Weak Conformance between Process Models and Synchronized Object Life Cycles

Andreas Meyer, Mathias Weske

Technische Berichte Nr. 91

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 | 91

Andreas Meyer | Mathias Weske

Weak Conformance between Process Models and Synchronized Object Life Cycles

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.dnb.de/ abrufbar.

Universitätsverlag Potsdam 2015
http://verlag.ub.uni-potsdam.de/

Am Neuen Palais 10, 14469 Potsdam
Tel.: +49 (0)331 977 2533 / Fax: 2292
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

Das Manuskript ist urheberrechtlich geschützt.

Online veröffentlicht auf dem Publikationsserver der Universität Potsdam
URN urn:nbn:de:kobv:517-opus-71722
http://nbn-resolving.de/urn:nbn:de:kobv:517-opus-71722

Zugleich gedruckt erschienen im Universitätsverlag Potsdam:
ISBN 978-3-86956-303-9

Process models specify behavioral execution constraints between activities as well as between activities and data objects. A data object is characterized by its states and state transitions represented as object life cycle. For process execution, all behavioral execution constraints must be correct. Correctness can be verified via soundness checking which currently only considers control flow information. For data correctness, conformance between a process model and its object life cycles is checked. Current approaches abstract from dependencies between multiple data objects and require fully specified process models although, in real-world process repositories, often underspecified models are found. Coping with these issues, we introduce the concept of synchronized object life cycles and we define a mapping of data constraints of a process model to Petri nets extending an existing mapping. Further, we apply the notion of weak conformance to process models to tell whether each time an activity needs to access a data object in a particular state, it is guaranteed that the data object is in or can reach the expected state. Then, we introduce an algorithm for an integrated verification of control flow correctness and weak data conformance using soundness checking.

Contents

1 Introduction 9

2 Synchronized Object Life Cycles 11

3 Mapping a Process Model with Data Constraints to a Petri net 15

4 Weak Conformance 18

5 Related Work 26

6 Conclusion 28

1 Introduction

Business process management allows organizations to specify their processes structurally by means of process models, which are then used for process execution. Process models comprise multiple perspectives with two of them receiving the most attention in recent years: control flow and data [32]. These describe behavioral execution constraints between activities as well as between activities and data objects. It is usually accepted that control flow drives execution of a process model. While checking control flow correctness using soundness [28] is an accepted method, correctness regarding data and control flow is not addressed in sufficient detail. In this paper, we describe a formalism to integrate control flow and data perspectives that is used to check for correctness.

In order to achieve safe execution of a process model, it must be ensured that every time an activity attempts to access a data object, the data object is in a certain expected data state or is able to reach the expected data state from the current one, i.e. data specification within a process model must conform to relevant object life cycles, where each describes the allowed behavior of a distinct class of data objects. Otherwise, the execution of a process model may deadlock. To check for deadlock-free execution in terms of data constraints, the notion of object life cycle conformance [13, 26] is used. This approach has some restrictions with respect to data constraint specification, because each single change of a data object as specified in the object life cycle, we refer to as data state transition, must be performed by some activity. [31] relaxes this limitation such that several state changes can be subsumed within one activity. However, gaps within the data constraints specification, i.e. implicit data state transitions, are not allowed although some other process may be responsible of performing a state change of an object, i.e. these approaches can only check whether an object is in a certain expected state. We assume that implicit data state transitions get realized by an external entity or by detailed implementations of process model activities. In real world process repositories, usually many of those *underspecified* process models exist, which motivates the introduction of the notion of weak conformance [18]. It allows to also check underspecified models.

Additionally, in real world, often dependencies between multiple data objects exist; e.g. an order may only be shipped to the customer after the payment is recognized. Non of above approaches supports this. Thus, we introduce the concept

of synchronized object life cycles that allows to specify dependencies between data states as well as state transitions of different life cycles. Based thereon, we extend the notion of weak conformance and describe how to compute it for a given process model and the corresponding object life cycles including synchronizations. We utilize the well established method of soundness checking [28] to check for process model correctness. In this light, we extend an existing control flow mapping [5] with capabilities to map the data constraints to a Petri net as well to enable an integrated checking of control flow and data correctness.

The remainder is structured as follows. Section 2 introduces the concept of synchronized object life cycles. Subsequently, we describe the mapping of data flow to a Petri net in Section 3. Utilizing this information, we introduce the extended notion of weak conformance and the procedure for integrated correctness checking in Section 4. Section 5 is devoted to related work before we conclude the paper in Section 6.

2 Synchronized Object Life Cycles

First, we give a generic process model definition and require the process model to be (i) syntactically correct with respect to the used modeling notation and to be (ii) structurally sound, i.e. a process model has exactly one start node and one end node and each further node is on a path from the start to the end node. Behaviorally, we require that the process model terminates for all execution paths and that every node participates in at least one execution path, i.e. the process model must be lifelock and deadlock free.

Definition 1 (Process Model)**.**
A *process model* $m = (N, D, Q, \mathfrak{C}, \mathfrak{F}, type, \mu, \varphi)$ consists of a finite non-empty set $N \subseteq A \cup E \cup G$ of control flow nodes being activities A, events E, and gateways G (A, E, and G are pairwise disjoint), a finite non-empty set D of data nodes, and a finite non-empty set Q of activity labels (N, D, and Q are pairwise disjoint). $\mathfrak{C} \subseteq N \times N$ is the control flow relation specifying the partial ordering of activities and $\mathfrak{F} \subseteq (A \times D) \cup (D \times A)$ is the data flow relation specifying input and output data constraints of activities. Function $type : G \rightarrow \{AND, XOR\}$ gives each gateway a type and function $\mu : A \rightarrow Q$ assigns to each activity a label. Function $\varphi : G \times (A \cup G) \nrightarrow D$ assigns to each control flow edge originating from a gateway of type XOR one condition indicating when to follow that edge. ◇

We assume that events represent start and end nodes only and that the assigned data conditions are non-blocking. For visualization of process models, we use BPMN [22], but the concepts described in this paper can be generically applied to other process description languages as well that follow above definition. Figure 2.1 shows a simple order delivery and payment process model with one start event, one end event, 8 activities, 4 gateways, and multiple data nodes. Each data node has a name, e.g. *Order*, and a specific data state shown in brackets, e.g. *confirmed* or *shipped*. Each data node modeled in the process model refers to one data class of the same name; several nodes with the same name reference the same class. A data class describes the structure of data nodes used in the process model including information about states to be assigned to a data node. A data state denotes a situation of interest for the execution of the business process. During process execution, each data node

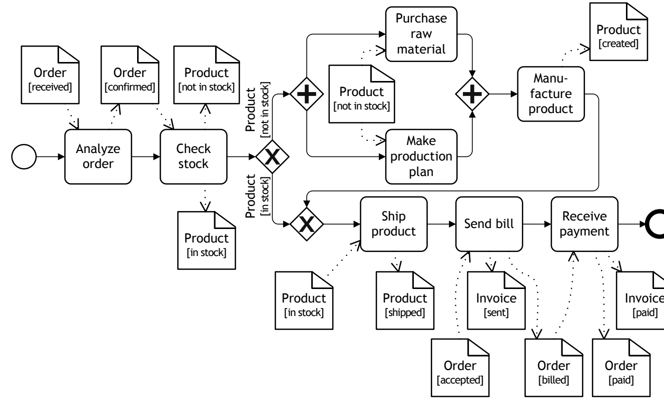

Figure 2.1: A simple *order delivery and payment* process model in BPMN notation

specified on model-level refers to one *data object* at run-time, where each of them is an instance of the corresponding data class. The relations between all states of a data class are represented within an object life cycle.

Definition 2 (Object Life Cycle)**.**
An *object life cycle* $l = (S, s_i, S_F, T, \Sigma, c)$ is a finite state machine and consists of a finite non-empty set S of data states, an initial data state $s_i \in S$, a non-empty set $S_F \subseteq S$ of final data states, and a finite set Σ of actions representing the manipulations on data objects resulting in state changes (S and Σ are disjoint). $T \subseteq S \times \Sigma \times (S \backslash \{s_i\})$ is the data state transition relation through which an object life cycle describes the relations between the data states of a data class. ◇

L denotes the set of all object life cycles for data classes utilized in the process model. Figure 2.2 (solid lines) shows the object life cycles for data classes *Order*, *Product*, and *Invoice* indicating, for instance, that state *confirmed* of class *Order* must precede states *shipped* and *paid*. A manipulation performed on one data object often does not only rely on the current data state of this data object but on the data states of further data objects as well. To handle these inter object life cycle dependencies, we introduce the concept of *object life cycle synchronization* (dotted edges). In this context, we define active data states – a prerequisite – as follows.

Definition 3 (Active Data State)**.**
A data state in an object life cycle is *active* at a specific point in time, if it is one of the data states the corresponding data object may currently be in at this specific point in time. ◇

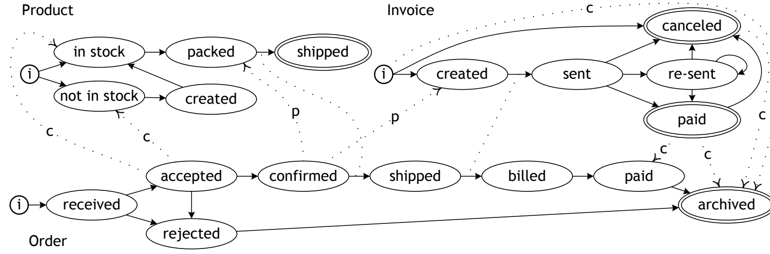

Figure 2.2: Object life cycle synchronization of *Order*, *Product*, and *Invoice* data classes

Deciding whether a certain data state is active requires information about manipulations done to that data object. This information can be derived, for instance, from process models. All data states on a path from the one accessed last to the one accessed next in the corresponding object life cycle including these two states are considered active. If an activity reads (writes) multiple data nodes of the same data class, all of them are considered accessed last respectively accessed next. In case, the activity representing the current point in time only reads (writes) nodes of the specific data class, activities succeeding (preceding) this activity are inspected until a match is found. If there exists no last accessed data state, the initial one is considered last accessed. If there exists no next accessed data state, all reachable final data states are considered next accessed. For instance, upon execution of activity *Manufacture product* in Figure 2.1, states *confirmed* and *shipped* of class *Order* are active. Next, we proceed with the concept of object life cycle synchronization, where synchronization is achieved by synchronization edges.

Definition 4 (Synchronization Edge)**.**
A *synchronization edge* $se = (src, tgt, dep)$ consists of a source src, a target tgt, and an optional dependency type dep to connect multiple object life cycles synchronizing the data state transitions between them. Thereby, it either connects two data states or two data state transitions of two object life cycles defining preconditions towards data state transitions or ensuring joint execution of the connected transitions respectively. An undirected synchronization edge $se_T = (t_1, t_2)$ connects data state transitions $t_1 \in T_{S,l_1}$ and $t_2 \in T_{S,l_2}$ $(l_1 \neq l_2)$. The third attribute, dep, is not used, i.e. an edge se_T is untyped. A directed synchronization edge $se_S = (s_1, s_2, dep)$ connects data states $s_1 \in S_{l_1}$ and $s_2 \in S_{l_2}$ $(l_1 \neq l_2)$ with s_1 being the source data state, s_2 being the target data state, and $dep = \{currently, previously\}$ describing the type of dependency between these data states, i.e. an edge se_S is typed. ◇

For synchronization edges connecting data states, *currently* means that the source state must be active in the corresponding object life cycle if a transition to the target state shall occur in another object life cycle. *Previously* relaxes this requirement such that the source data state must have been active some time in the past to allow the data state transition to the target data state. Two data state transitions connected by a synchronization edge get combined such that they are executed together. This property is transitive. Referring to a process model, this means that one activity changes the state of multiple data nodes respectively objects. SE denotes the finite set of all synchronization edges.

Putting the concepts together, we define a synchronized object life cycle as follows.

Definition 5 (Synchronized Object Life Cycle)**.**
A *synchronized object life cycle* $\mathcal{L} = (L, SE)$ consists of a finite non-empty set L of object life cycles and a finite set SE of synchronization edges connecting various object life cycles. ◇

Visualization of synchronization edges is achieved by dotted directed edges between the data states stated in a tuple and a label with respect to the type of dependency or undirected edges between the data state transitions stated in a tuple. Figure 2.2 shows the synchronized object life cycle for the process model given in Figure 2.1 containing object life cycles of classes *Order*, *Product*, and *Invoice*. A data state within an object life cycle is only reachable if the dependencies described by the synchronization edges are fulfilled. Multiple synchronization edges with the same target data state are handled with respect to the origin of the source data state. If they belong to the same object life cycle, the described dependencies are disjunctions. If they belong to different object life cycles, the described dependencies are conjunctions. For instance, an *Order* may only reach state *archived*, if the *Invoice* currently is either in state *i* (for initial), state *canceled*, or state *paid*.

When a data state transition within an object life cycle shall take place, the synchronization validation function has to be executed for the affected synchronization edges. If an appropriate subset regarding the mentioned disjunctions and conjunctions evaluates to true, the transition may take place. Otherwise, the transition must not execute.

Definition 6 (Synchronization Validation Function)**.**
Given a synchronization edge $se = (src, tgt, dep)$, the *synchronization validation function* $\xi : SE \rightarrow \{true, false\}$ evaluates to true, if both data state transitions are enabled or if the data state src is active (either dependency type) or was active earlier (dependency type previously) in the corresponding object life cycle. Otherwise, ξ evaluates to false. ◇

3 Mapping a Process Model with Data Constraints to a Petri net

The generic process model described in the last section builds the basis for process description languages currently used in industry as, e.g. BPMN [22], EPCs [10], and activity diagrams [23]. These languages usually lack formal semantics and analysis techniques to check, amongst others, behavioral consistency. Therefore, we utilize Petri nets [24], a well established formalism to verify various properties of process models [28]. Most existing process description languages can be transformed into Petri nets; [16] gives an overview. One such mapping was introduced by Dijkman et al. [5] for BPMN 1.0 whose modeling constructs with respect to control flow are a superset of the ones presented in Definition 1 allowing to generalize that mapping. Though, the consideration of data is omitted. In this paper, we utilize this mapping as basis for the control flow mapping and extend it by a set of eleven rules to cover the data flow as well. Figure 3.1 summarizes these rules. In fact, the combination of rules given in [5] and the ones given in Figure 3.1 transforms a process model into its Petri net representation allowing further behavioral correctness checks as the weak conformance check introduced in Section 4.

Application of this rule set requires some assumptions to hold: (i) the process model follows Definition 1, (ii) the data annotations specify the information required to execute an activity (read) and the information expected to exist after termination (write), (iii) multiple data nodes with the same name read or written by one activity are disjunctive while data nodes with different names are conjunctive, (iv) XOR decision are based on the data results of the activity directly preceding the XOR gateway, (v) all data nodes with the corresponding data states required for some view on the process model are annotated to the activities, but the data annotation does not need to be complete in terms of comprising all possibly occurring data state transitions during process execution, (vi) an activity is enabled, if and only if the control flow reaches that activity, all data objects read by the activity exist in the states specified by the data nodes in the process model, and the synchronization validation function ξ evaluates to true for all affected synchronization edges, and (vii) concurrent read of a data node is allowed, whereas concurrent write or a mixture of concurrent read and write are forbidden.

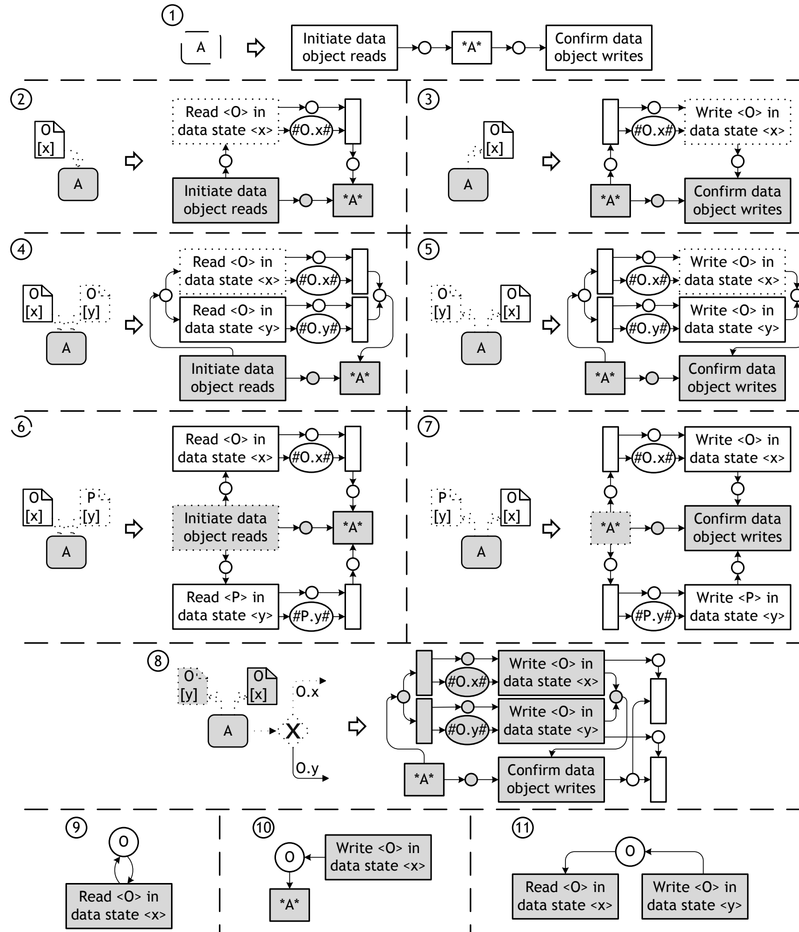

Figure 3.1: Rules to map data dependencies of a process model to a Petri net

The mapping rules introduced in this paper can be distinguished into three categories: control flow mapping (rules 1 and 8), data flow mapping (rules 2 to 7), and concurrency handling using semaphores (rules 9 to 11). For all rules from Figure 3.1, it holds that the white modeling constructs are the ones, which are tackled or affected by a rule. The gray modeling constructs are helpers setting the context for the white modeling constructs. Rule 1 extends the mapping of one activity to one transition from [5] to a set of three transitions to initiate data reads and to confirm and synchronize data writes. The second transition comprises the actual work performed during activity execution indicated by two asterisks enclosing the activity label. Rule 2 describes the read of a single data node with a specific data state, rule 4 describes the reading of one data node in one out of two data states (disjunction), and rule 6 describes the read of two independent data nodes (conjunction). Rules 3, 5, and 7 describe the corresponding write procedures. The semaphore place is marked based on the type of access to the data node by an activity. We distinguish read

(rule 9), write (rule 10), and modifying (rule 11) access. Rule 8 introduces XOR split determinism. During the mapping process, places and labeled transitions, except for the *initiate read* and *confirm write* transitions, with identical labels are identical and are therefore merged into single places or activities respectively.

The presented rules guarantee that the resulting Petri net satisfies the soundness property [28] by construction under the assumptions that no concurrent data modifications take place and that the original process model is deadlock and lifelock free from the control flow perspective. Each of the fragments replacing an activity or a data node following rules 1 to 7 is a single entry single exit fragment and sound such that their composition also remains sound. The soundness property also holds for the semaphore rules 9 to 11, if no concurrent data access takes place, because there is always a transition consuming the token and either the same transition or one succeeding it shortly puts the token back to the semaphore place without influencing the control flow. With respect to our assumption in Section 2 that data conditions assigned to control flow edges are non-blocking, we can safely reason that rule 8 does not induce deadlocks into the net. Finally, the mapping from [5] produces sound Petri nets and therefore, the resulting Petri net after applying control flow and data flow rules is sound by construction.

4 Weak Conformance

The notion of weak conformance has been initially proposed in [18] as extension to the notion of object life cycle conformance [13, 26] to allow the support of underspecified process models. A fully specified process model contains all reads and writes of data nodes by all activities. Additionally, each activity reads and writes at least one data node except for the first and last activities, which may lack reading respectively writing a data node in case they only create respectively consume a data node. In contrast, underspecified process models may lack some reads or writes of data nodes such that they are implicit, performed by some other process, or they are hidden in aggregated activities changing the state multiple times with respect to the object life cycle. Though, full support of underspecified process models requires that the process model may omit state changes of data nodes although they are specified in the object life cycle.

Table 4.1: Applicability and time complexity of data conformance computation algorithms

Attribute	[13, 26]	[31]	[18]	this
Full specification	+	+	+	+
Underspecification	-	o	+	+
Synchronization	-	-	-	+
Complexity	poly.	exp.	–	exp.

In this paper, we extend the notion of weak conformance to also support object life cycle synchronization. First, we compare different approaches to check for conformance between a process model and object life cycles. Table 4.1 lists the applicability and specifies the time complexity of the computation algorithms for approaches described in [13, 26], [31], [18], and this paper. The notion from [13, 26] requires fully specified process models and abstracts from inter-dependencies between object life cycles by not considering them for conformance checking in case they are modeled. Conformance computation is done in polynomial time. In [31], underspecification of

process models is partly supported, because a single activity may change multiple data states at once (aggregated activity). Though, full support of underspecified process models would require that the process model may omit data state changes completely although they are specified in the object life cycle. Synchronization between object life cycles is not considered in that approach and complexity-wise, it requires exponential time. [18] supports fully and underspecified process models but lacks support for object life cycle synchronization, which is then solved by the extension described in this section. For [18], no computation algorithm is given such that no complexity can be derived. The solution presented in this paper requires exponential time through the Petri net mapping and subsequent soundness checking as described in Section 4.2. However, state space reduction techniques may help to reduce the computation time for soundness checking [8]. The choice of using soundness checking to verify weak conformance allows to check for control flow soundness as well as weak conformance in one analysis and still allows to distinguish occurring violations caused by control flow or data flow.

4.1 The Notion of Weak Conformance

Weak conformance is checked for a process model with respect to the object life cycles referring to data classes used within the process model. To such concept, we refer as *process scenario* $h = (m, \mathcal{L}, C)$, where m is the process model, $\mathcal{L}$ is the synchronized object life cycle, and C is the set of data classes. Next, we define several notions for convenience considerations before we introduce the notion of weak conformance. Let $f \in \mathfrak{F}_m$ be a data flow edge of process model m. With f_A and f_D, we denote the activity and data node component of f, respectively. For instance, if f is equal to (a, d) or to (d, a), then (in both cases) $f_A = a$ and $f_D = d$. With $\vartheta(f)$, we denote the data state r_d involved in a read ($f = (d, a) \in \mathfrak{F}$) or write ($f = (a, d) \in \mathfrak{F}$) operation. We denote the set of synchronization edges having data state r_d as target data state with SE_r. Further, $a \Rightarrow_m a'$ denotes that there exists a path in process model m which executes activity $a \in A_m$ before activity $a' \in A_m$. Analogously, $s \Rightarrow_{l_c} s'$ denotes that there exists a path in the object life cycle l_c of data class c which reaches state $s \in S_c$ before state $s' \in S_c$.

Definition 7 (Weak Data Class Conformance)**.**
Given process scenario $h = (m, \mathcal{L}, C)$, $m = (N, D, Q, \mathfrak{C}, \mathfrak{F}, type, \mu, \varphi)$ and $\mathcal{L} = (L, SE)$, process model m satisfies *weak conformance* with respect to data class $c \in C$ if for all $f, f' \in F$ such that $f_D = d = f'_D$ with d referring to c holds (i) $f_A \Rightarrow_m f'_A$ implies $\vartheta(f) \Rightarrow_{l_c} \vartheta(f')$, (ii) $\forall se \in SE_{\vartheta(f')}$ originating from the same object life cycle

$l \in L : \exists \xi(se) == true$, and (iii) $f_A = f'_A$ implies f represents a read and f' represents a write operation of the same activity. ◇

Given a process scenario, we say that it satisfies *weak conformance*, if the process model satisfies weak conformance with respect to each of the used data classes. Weak data class conformance is satisfied, (i),(iii) if for the data states of each two directly succeeding data nodes referring to the same data class in a process model there exists a path from the first to the second data state in the corresponding object life cycle and (ii) if the dependencies specified by synchronization edges with a target state matching the state of the second data node of the two succeeding ones hold such that all dependency conjunctions and disjunctions are fulfilled. Two data nodes of the same class are directly succeeding in the process model, if either (1) they are accessed by the same activity with one being read and one being written or (2) there exists a path in the process model in which two different activities access data nodes of the same class in two data states with no further access to a node of this data class in-between.

The process model of the process scenario used in this paper is given in Figure 2.1. It contains three data classes: *Order*, *Product*, and *Invoice*. The corresponding synchronized object life cycle is shown in Figure 2.2. The process model satisfies weak conformance with respect to data class *Invoice* and does not satisfy weak conformance with respect to the other data classes. Indeed, there exists a path in the process model, which executes activity *Check stock* before activity *Send bill* such that both are directly succeeding with regards to accessing nodes of class *Order* in states *confirmed* and *accepted* respectively. However, there does not exist a path from state *confirmed* to state *accepted* in the object life cycle. The weak conformance check regarding class *Product* fails for synchronization issues. States *in stock* and *not in stock* can only be reached, if the order is in state *accepted* once the transition to either of the mentioned states of the product shall occur. But as there exists one case where this dependency of type *currently* does not hold (transition from data state *i* to state *in stock* by activity *Check stock*), the weak conformance check is not satisfied. In fact, in the given process scenario, the order is never in state *accepted* when the product shall transition to state *in stock* or state *not in stock*.

4.2 Computation of Weak Conformance via Soundness Checking

A given process scenario $h = (m, \mathcal{L}, C)$ can be checked for weak conformance by applying the following four steps in sequence:

1. Map the process model m and the synchronized object life cycle $\mathcal{L}$ to Petri nets,
2. integrate both Petri nets,
3. post-process the integrated Petri net and transform it to a workflow net system, and
4. apply soundness checking to identify violations within the process scenario h.

Before we discuss these four steps, we recall the notions of preset and postset. A preset of a transition t respectively a place p denotes the set of all places respectively transitions directly preceding t respectively p. A postset of a transition t respectively a place p denotes the set of all places respectively transitions directly succeeding t respectively p.

1—Petri net mapping: The process model is mapped to a Petri net following the rules described in [5] for the control flow and in Section 3 for the data flow. The mapping of the synchronized object life cycle is split. First, each single object life cycle $l \in L$ is mapped to a Petri net, which than secondly are integrated utilizing the set of synchronization edges. The mapping of single object life cycles utilizes the fact that Petri nets are state machines, if and only if each transition has exactly one preceding and one succeeding place [29]. Thus, each state of an object life cycle is mapped to a Petri net place and each data state transition connecting two states is mapped to a Petri net transition connecting the corresponding places.

For each typed synchronization edge, one place is added to the Petri net. If two typed synchronization edges have the same source and the same dependency type, target the same object life cycle, and if the corresponding target states each have exactly one incoming synchronization edge, both places are merged to one. Similarly, two places are merged, if two typed synchronization edges have the same target, the same dependency type, and origin from the same object life cycle. The preset of an added place comprises all transitions directly preceding the places representing the source and the target data states of the corresponding synchronization edge. The postset of an added place comprises all transitions directly preceding the place representing the target state of the synchronization edge. For currently typed edges, the postset additionally comprises the set of all transitions directly succeeding the place representing the source state.

For each untyped synchronization edge, one transition is added to the Petri net. If $\bigcap_{se_T}\{src \cup tgt\} \neq \emptyset$ for two untyped synchronization edges, i.e. they share one data state, then both transitions are merged. The preset and postset of each transition comprise newly added places; one for each (transitively) involved synchronization edge for the preset and the postset respectively. Such preset place directly succeeds the transitions that in turn are part of the preset of the place representing the data state from which the data state transition origins. Such postset place directly precedes the transition representing the corresponding source or target transition of the typed synchronization edge. Figure 4.1 visualizes this for synchronization edges

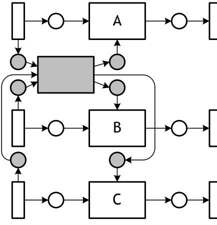

Figure 4.1: Mapping of untyped synchronization edges

$se_{T,1} = (a, b)$ and $se_{T,2} = (b, c)$. The gray colored places and transition have been added to the Petri net.

2—Petri net integration: First, data states occurring in the object life cycles but not in the process model need to be handled to ensure deadlock free integration of both Petri nets. We add one place p to the Petri net, which handles all not occurring states, i.e. avoids execution of these paths. Let each q_i be a place representing such not occurring data state. Then, the preset of each transition t_j being part of the preset of q_i is extended with place p, if the preset of t_j contains a data state which postset comprises more than one transition in the original Petri net mapped from the synchronized object life cycle.

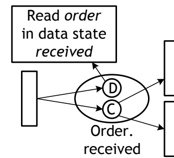

Figure 4.2: Internal places for a place representing a data state

Each data state represented as place in the Petri net mapped from the process model consists of a control flow and a data flow component as visualized in Figure 4.2 with C and D. Within the integrated Petri net, the control flow component is responsible for the flow of the object life cycle and the data flow component is responsible for the data flow in the process model. The integration of both Petri nets follows three rules, distinguishable with respect to read and write operations. The rules use the data flow component of data state places.

(IR-1) A place p from the object life cycle Petri net representing a data state of a data class to be read by some activity in the process model is added to the preset of the transition stating that this data node (object) is read in this specific state, e.g. the preset of transition *Read order in data state received* is extended with the place

representing data state *received* of class *Order* (cf. Figure 4.3), and (IR-2) a new place q is added to the integrated Petri net, which extends the postset of the transition stating that the data node (object) is read in the specific state and which extends the preset of each transition being part of the postset of place p, e.g. the place connecting transition *Read order in data state received* and the two transitions succeeding the place labeled *Order.received*. (IR-3) Let v be a place from the object life cycle Petri net representing a data state of a class to be written by some activity in the process model. Then a new place w is added to the integrated Petri net, which extends the preset of each transition being part of the preset of w and which extends the postset of the transition stating that the data node (object) is written in the specific state, e.g. the place connecting the two transitions preceding the place labeled *Product.inStock* and the transition *Write product in data state inStock*.

3—Workflow net system: Soundness checking has been introduced for workflow net systems [17,28]. Workflow nets are Petri nets with a single source and a single sink place and they are strongly connected after adding a transition connecting the sink place with the source place [28]. The integrated Petri net needs to be post-processed towards these properties by adding *enabler* and *collector* fragments. The enabler fragment consists of the single source place directly succeeded by a transition y. The postset of y comprises all places representing an initial data state of some object life cycle and the source place of the process model Petri net. The preset of each place is adapted accordingly.

The collector fragment first consists of a transition t preceding the single sink node. For each distinct data class of the process scenario, one place p_i and one place q_i are added to the collector. Each place p_i has transition t as postset[1]. Then, for each final data state of some object life cycle, a transition u_i is added to the collector. Each transition u_i has as preset the place representing the corresponding data state and some place q_i referring to the same data class. The postset of a transition u_i is the corresponding place p_i also referring to the same data class. Additionally, a transition z succeeded by one place is added to the collector. The place's postset is transition t. The preset of z is the sink place of the process model Petri net. The postset of z is extended with each place q_i.

Next, the synchronization places need to be considered. If a typed synchronization edge involves the initial state of some object life cycle as source, then the corresponding place is added to the postset of transition y of the enabler fragment. For all synchronization edges typed previously, the postset of the corresponding place is extended with transition t of the collector. If a currently typed synchronization edge involves a final state of some object life cycle as source, then the corresponding place is added to the postset of the corresponding transition u_i of the collector fragment.

[1] Generally, we assume that addition of one element a to the preset of another element b implies the addition of b to the postset of a and vice versa.

Finally, the semaphore places need to be integrated. Therefore, for each semaphore place, the preset is extended with transition y from the enabler and the postset is extended with transition t from the collector fragments. Now, connecting sink and source node, the workflow net is strongly connected. A workflow net system consists of a workflow net and some initial marking. The workflow net is given above and the initial marking puts a token into the single source place and nowhere else. Figure 4.3 shows the final workflow net system, where the gray colored modeling constructs represent the control flow, the white ones represent the data access including the semaphore places, the black places indicate the XOR split determinism, and the shaded constructs are the enabler and the collector.

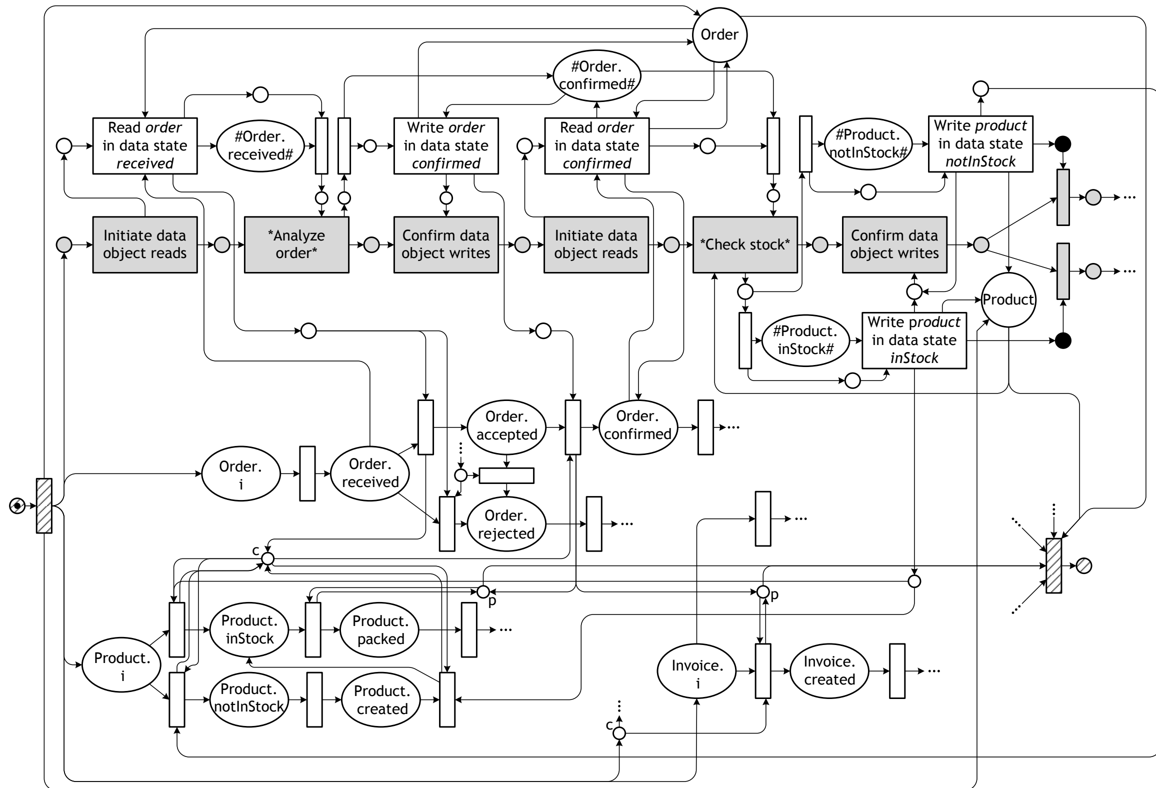

Figure 4.3: Extract of workflow net system representing the process scenario given in Figure 2.1 and 2.2

4—Soundness checking: Assuming control flow correctness, if the workflow net system satisfies the soundness property [28], no contradictions between the process model and the object life cycles exist and all data states presented in all object life cycles are implicitly or explicitly utilized in the process model, i.e. all paths in the

object life cycles may be taken. If it satisfies the weak soundness property [17], no contradictions between the process model and the object life cycles exist but some of the data states are never reached during execution of the process model; for instance, data state *rejected* of class *Order* in the given process scenario. As indicated above, the given process scenario does not satisfy the notion of weak conformance. Thus, the workflow net system neither fulfills the soundness nor the weak soundness property. It deadlocks for two reasons. First, transition *Read order in data state accepted* (not shown in Figure 4.3) will never be enabled, because it requires a token in place *Order.accepted* but this token already advanced to place *Order.confirmed*. Second, the workflow net system deadlocks when trying to write either data state *inStock* or state *notInStock* of class *Product*. With respect to Figure 2.2, both states are only allowed to be written, if the *Order* is in state *accepted* at this point in time, which is not the case. In case, control flow inconsistencies would appear, places and transitions representing the control flow would cause the violation allowing to distinguish between control flow and data conformance issues.

Validation. The described approach reliably decides about weak conformance of a process scenario. It takes sound Petri net fragments as input and combines them with respect to specified data dependencies. Single source and sink places are achieved through the addition of elements either marking the original source places or collecting tokens from the original final places. Thus, they do not change the behavior of the process model and the object life cycles, i.e. they do not influence the result.

5 Related Work

The increasing interest in the development of process models for execution has shifted the focus from control flow to data flow perspective leading to integrated scenarios providing control as well as data flow views. One step in this regard are object-centric processes [4, 21, 33] that connect data classes with the control flow of process models by specifying object life cycles. [12] introduces the essential requirements of this modeling paradigm. [13,26] present an approach, which connects object life cycles with process models by determining commonalities between both representations and transforming one into the other. Covering one direction of the integration, [14] derives object life cycles from process models considering synchronization between actions (state transitions). Similarly, [33] defines synchronization dependencies between transitions of different object life cycles. [30] also stresses the importance of handling inter-dependencies between data classes for process execution they refer to as coupling that in turn corresponds to typed synchronization edges. While we specify these inter-dependencies explicitly in the object life cycles, the authors predict probable couplings between implementations of data objects. Further, in contrast to these works, we combine both synchronization methods. Tackling the integration of control flow and data, [19,20] enable to model data constraints and to enforce them during process execution directly from the model. Similar to the mentioned approaches, we concentrate on integrated scenarios incorporating process models and object life cycles removing the assumption that both representations must completely correspond to each other. Instead, we set a synchronized object life cycle as reference that describes data manipulations allowed in a traditionally modeled process scenario, i.e. activity driven as with, for instance, BPMN [22].

Correctness, or compliance, in process models often refers to checks of the process model with respect to a defined rule set containing, for instance, business policies. The field of compliance is well researched, especially with respect to control flow compliance [2, 9, 25]. However, some works considered data for correctness as well. [11] describes a multi perspective compliance checking including data. [3] introduces means to check for compliance with respect to data dependencies, e.g. an object is required to be in a certain state for activity execution. Compared to our approach, the authors require to explicitly state data dependency rules instead of checking against a graphical representation as, for instance, object life cycles.

Furthermore, [15] applies compliance checking to object-centric processes by creating process models following this paradigm from a set of rules. These rules most often specify control flow requirements. [7] provides a technique to check for conformance of object-centric processes containing multiple data classes by mapping to an interaction conformance problem, which can be solved by decomposition into smaller sub-problems, which in turn are solved by using classical conformance checking techniques. [33] introduces a framework that ensures consistent specialization of object-centric processes, i.e. it ensures consistency between two object life cycles. In contrast, we check for consistency between a traditional process model and an object life cycle. Eshuis [6] uses a symbolic model checker to verify conformance of UML activity diagrams [23] considering control and data flow perspectives while data states are not considered in his approach. [13] introduces compliance between a process model and an object life cycle as the combination of object life cycle conformance (all data state transitions induced in the process model must occur in the object life cycle) and coverage (opposite containment relation). [31] introduces conformance checking between process models and product life cycles, which in fact are object life cycles, because a product life cycle determines for a product the states and the allowed state transitions. Compared to the notion of weak conformance, both notions do not support data synchronization and both set restrictions with respect to data constraints specification in the process model.

There do also exist approaches to represent data in Petri nets. [1] introduces a mapping from BPMN to Petri nets based on six rules. Compared to our mapping, the authors duplicate transitions in the Petri net to specify each data constraint of the corresponding activity separately and did not address challenges with respect to parallel data access. [27] discusses WFD-nets, which are workflow nets extended with data. WFD-nets could directly be used to represent business processes with the disadvantage that data flow cannot be visualized graphically and data states are not regarded.

6 Conclusion

In this paper, we presented an approach for the integrated verification of control flow correctness and weak data conformance using soundness checking considering dependencies between multiple data classes, e.g. an order is only allowed to be shipped after the payment was received but needs to be shipped with an confirmed invoice in one package. Therefore, we introduced the concept of synchronized object life cycles. For checking data correctness, we use the notion of weak conformance and extended it with means for object life cycle synchronization. Additionally, we described a mapping of data constraints modeled within a process model to Petri nets extending an existing control flow mapping. The resulting Petri net is integrated with a Petri net representation of the synchronized object life cycle, which then is used for soundness checking. With respect to the places or transitions causing soundness violations, we can distinguish between control flow and data flow issues and therefore, we can verify the notion of weak conformance. Revealed violations can be highlighted in the process model and the synchronized object life cycle to support correction. In this paper, we focused on the violation identification such that correction is subject to future work.

References

[1] Ahmed Awad, Gero Decker, and Niels Lohmann. Diagnosing and Repairing Data Anomalies in Process Models. In *BPM Workshops*, pages 5–16. Springer, 2010.

[2] Ahmed Awad, Gero Decker, and Mathias Weske. Efficient Compliance Checking Using BPMN-Q and Temporal Logic. In *Business Process Management*, pages 326–341. Springer, 2008.

[3] Ahmed Awad, Matthias Weidlich, and Mathias Weske. Specification, Verification and Explanation of Violation for Data Aware Compliance Rules. In *ICSOC*, pages 500–515. Springer, 2009.

[4] David Cohn and Richard Hull. Business Artifacts: A Data-centric Approach to Modeling Business Operations and Processes. *IEEE Data Engineering Bulletin*, 32(3):3–9, 2009.

[5] Remco M. Dijkman, Marlon Dumas, and Chun Ouyang. Semantics and Analysis of Business Process Models in BPMN. *Information & Software Technology*, 50(12):1281–1294, 2008.

[6] Rik Eshuis. Symbolic Model Checking of UML Activity Diagrams. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 15(1):1–38, 2006.

[7] Dirk Fahland, Massimiliano de Leoni, Boudewijn F. Dongen, and Wil M. P. van der Aalst. Conformance Checking of Interacting Processes with Overlapping Instances. In *BPM*, pages 345–361. Springer, 2011.

[8] Dirk Fahland, Cédric Favre, Barbara Jobstmann, Jana Koehler, Niels Lohmann, Hagen Völzer, and Karsten Wolf. Instantaneous Soundness Checking of Industrial Business Process Models. In *Business Process Management*, pages 278–293. Springer, 2009.

[9] Guido Governatori, Zoran Milosevic, and Shazia Sadiq. Compliance checking between business processes and business contracts. In *EDOC*, pages 221–232. IEEE, 2006.

[10] Gerhard Keller, August-Wilhelm Scheer, and Markus Nüttgens. Semantische Prozessmodellierung auf der Grundlage "Ereignisgesteuerter Prozessketten (EPK)". Technical Report Heft 89, Institut für Wirtschaftsinformatik, University of Saarland, 1992.

[11] David Knuplesch, Manfred Reichert, Linh Thao Ly, Akhil Kumar, and Stefanie Rinderle-Ma. Visual modeling of business process compliance rules with the support of multiple perspectives. In *ER*, pages 106–120. Springer, 2013.

[12] Vera Künzle, Barbara Weber, and Manfred Reichert. Object-aware Business Processes: Fundamental Requirements and their Support in Existing Approaches. *IJISMD*, 2(2):19–46, 2011.

[13] Jochen Küster, Ksenia Ryndina, and Harald Gall. Generation of Business Process Models for Object Life Cycle Compliance. In *Business Process Management*, pages 165–181. Springer, 2007.

[14] Rong Liu, Frederick Y. Wu, and Santhosh Kumaran. Transforming Activity-Centric Business Process Models into Information-Centric Models for SOA Solutions. *J. Database Manag.*, 21(4):14–34, 2010.

[15] Niels Lohmann. Compliance by design for artifact-centric business processes. In *Business Process Management*, pages 99–115. Springer, 2011.

[16] Niels Lohmann, Eric Verbeek, and Remco Dijkman. Petri net transformations for business processes – a survey. In *T. Petri Nets and Other Models of Concurrency*, pages 46–63. Springer, 2009.

[17] Axel Martens. On Usability of Web Services. In *Web Information Systems Engineering Workshops*, pages 182–190. IEEE, 2003.

[18] Andreas Meyer, Artem Polyvyanyy, and Mathias Weske. Weak Conformance of Process Models with respect to Data Objects. In *Services and their Composition (ZEUS)*, pages 74–80, 2012.

[19] Andreas Meyer, Luise Pufahl, Kimon Batoulis, Sebastian Kruse, Thorben Lindhauer, Thomas Stoff, Dirk Fahland, and Mathias Weske. Automating Data Exchange in Process Choreographies. In *CAiSE*, pages 316–331. Springer, 2014.

[20] Andreas Meyer, Luise Pufahl, Dirk Fahland, and Mathias Weske. Modeling and Enacting Complex Data Dependencies in Business Processes. In *BPM*, pages 171–186. Springer, 2013.

[21] Anil Nigam and Nathan S. Caswell. Business artifacts: An approach to operational specification. *IBM Systems Journal*, 42(3):428–445, 2003.

[22] OMG. Business Process Model and Notation (BPMN), Version 2.0, January 2011.

[23] OMG. Unified Modeling Language (UML), Version 2.4.1, August 2011.

[24] Carl A. Petri. *Kommunikation mit Automaten*. PhD thesis, Institut für Instrumentelle Mathematik, University of Bonn, 1962.

[25] Anne Rozinat and Wil M. P. van der Aalst. Conformance Checking of Processes Based on Monitoring Real Behavior. *Information Systems*, 33(1):64–95, 2008.

[26] Ksenia Ryndina, Jochen Küster, and Harald Gall. Consistency of Business Process Models and Object Life Cycles. In *MoDELS Workshops*, pages 80–90. Springer, 2006.

[27] Natalia Sidorova, Christian Stahl, and Nikola Trčka. Soundness Verification for Conceptual Workflow Nets with Data: Early Detection of Errors with the Most Precision Possible. *Information Systems*, 36(7):1026–1043, 2011.

[28] Wil M. P. van der Aalst. Verification of Workflow Nets. In *Application and Theory of Petri Nets*, pages 407–426. Springer, 1997.

[29] Wil M. P. van der Aalst. Workflow Verification: Finding Control-Flow Errors Using Petri-Net-Based Techniques. In *Business Process Management*, pages 161–183. Springer, 2000.

[30] Ksenia Wahler and Jochen Küster. Predicting Coupling of Object-Centric Business Process Implementations. In *Business Process Management*, pages 148–163. Springer, 2008.

[31] Zhaoxia Wang, Arthur H. M. ter Hofstede, Chun Ouyang, Moe Wynn, Jianmin Wang, and Xiaochen Zhu. How to Guarantee Compliance between Workflows and Product Lifecycles? Technical report, BPM Center Report BPM-11-10, 2011.

[32] Mathias Weske. *Business Process Management: Concepts, Languages, Architectures. Second Edition*. Springer, 2012.

[33] Sira Yongchareon, Chengfei Liu, and Xiaohui Zhao. A Framework for Behavior-Consistent Specialization of Artifact-Centric Business Processes. In *BPM*, pages 285–301. Springer, 2012.

Aktuelle Technische Berichte des Hasso-Plattner-Instituts

Band	ISBN	Titel	Autoren / Redaktion
90	978-3-86956-296-4	**Embedded Operating System Projects**	Uwe Hentschel, Daniel Richter, Andreas Polze (Hrsg.)
89	978-3-86956-291-9	**openHPI: 哈索•普拉特纳研究院的 MOOC（大规模公开在线课）计划**	Christoph Meinel, Christian Willems
88	978-3-86956-282-7	**HPI Future SOC Lab : Proceedings 2013**	Christoph Meinel, Andreas Polze, Gerhard Oswald, Rolf Strotmann, Ulrich Seibold, Bernhard Schulzki (Hrsg.)
87	978-3-86956-281-0	**Cloud Security Mechanisms**	Christian Neuhaus, Andreas Polze (Hrsg.)
86	978-3-86956-280-3	**Batch Regions**	Luise Pufahl, Andreas Meyer, Mathias Weske
85	978-3-86956-276-6	**HPI Future SOC Lab: Proceedings 2012**	Christoph Meinel, Andreas Polze, Gerhard Oswald, Rolf Strotmann, Ulrich Seibold, Bernhard Schulzki (Hrsg.)
84	978-3-86956-274-2	**Anbieter von Cloud Speicherdiensten im Überblick**	Christoph Meinel, Maxim Schnjakin, Tobias Metzke, Markus Freitag
83	978-3-86956-273-5	**Proceedings of the 7th Ph.D. Retreat of the HPI Research School on Service-oriented Systems Engineering**	Christoph Meinel, Hasso Plattner, Jürgen Döllner, Mathias Weske, Andreas Polze, Robert Hirschfeld, Felix Naumann, Holger Giese, Patrick Baudisch (Hrsg.)
82	978-3-86956-266-7	**Extending a Java Virtual Machine to Dynamic Object-oriented Languages**	Tobias Pape, Arian Treffer, Robert Hirschfeld
81	978-3-86956-265-0	**Babelsberg: Specifying and Solving Constraints on Object Behavior**	Tim Felgentreff, Alan Borning, Robert Hirschfeld
80	978-3-86956-264-3	**openHPI: The MOOC Offer at Hasso Plattner Institute**	Christoph Meinel, Christian Willems
79	978-3-86956-259-9	**openHPI: Das MOOC-Angebot des Hasso-Plattner-Instituts**	Christoph Meinel, Christian Willems
78	978-3-86956-258-2	**Repairing Event Logs Using Stochastic Process Models**	Andreas Rogge-Solti, Ronny S. Mans, Wil M. P. van der Aalst, Mathias Weske

ISBN 978-3-86956-303-9
ISSN 1613-5652