueness property explicitly enumerating all activity instance states that occur within activity instance groups. Instead of dealing with concrete activity instance groups, we introduce activity instance group equivalence classes. For a process instance

Algorithm 2 Validation of activity instance state transition correctness

```
 1: checkStateTransitionCorrectness(Predicate[] $ps$, LifeCycle $lifeCycle$)
 2: for all $p$ in $ps$ do
 3:    for all $S \subseteq \mathcal{S} : p(S) = true$ do
 4:       for all $s \in (S \backslash lifeCycle.\mathcal{S}')$ do
 5:          for all $t \in lifeCycle.\mathcal{T}$, where $tran(s,t)$ is defined do
 6:             $S' = S \cup \{tran(s,t)\}$
 7:             if $stp(S') \neq stp(S)$ and $tran(stp(S),t) \neq stp(S')$ then
 8:                return false
 9:             $S' = S' \backslash \{s\}$
10:             if $stp(S') \neq stp(S)$ and $tran(stp(S),t) \neq stp(S')$ then
11:                return false
12: return true
```

$i = (m, I, inst, st)$ two activity instance groups $I_1, I_2 \subseteq I$ belong to one equivalence class, if in both groups the same set of activity instance states is observed, i.e., $\forall i_1 \in I_1 \exists i_2 \in I_2 : stat(i_1) = stat(i_2) \wedge \forall i_2 \in I_2 \exists i_1 \in I_1 : stat(i_2) = stat(i_1)$. For instance, a pair of activity instances with states (*init*, *terminated*) and an activity triple with states (*init*, *init*, *terminated*) belong to one class with observed states $S = \{$*init*, *terminated*$\}$. Notice that this classification covers all possible state combinations, thereby the algorithm checks all cases. We can consider such classes of activity instance groups, since the predicates make use of existential and universal quantifiers. Each equivalence class is represented by the state set $S \subseteq \mathcal{S}$—the state set observed in the members of this class. If for every equivalence class representative exactly one predicate evaluates to *true*, we claim that activity instance state uniqueness holds for the validated state propagation mechanism.

Algorithm 1 validates activity instance state uniqueness. The algorithm takes a set of predicates and an activity instance life cycle as inputs; it returns *true*, once the property holds. As the number of equivalence classes is exponential to the number of states in the activity instance life cycle, the computational complexity of Algorithm 1 is also exponential. However, in practice the number of activity instance states is typically low.

4.3 Activity Instance State Transition Correctness

Each activity instance must behave according to the assumed life cycle: only the state transitions allowed by the life cycle are permitted. To validate activity instance state transition correctness we propose Algorithm 2. The algorithm takes a set of predicates and an activity instance life cycle as inputs; it returns *true*, if state transitions are correct. The key idea of the algorithm is the observation that an instance of activity x in the abstract model changes its state, when *one* of the activity instances that refines x changes its state. Due to this observation, the validation considers all possible state transitions. Hence, for each predicate p specifying a state propagation rule the algorithm constructs state sets $S \subseteq \mathcal{S}$,

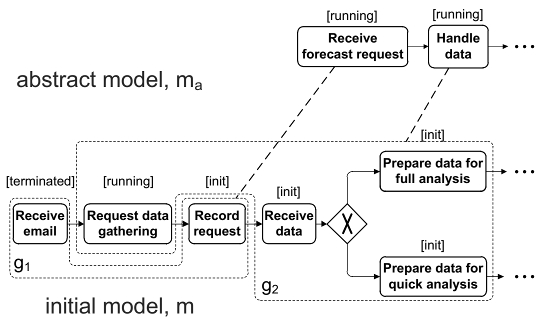


Fig. 4. Behavioral inconsistency in a process instance for the business process in Fig. 1

where the predicate evaluates to *true* (lines 2–3). For instance, predicate p_{in} has one such set $\{init\}$. In the next step, the validation constructs state set S' reachable from S by one state transition of the activity instance life cycle (lines 4–6 and line 9). In the example state set $S = \{init\}$ evolves to sets $\{ready\}$ and $\{init,\ ready\}$. For each of those reachable state sets S' function *stp* is evaluated. If for each S' the state $stp(S')$ equals $stp(S)$ or can be reached from $stp(S)$ using the same state transition as required to reach S' from S, the state propagation rules are valid. Algorithm 2 realizes the checks in lines 7 and 10 and reports correctness in line 12.

5 Behavioral Inconsistencies

This section elaborates on the problem of behavioral inconsistencies. We start with the motivation, then introduce auxiliary formal concepts and define the notion of behavioral inconsistency. Finally, we present a classification of behavioral inconsistencies.

5.1 Example

An abstract process model dictates activity execution order. At the same time, the designed state propagation mechanism disregards the control flow, but influences the states of activities in m_a. In this setting one can observe *behavioral inconsistencies.* Fig. 4 exemplifies the behavioral inconsistency. Activities *Receive forecast request* and *Handle data* are refined with activity groups g_1 and g_2, respectively. According to the state propagation mechanism, once *Receive email* terminates, *Receive forecast request* is in state *running* until *Record request* terminates. While *Request data gathering* runs, *Handle data* is in the state *running.* Hence, according to state propagation we observe activities *Receive forecast request* and *Handle data* in state *running* at the same time. However, model m_a prescribes sequential execution of these activities: *Handle data* can be executed, once *Receive forecast request* terminates. Hence, these states are inconsistent with the control flow of model m_a.

Behavioral inconsistencies have two causes, both stemming from the properties of the abstraction approach. The first cause is activity grouping. Consider the example in Fig. 4, where activities in groups g_1 and g_2 interleave: *Receive email* $\rightsquigarrow_m$ *Request data gathering* and *Request data gathering* $\rightsquigarrow_m$ *Record request*. The second reason is that the assumed abstraction approach preserves neither activity *optionality*, nor *causality*. We say that an activity is optional, if there is such a process trace, where this activity is not observed. Considering the example in Fig. 4 *Prepare data for full analysis* is optional. Activity causality implies that 1) an order of execution for two activities is given and 2) two activities appear together in all process executions. One can observe causality relation for *Receive email* and *Receive data*, but not for *Receive email* and *Prepare data for full analysis*. Next section formalizes the notion of behavioral inconsistencies.

5.2 Formalization of Behavioral Inconsistencies

While behavioral profiles enable judgment on activity ordering, they do not capture causality. Following on [23] we make use of auxiliary *co-occurrence relation* and *causal behavioral profile*.

Definition 7 (Causal Behavioral Profile). Let $m = (A, G, F, s, e, t)$ be a process model and $\mathcal{T}_m$ be its set of traces. A pair $(a, b) \in (A \times A)$ is in the *co-occurrence relation* $\gg_m$ iff for all traces $\sigma = n_1, \ldots, n_l$ in $\mathcal{T}_m$ it holds $n_i = a, i \in \{1, \ldots, l\}$ implies that $\exists j \in \{1, \ldots, l\}$ such that $n_j = b$. Then $\{\rightsquigarrow_m, ||_m, +_m, \gg_m\}$ is the *causal behavioral profile of* m.

The causality relation holds for $a, b \in A$ if $a \rightsquigarrow_m b$ and $a \gg_m b$.

The example in Fig. 4 witnesses that state propagation allows concurrent activity execution. However, the behavioral profile relations are defined on the trace level and do not capture concurrency. To formalize the *observed* behavior of activities, we introduce relations defined on the process instance level. These relations build on top of causal behavioral profile relations. However, they consider not traces, but process instances.

We say $(x, y) \in \rightsquigarrow_{obs}$ if there is a process instance where x is executed before y, but no instance, where y is executed before x. Similarly, relation $x \rightsquigarrow_{obs}^{-1} y$ means that there is a process instance where y is executed before x, but no instance, where x is executed before y. Relation $x +_{obs} y$ holds if there is no instance where x and y both take place. Relation $||_{obs}$ corresponds to the existence of 1) an instances where x is executed before y, 2) an instance where y is executed before x and 3) an instance where x and y are executed concurrently. Finally, $x \gg_{obs} y$ holds if for every instance, where x is executed, y is executed as well. Then, the behavioral inconsistency can be defined as follows.

Definition 8 (Behavioral Inconsistency). Let $m = (A, G, F, s, e, t)$ be a process model and $i = (m, I, inst, stat)$—its instance. $m_a = (A_a, G_a, F_a, s_a, e_a, t_a)$ is the abstract model obtained from m and having the instance $i_a = (m_a, I_a, inst_a, stat_a)$, where function $stat_a$ is defined as $stat_a(inst_a(x)) =$

$stp(st_{agg}(x))$. We say that there is a *behavioral inconsistency,* if for activities $(x, y) \in (A_a \times A_a)$ the causal behavioral profile relations do not coincide with the observed behavioral relations:

– $(x, y) \in \rightsquigarrow_{m_a}$ and $(x, y) \notin \rightsquigarrow_{obs}$;
– $(x, y) \in \rightsquigarrow^{-1}_{m_a}$ and $(x, y) \notin \rightsquigarrow^{-1}_{obs}$;
– $(x, y) \in +_{m_a}$ and $(x, y) \notin +_{obs}$;
– $(x, y) \in ||_{m_a}$ and $(x, y) \notin ||_{obs}$;
– $(x, y) \in \gg_{m_a}$ and $(x, y) \notin \gg_{obs}$;

5.3 Classification of Behavioral Inconsistencies

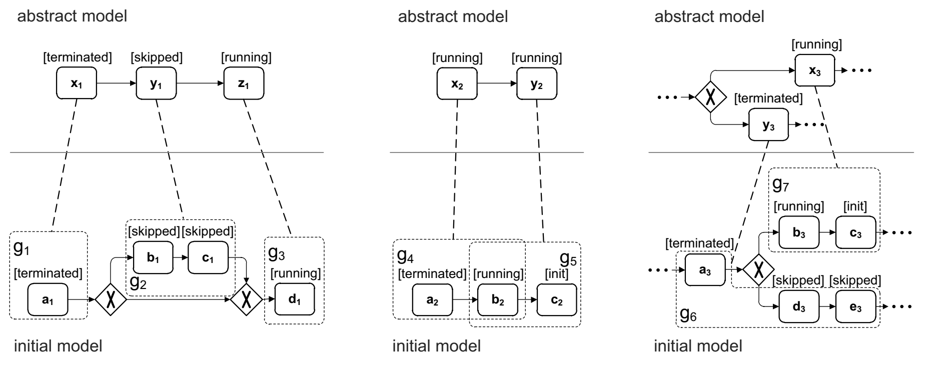
Table 1 classifies behavioral inconsistencies comparing the declared and observed behavioral constraints for abstract process model activities x and y. A table row corresponds to behavioral profile relations declared by an abstract model. Columns capture the observed behavioral relations. A cell of Table 1 describes an inconsistency between the observed and declared behavioral relations.

The "+" sign witnesses no inconsistency since the declared and observed constraints coincide. We identify one class of activity groups that cause no inconsistency. Consider a pair of activities $x, y \in A_a$. If $\forall (a, b) \in aggregate(x) \times aggregate(y)$ the same causal behavioral profile relation holds, no behavioral inconsistency is observed. Indeed, in such a setting the employed abstraction algorithm results in the same behavioral profile relation for (x, y) as the one observed for (a, b). Hence, the observed behavioral constraint coincides with the behavioral constraint imposed by the model. A prominent example of activity groups that fulfill the defined requirement are groups resulting from the canonical decomposition of a process model into single entry single exit fragments, see [18, 19].

Cells marked with "–" represent inconsistencies that do not occur, given the considered abstraction method. We organize these inconsistencies in three clusters. The first cluster includes cases where $x \rightsquigarrow_{m_a} y$ and $x \rightsquigarrow^{-1}_{obs} y$. Algorithm 1 in [16], delivers $x \rightsquigarrow_{m_a} y$ only if exist $a \in aggregate(x)$ and $b \in aggregate(y)$ such that $a \succ_m b$. Hence, it holds either $x \rightsquigarrow_{obs} y$ or $x||_{obs}y$, but not $x \rightsquigarrow^{-1}_{obs} y$. Similar argumentation enables reasoning about the second cluster: a combination of

		$x \rightsquigarrow_{obs} y$		$x \rightsquigarrow^{-1}_{obs} y$		$x +_{obs} y$	$x\|\|_{obs}y$
		$x \gg_{obs} y$	$x \not\gg_{obs} y$	$x \gg_{obs} y$	$x \not\gg_{obs} y$		
$x \rightsquigarrow_{m_a} y$	$x \gg_{m_a} y$	+	A	–	–	–	B
	$x \not\gg_{m_a} y$	±	+	–	–	–	B
$x \rightsquigarrow^{-1}_{m_a} y$	$x \gg_{m_a} y$	–	–	+	A	–	B
	$x \not\gg_{m_a} y$	–	–	±	+	–	B
$x +_{m_a} y$		C	C	C	C	+	C
$x\|\|_{m_a}y$		±	±	±	±	–	+

Table 1. Classification of behavioral inconsistencies



(a) Co-occurrence loss (b) Order loss (c) Exclusiveness loss

Fig. 5. Examples of behavioral inconsistencies: one example per class

$x \rightsquigarrow_{m_a}^{-1} y$ and $x \rightsquigarrow_{obs} y$. Finally, we argue that $x +_{obs} y$ is not possible, once $x \rightsquigarrow_{m_a} y$, $x \rightsquigarrow_{m_a}^{-1} y$, or $x||_{m_a} y$. The abstraction algorithm results in $x \rightsquigarrow_{m_a} y$ only if $a \succ_m b$, where $a \in \mathit{aggregate}(x), b \in \mathit{aggregate}(y)$. This contradicts observation of $x +_{obs} y$. By analogy one can show the contradiction for $x \rightsquigarrow_{m_a}^{-1} y$ and $x +_{obs} y$ along with $x||_{m_a} y$ and $x +_{obs} y$.

Six table cells are marked with "$\pm$" symbol. Every cell corresponds to an inconsistency, where no contradiction takes place: an observed relation restricts a declared behavioral relation. Consider, for instance, the behavioral inconsistency, where $x||_{m_a} y$, while $x \rightsquigarrow_{obs}^{-1} y$ and $x \gg_{obs} y$. This inconsistency has no contradiction, since the observed behavior only restricts the declared one.

We identify three classes of behavioral inconsistencies marked in Table 1 and illustrate them by the examples in Fig. 5.

`A:` `Co-occurrence loss` Behavioral inconsistencies of this type take place when the model declares co-occurrence for an activity pair, while both activities are observed only in some process instances. The cause of inconsistency is the abstraction approach loosing information about the causal coupling of an activity pair. Notice, that abstraction preserves the behavioral profile relation, i.e., (inverse) strict order. The example in Fig. 5(a) illustrates this inconsistency type. Since activities of group g_2 are skipped, activity y_1 is in state *skipped* as well. However, it can not be skipped according to the control flow of the abstract model.

`B:` `Order loss` For a pair of activities in (inverse) strict order, the user observes interleaving execution. A behavioral inconsistency of this type is exemplified in Fig. 5(b). Such inconsistencies have the following roots: 1) $\mathit{aggregate}(x) \cap \mathit{aggregate}(y) \neq \emptyset$ or 2) exist $a_1, a_2 \in \mathit{aggregate}(x)$ and $b_1, b_2 \in \mathit{aggregate}(y)$ such that it holds $a_1 \succ_m b_1$ and $b_2 \succ_m a_2$. In Fig. 5(b) activity b_2 belongs to groups g_1 and g_2. As a consequence, once b_2 runs both sequential activities x_2 and y_2 are running concurrently.

`C:` `Exclusiveness loss` While the model prescribes exclusiveness relation for x and y, both activities are observed within one instance. These inconsistencies take place, once the abstraction algorithm ignores $a \succ_m b$ or $b \succ_m a$ in the initial model, as relations $a \not\succ_m b$ and $b \not\succ_m a$ dominate. Fig. 5(c) exemplifies this inconsistency. Given the constellation of activity groups in the initial model, activities x_3 and y_3 are exclusive. However, in the presented process instance both x_3 and y_3 are executed.

6 Related Work

We identify two directions of the related work. The first one is the research on business process model abstraction. The second one is the body of knowledge discussing similarity of process models.

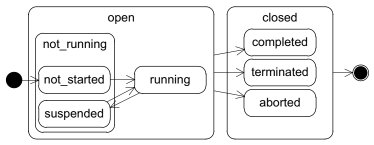


Fig. 6. Activity instance life cycle as presented in [13]

The problem of business process model abstraction has been approached by several authors. The majority of the solutions consider various aspects of model transformation. For instance, [5, 10, 11, 14, 16] focus on the structural aspects of transformation. Among these papers [16] enables the most flexible activity grouping. Several papers study how the groups of semantically related activities can be discovered [6, 15]. A few works elaborate on the relation between process instances and abstract process models, e.g. [4, 13]. In [4] Bobrik, Reichert, and Bauer discuss state propagation and associated behavioral inconsistencies, but do not use the concept of activity instance life cycle. [13] suggests state propagation approach that builds on the activity instance life cycle shown in Fig. 6. In [13] Liu and Shen order 3 states according to how "active" they are: *not_started* < *suspended* < *running*. The state propagation rules make use of this order, e.g., if a coarse-grained activity is refined by activities in one of the *open* states, the high-level activity is in the most "active" state. Against this background, consider an example activity pair evolving as follows: (*not_started*, *not_started*) to (*not_started*, *running*) to (*not_started*, *completed*). According to the rules defined in [13] the high level activity evolves as *not_started* to *running* to *not_started*, which contradicts the predefined activity instance life cycle. As we mentioned above, the majority of works on business process model abstraction consider only the *model* level. Meanwhile, the papers that take into account process instances have gaps and limitations. For instance, [4, 13] motivated us not only to introduce the state propagation approach, but also to identify formal properties for such approaches and develop algorithms for their validation.

The works on similarity of process models can be refined into two substreams. A series of papers approaches process model similarity analyzing model structure and labeling information, see [7, 20]. These works provide methods to discover matching model elements. Several research endeavors analyze behavioral similarity of process models. In particular, [3] introduces several notions of inheritance and operations on process models preserving the inheritance property. Recently, Weidlich, Dijkman, and Weske investigated behavioral compatibility of models capturing one business process [21]. [9] elaborates on process model similarity considering both model element labeling and model behavior. Considering that processes are inherently concurrent systems, various notions of behavioral equivalence for concurrent systems can be leveraged to compare the behavior of initial and abstract process models [17]. The enumerated papers help to compare the behavior of initial and abstract process models. As such, the notions of behavioral equivalence and behavioral compatibility might give additional insights into the causes of behavioral inconsistencies, see Section 5, and help to classify them further.

7 Conclusion and Future Work

Although business process model abstraction has been thoroughly studied earlier, the relations between process instances and abstract process models have been barely explored. The current paper bridged this gap. First, we developed activity instance state propagation mechanism that allows to describe the process instance state by means of an abstract process model. Second, we have identified two formal properties for state propagation and proposed methods for their validation. Finally, we elaborated on behavioral inconsistencies that can be observed, once the assumed abstraction and state propagation mechanisms are used.

We foresee several directions of the future work. The direct next step is the extension of the considered model class. As we leverage dead path elimination to spread activity instance state *skipped* over not executed activities, the state propagation approach is limited to acyclic models. Substitution of dead path elimination with an alternative approach would facilitate handling of cyclic models. Another direction is the further study of the behavioral inconsistencies and methods for their resolution. With that respect, it is valuable to integrate control flow information into state propagation mechanism. Finally, the applications of the introduced technique call for in deep investigation. One direct application of our approach is business process monitoring [24], where abstract models help users to follow the progress of running business processes.

References

1. W. M. P. van der Aalst. The Application of Petri Nets to Workflow Management. *JCSC*, 8(1):21–66, 1998.
2. W. M. P. van der Aalst. Workflow Verification: Finding Control-Flow Errors Using Petri-Net-Based Techniques. In *BPM*, volume 1806 of *LNCS*, pages 161–183, 2000.
3. W. M. P. van der Aalst and T. Basten. Life-Cycle Inheritance: A Petri-Net-Based Approach. In *ICATPN*, volume 1248 of *LNCS*, pages 62–81. Springer, 1997.
4. R. Bobrik, M. Reichert, and T. Bauer. Parameterizable Views for Process Visualization. Technical Report TR-CTIT-07-37, Centre for Telematics and Information Technology, University of Twente, Enschede, April 2007.
5. R. Bobrik, M. Reichert, and T. Bauer. View-Based Process Visualization. In *BPM*, volume 4714 of *LNCS*, pages 88–95, Berlin, 2007. Springer.

6. C. Di Francescomarino, A. Marchetto, and P. Tonella. Cluster-based Modularization of Processes Recovered from Web Applications. *Journal of Software Maintenance and Evolution: Research and Practice*, 2010.
7. R. M. Dijkman, M. Dumas, and L. García-Bañuelos. Graph Matching Algorithms for Business Process Model Similarity Search. In *BPM*, volume 5701 of *LNCS*, pages 48–63. Springer, 2009.
8. R. M. Dijkman, M. Dumas, and Ch. Ouyang. Semantics and Analysis of Business Process Models in BPMN. *IST*, 50(12):1281–1294, 2008.
9. B. van Dongen, R. M. Dijkman, and J. Mendling. Measuring Similarity between Business Process Models. In *CAiSE*, pages 450–464. Springer, 2008.
10. R. Eshuis and P. Grefen. Constructing Customized Process Views. *DKE*, 64(2):419–438, 2008.
11. C. W. Günther and W. M. P. van der Aalst. Fuzzy Mining—Adaptive Process Simplification Based on Multi-perspective Metrics. In *BPM*, volume 4714 of *LNCS*, pages 328–343, Berlin, 2007. Springer.
12. F. Leymann and D. Roller. *Production Workflow: Concepts and Techniques*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2000.
13. Duen-Ren Liu and Minxin Shen. Business-to-business Workflow Interoperation based on Process-Views. *Decis. Support Syst.*, 38:399–419, December 2004.
14. A. Polyvyanyy, S. Smirnov, and M. Weske. The Triconnected Abstraction of Process Models. In *BPM*, LNCS, pages 229–244. Springer, 2009.
15. S. Smirnov, R. M. Dijkman, J. Mendling, and M. Weske. Meronymy-Based Aggregation of Activities in Business Process Models. In *ER*, volume 6412 of *LNCS*, pages 1–14. Springer, 2010.
16. S. Smirnov, M. Weidlich, and J. Mendling. Business Process Model Abstraction Based on Behavioral Profiles. In *ICSOC*, volume 6470 of *LNCS*, pages 1–16, 2010.
17. R. J. van Glabbeek. The Linear Time-Branching Time Spectrum (Extended Abstract). In *CONCUR*, volume 458 of *LNCS*, pages 278–297. Springer, 1990.
18. J. Vanhatalo, H. Völzer, and J. Koehler. The Refined Process Structure Tree. In *BPM*, volume 5240 of *LNCS*, pages 100–115. Springer, 2008.
19. J. Vanhatalo, H. Völzer, and F. Leymann. Faster and More Focused Control-Flow Analysis for Business Process Models Through SESE Decomposition. In *ICSOC*, volume 4749 of *LNCS*, pages 43–55. Springer, 2007.
20. M. Weidlich, R. M. Dijkman, and J. Mendling. The ICoP Framework: Identification of Correspondences between Process Models. In *CAiSE*, volume 6051 of *LNCS*, pages 483–498. Springer, 2010.
21. M. Weidlich, R. M. Dijkman, and M. Weske. Deciding Behaviour Compatibility of Complex Correspondences between Process Models. In *BPM*, volume 6336 of *LNCS*, pages 78–94. Springer, 2010.
22. M. Weidlich, J. Mendling, and M. Weske. Efficient Consistency Measurement based on Behavioural Profiles of Process Models. *IEEE TSE*, 2010. to appear.
23. M. Weidlich, A. Polyvyanyy, J. Mendling, and M. Weske. Efficient Computation of Causal Behavioural Profiles Using Structural Decomposition. In *Petri Nets*, volume 6128 of *LNCS*, pages 63–83. Springer, 2010.
24. M. zur Muehlen. *Workflow-based Process Controlling - Foundation, Design and Application of workflow-driven Process Information Systems.* PhD thesis, University of Münster, 2002.

ISBN 978-3-86956-130-1
ISSN 1613-5652