Journal on Data Semantics (2021) 10:143–163
https://doi.org/10.1007/s13740-021-00120-3

ORIGINAL ARTICLE

Formal Framework for Checking Compliance of Data-Driven Case Management

Stephan Haarmann[1] · Adrian Holfter[1] · Luise Pufahl[2] · Mathias Weske[1]

Received: 8 May 2020 / Revised: 15 January 2021 / Accepted: 25 January 2021 / Published online: 19 February 2021

Abstract
Business processes are often specified in descriptive or normative models. Both types of models should adhere to internal and external regulations, such as company guidelines or laws. Employing compliance checking techniques, it is possible to verify process models against rules. While traditionally compliance checking focuses on well-structured processes, we address case management scenarios. In case management, knowledge workers drive multi-variant and adaptive processes. Our contribution is based on the fragment-based case management approach, which splits a process into a set of fragments. The fragments are synchronized through shared data but can, otherwise, be dynamically instantiated and executed. We formalize case models using Petri nets. We demonstrate the formalization for design-time and run-time compliance checking and present a proof-of-concept implementation. The application of the implemented compliance checking approach to a use case exemplifies its effectiveness while designing a case model. The empirical evaluation on a set of case models for measuring the performance of the approach shows that rules can often be checked in less than a second.

Keywords Compliance checking · Case management · Model verification · Data-centric processes

1 Introduction

Business processes include activities that are executed by systems and workers to create business value. The tasks are in a relationship with each other and with data. This constrains the possible orders in which tasks are executed [37]. Business process management (BPM) provides methodologies, methods, and tools for companies to design, implement, enact, and analyze their processes [56]. At any point, companies must comply with laws and regulations, which must be reflected in the processes, respectively. For example, a company must get consent from their customers before processing personal data. Compliance checking is a set of methods that allow verifying processes against rules [20,46]: they enable companies to check process models at design-time, to monitor executions with regard to compliance during run-time, and to detect violations in event logs that record past behavior. We focus on model-based compliance checking to find violating execution paths at design-time and and situations leading to violations at run-time.

While traditionally compliance checking in BPM can be applied to highly structured processes that can be specified through imperative process models (e.g., using process modeling languages common in industry, such as Business Process Model and Notation (BPMN) [39]), there exists little support for compliance checking of knowledge-intensive processes. Such processes are executed by highly trained workers (i.e., *knowledge workers*) that perform various interconnected decision-making tasks to reach a certain goal for a case (e.g., consulting, healthcare, or education processes) [12]. They are richer in their variations, are information- and data-driven, and often require ad hoc adaptation at run-time [12].

Case management addresses these processes by enabling knowledge workers to specify, execute, and adapt them as

✉ Stephan Haarmann
stephan.haarmann@hpi.de

Adrian Holfter
adrian.holfter@student.hpi.de

Luise Pufahl
luise.pufahl@tu-berlin.de

Mathias Weske
mathias.weske@hpi.de

[1] Hasso Plattner Institute, University of Potsdam, Prof.-Dr.-Helmert Str. 2-3, 14482 Potsdam, Germany

[2] Software and Business Engineering, Technische Universitaet Berlin, Berlin, Germany

needed [53]. To this end, the fragment-based Case Management (fCM) approach combines activity-centric process fragments with data requirements [22]. Fragments can be instantiated and executed dynamically as well as added during execution. All fragments operate on shared data; data requirements, which are modeled in the fragments, lead to dependencies and, hence, synchronization among fragments. Due to the loose coupling, fCM case models describe highly concurrent and multi-variant processes in a compact yet precise way. This characteristic of fCM case models makes model verification hard (due to the exponentially growing state-space) but, at the same time, necessary (since errors may be easily overlooked by process designers).

Consider an emergency ward at a hospital (the detailed case model for the example is presented in Sect. 3). After a patient arrives, he or she will be examined, an X-ray may be conducted, and different treatments may be performed. Every patient is different, and it is the physicians' responsibility to decide on the right course of action. However, clinical guidelines and laws may specify rules that must not be violated, and an activity may require data objects, which must be gathered upfront. These rules and requirements can be captured in fCM's fragments. Physicians may than instantiate and execute activities dynamically to fit the specific case as long as the specified constraints are not violated. This provides physicians with guidance but still offers the required flexibility. To guide knowledge workers reliably, it is crucial that the fCM model complies with the given guidelines and laws.

In this paper, we present a framework for compliance checking case models at design-time and detecting situations that inevitably lead to compliance violations at run-time. According to Johnson a framework is defined as "a skeleton of an application that can be customized by an application developer" [28]. In this paper, we present a formal framework that can be implemented in different systems. It can be deployed as part of a larger compliance management system to verify case models and instances against temporal logic formulas. The developed framework provides model checking for case models by covering the traditional three steps of model checking: (1) formalization of the model, (2) formalization of the rule to check, and (3) model checking. Therefore, we present a translational semantics that maps case models to Petri nets. Additionally, we present adaptations reducing the state space to improve the efficiency of the compliance checking. We use the formalism for design-time compliance checking.

Furthermore, we present a solution that takes the current state of a running case (i.e., instance) and the formal model to check whether a compliance violation is inevitable for that specific instance. Furthermore, we can verify adaptations made at run-time: when a new fragment is added, the updated case model can be formalized. Using the current state of the system as a starting point, our framework can check whether the adapted model includes any violations.

This paper extends our previous work in [26]. We extend the paper by providing detailed and formal description of the translational semantics as pseudo-code algorithms. Furthermore, we evaluate our approach on a use case and empirically investigate case models of different complexity.

The remainder of this paper is structured as follows. In Sect. 2, we provide an overview of related work regarding case management, compliance checking, and Petri net-based formalization. Preliminaries, such as case models and Petri nets, are introduced in Sect. 3. Next, we provide a formalization for case models as well as adaptations used for compliance checking (Sect. 4). In Sect. 5, the approach of checking case models for compliance and an approach finding situations leading to violations at run-time is introduced and explained. Section 6 contains a description of our proof-of-concept implementation as well as an evaluation. In the evaluation, we discuss a case study centered on the process of computer-aided translation of documents. Furthermore, we empirically analyze the performance of our tool using case models of varying complexity and different assumptions. Finally, we conclude our paper in Sect. 7.

2 Related Work and Background

In the following, we provide an overview of related work divided into three categories: first, we describe related work in the field of compliance checking for business processes. Next, we give an overview of case management approaches and corresponding modeling languages. The last paragraph lists works about formalizing and analyzing process models with Petri nets.

2.1 Compliance Checking

Compliance checking examines whether processes adhere to regulations such as company guidelines and laws. Hashmi et al. survey compliance checking approaches aiming at an holistic overview [20]. Following the categorization introduced by Hashmi et al., our approach is based on model checking and allowing the verification of control flow and data related compliance rules of flexible processes at design-time and at run-time. Awad et al. verified BPMN process models against temporal logic properties [1] using a visual languages for rules (BPMN-Q) [4]. The authors extended the approach to data objects and their state [3]. Avad et al. limit processes to highly structured ones; rules are limited to ordering and existential constraints. More recent works by different authors lift assumptions and consider a growing set of perspectives such as data attributes [29], decision logic [18], roles [49], and more. Knuplesch et al. provide a visual language for multi-perspective compliance rules as well as an implementation to verify models and to monitor processes [30,35,36]. Fragment-based case management (fCM) models contain information about the control flow, data objects, and data dependencies. Therefore, we focus on the control flow perspective while considering data objects and their states similar to [3]. In contrast, we focus on semi-structured, flexible processes that are driven by data and decisions.

Besides compliance checking, we can assert compliant models through *compliance by design* methods. Such methods are common for flexible processes: declarative process modeling languages, e.g., DECLARE [42] or DCR graphs [24], are based on rules. Instead of building up the behavior in an imperative way, they enable scoping it through a set of constraints. Such constraints can also represent compliance rules. A process that is designed this way is compliant. However, corresponding models become complex if they contain highly structured process segments, and they do not support data-centric processes. While there exist extensions supporting additional perspectives, such as data and time [7,9], complexity and comprehensibility of the models remains problematic. Our approach addresses this gap. We target fCM that combines highly structured process fragments with declarative, data-driven dependencies.

Compliance management permeates the whole process life cycle. However, we focus on process models and, thus, limit this overview to respective work. The reader interested in compliance monitoring, conformance checking, or a posteriori violation detection is referred to the surveys by Hashmi et al. [20] and by Sackmann et al. [46].

2.2 Case Management

Case management aims to support knowledge workers performing processes that require flexibility and adaptability. Knowledge-intensive processes are often data-centric and driven by the knowledge workers [12]. In general, case management approaches are divided into adaptive case management and production case management. The former mixes design and execution into a single phase that allows knowledge workers to define the process on the go. Production case management assumes a (under-specified) base model designed upfront that can be adapted during execution [53]. In the following, we focus on production case management.

The Guard Stage Milestone (GSM) approach is a data-centric case management approach, which arranges activities in stages [27]. Stages have guards (data pre-conditions) and milestones (data post-conditions); if a guard is satisfied, the stage can be entered and the activities inside can be executed. Once a milestone is satisfied, the stage *can* be left. Multiple approaches to verify GSM models exists [5,16,34,51]; however, verification is in general undecidable [51]. Inspired

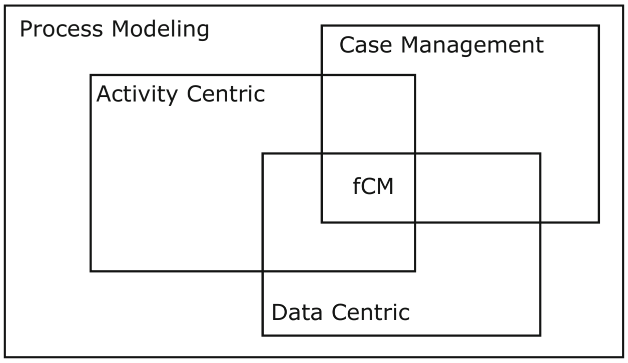


Fig. 1 Process modeling comprises activity-centric, data-centric, and hybrid approaches. Case management addresses knowledge-intensive (and often flexible) processes. fCM is a hybrid case-management approach that combines activity-centric and data-centric process modeling

by the GSM approach, the Case Management Model and Notation (CMMN) [40] standard has been developed. The key concept of CMMN are also stages consisting of activities with entry and exit criteria. Activities can be repetitive, mandatory, or optional. Dependencies among activities and stages can be defined by links. CMMN is rather an activity-centric case management approach because data is captured implicitly and not modeled visually.

Despite an existing standard, other related approaches were still continued or newly developed, most prominently PHILharmonicFlows [32], DCR Graph (Dynamic Condition Response Graph) [50], and fragment-based Case Management (fCM) [22]. PHILharmonicFlows [32] is a data-centric process modeling approach, which represents an alternative to widespread activity-centric process modeling languages such as BPMN.

DCR Graphs [50], an alternative activity-centric modeling approach, consist of a set of process activities, which are visually connected by a temporal constraint relation instead of traditional control flow. However, data artifacts do not play an essential role.

We base our work on the fCM approach [22]. fCM is a hybrid approach combining ideas from data-centric and activity-centric process modeling. Hybrid approaches have been used, e.g., by [10,41], to strengthen and verify the connection of the processes to information systems and databases or by [47] to incorporate imperative and declarative business rules. fCM is different from [10,41] because it focuses on flexible and knowledge-intensive processes. While [47] combines declarative and imperative processes modeling resulting in two connected models of the process, fCM combines activity-centric process fragments with object life cycles and data models. fCM fragments support imperative control flow and declarative data conditions, while the data model and object life cycles complement the model with detailed information about the objects in the process. Thereby, structured parts can be expressed in a comprehensible way while flexibility through loose coupling of fragments adds little complexity to the models visual representation [23]. We detail the fCM approach in Sect. 3.

2.3 Formal Semantics for Process Models

Business process models are used for specification, analysis, and verification. These tasks require a precise understanding of the models' semantics. To this end, formal models are employed. A common formalization for business processes are Petri nets, which offer a visual representation and precise semantics. Van Hee et al. demonstrate how Petri nets can be used to model and verify business processes [21]. The authors focus on workflow nets and present different variants. While workflow-nets can be used to model processes directly, they quickly become hard to comprehend when used for multi-variant and highly concurrent processes. We use fCM for a concise representation and establish formal semantics through a Petri net mapping, which is hidden from the end-user. Thereby, we combine the advantages of Petri nets clear semantics with the benefits of a more abstract modeling language such as fCM.

Petri net-based semantics for BPMN models have been defined by Dijkman et al. [13]. Awad et al. added data access semantics [3] (for data objects with a finite set of states). A first mapping of fCM models to Petri nets has been defined by Sporleder [52]. These three mappings are the foundation for our Petri net-based formalization. In contrast to related work, we consider the concurrent execution of activities, adaptations necessary to verify flexible models, run-time extensions (of the model), as well as the mapping's application to compliance checking for *flexible* processes.

Besides Petri nets, process algebras, such as π-calculus [38] or CSP [25], are another common means to formalize process models. While Petri nets advantage is a visual representation, process algebras allow composing and decomposing processes. We use Petri nets since they are widely used in the BPM community, are structured and visualized similarly to the original process model, and are supported by a set of sophisticated tools, such as LoLA [48].

3 Preliminaries

Model-based compliance checking investigates whether a process model adheres to rules. To automate this step, it is necessary that both the model and the rules have formal semantics. We dedicate this section to introduce and conceptualize both fragment-based case management (fCM) models as well as Petri nets, which are used to assign formal semantics.

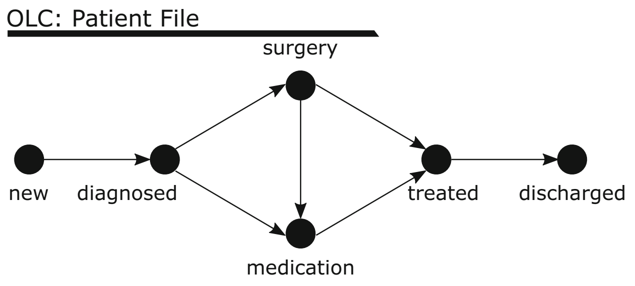


Fig. 2 OLC for an object of type *patient file*: A state change is only valid if the corresponding nodes are connected

3.1 Fragment-Based Case Management

In case management, a case evolves around data. The knowledge workers consult and create data objects while performing tasks. Thereby, they advance the state of the process. Each case has a central object, the case object. Progressing the state of the case object is the main obligation of a case. On a model level, a domain model (Definition 1) of a fragment-based case model describes the types of data objects, the (data) classes, that are involved in a case.

Definition 1 (*Domain model*) Let C be a non-empty, finite set of classes in a case, and let $c_c \in C$ be the central case class. The domain model for a case model is $D = (C, c_c)$.

Consider an emergency ward. Patients arrive, are diagnosed and treated. Meanwhile, multiple documents are created and updated: a *patient file* contains all patient related information (it is the case object), but further objects such as an *X-ray*, a *prescription*, and a *report* may be created. Certain objects are required by certain steps. As in the case of the *prescription* object, which is necessary to administer medication.

During a case, the involved objects run through a series of states. A patient arrives (*patient file* in state *new*), is diagnosed (*patient file* in state *diagnosed*), and, eventually, is *released*. The possible sequences for each object are limited, e.g., a patient that just arrived in the emergency ward cannot undergo surgery without a prior diagnosis. A class-specific object life cycle (OLC) defines all possible states and state transitions of corresponding objects (Definition 2 and Fig. 2).

Definition 2 (*Object life cycle*) Let C be a set of classes, $c \in C$ a class, and Q_c the set of possible states for objects of the class c. Furthermore, let $\overset{c}{\rightarrow} \subseteq Q_c \times Q_c$ be the set of valid state transitions. The object life cycle of c is a state transition system $l(c) = (Q_c, \overset{c}{\rightarrow})$. $l(C)$ describes the set of object life cycles corresponding to classes in C.

Data objects with state information are essential to case models. They are not only input and output of activities, they are also used to define the goal state of the case. In order to close a case properly, the goal must have been accomplished. In our example, the patient must be discharged. This can be expressed by a data condition that requires the object *patient file* in state *discharged*. Furthermore, the inputs and outputs of activities can be specified by such data conditions as well.

Definition 3 (*Data object condition*) Let C be a set of classes, c a class, and Q_c the set of corresponding states. We define that $(c, q) \in \{c\} \times Q_c$ is a data object condition that requires an object of type c in state q. $O(c) = \{c\} \times Q_c$ is the set of all possible data object conditions for a single class, and $O(C)$ denotes the set of all possible data object conditions of a set of classes C.

Classes are instantiated and objects are updated through activities. Knowledge workers drive the case by deciding on the next activity and executing it. However, they cannot choose freely. Control flow creates dependencies among activities of one process fragment (see Definition 4), and data requirements add additional constraints. A fragment encapsulates activities that must be executed in a structured way. There are initial fragments with start events representing the beginning of a new case and regular fragments without events that can be instantiated and executed arbitrarily often (as long as all data requirements are satisfied).

A case in the emergency ward (cf. Fig. 3) always begins with the admission of a new patient followed by the diagnoses. However, different treatments, such as a surgery or medication, may be conducted before discharging the patient. The fragment *f1* (*diagnosis*) is an initial fragment. It starts with a new patient arriving, continues with examination, requesting an X-ray, and finally deciding on a treatment. Note, that the X-ray is made in another fragment (*f2*), which can be executed if the X-ray is required. Instances of fragments can run concurrently and thereby influence each other through data.

Definition 4 (*Fragment*) Let $D = (C, c_c)$ be a domain model and $l(C)$ the set of corresponding object life cycles, where $(Q_{ci}, \xrightarrow{c_i})$ denotes the life cycle of a class $c_i \in C$. Let $A = \{a_1, a_2, \ldots, a_j\}$ be a finite non-empty set of activities and G a finite set of exclusive gateways. We define a fragment as a tuple $f = (A, G, s, \xrightarrow{cf}, read, write)$ where

- s is the start event or the special value $\perp$ if the fragment has no start event;
- $\xrightarrow{cf} \subset (A \cup G \cup \{s\}) \times (A \cup G)$ is a set of control flow connections (a fragment is always acyclic);
- $read : A \rightarrow 2^{O(C)}$ assigns each activity a set of data conditions. For an activity $a \in A$, $read(a)$ denotes the data objects that can be read by a.

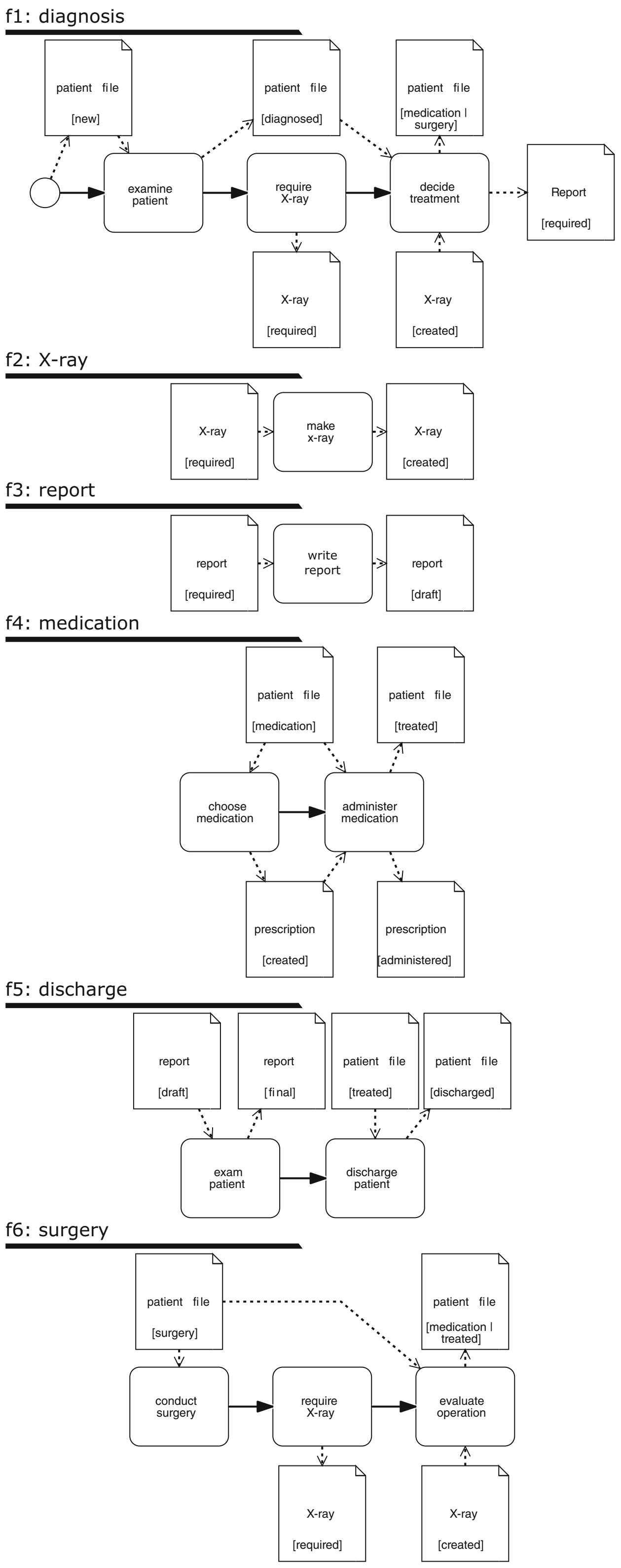


Fig. 3 Fragments of a case model describing the emergency ward at a hospital; we use a subset of the BPMN notation [39] to describe fragments. If an activity can write an object in one of multiple states, we represent this as a single node, separating the states with |(see fragment **f6**)

- $write : (A \cup \{s\}) \rightarrow 2^{O(C)}$ assigns each activity a second set of data conditions. Given an activity $a \in A$, $write(a)$ denotes the data objects that can be written by a.

Let $c \in C$ be a data class with the object life cycle $l(c) = (Q_c, \xrightarrow{c})$. Let $a \in A$ be an activity that reads and writes an object of type c then $\forall q_1 \in \{q \in Q_c | (c, q) \in read(a)\}, \exists q_2 \in \{q \in Q_c | (c, q) \in write(a)\} : (q_1, q_2) \in \xrightarrow{c}$. Furthermore, we define the set of input-sets and output-sets of a. Let $C_i^a = \{c | \exists q \in Q : (c, q) \in read(a)\}$ and $C_o^a = \{c | \exists q \in Q : (c, q) \in write(a)\}$ be the set of classes which objects may be read or written by a, respectively. The input sets of a are defined as $i(a) = \Pi_{c \in C_i^a} \{(c, q) \in read(a)\}$ and the set of output-sets as $o(a) = \Pi_{c \in C_o^a} \{(c, q) \in write(a)\}$ where Π forms all combinations of the sets (similar to a Cartesian product but creating a set of sets instead of a set of tuples).

Definitions 1–4 each specify an aspect of case models. Next, we need to combine the different components to one case model. The domain model, the OLCs, the termination conditions, and the fragments compose the case model (Definition 5). Case models are blue prints for a set of similar cases, but they are also under-specified. To find a balance between guidance and flexibility for the involved knowledge workers, case models couple fragments loosely and allow adaptation at run-time. Knowledge workers may extend the case model by adding new fragments that comply with the object life cycles at run-time.

Definition 5 (*Case model*) Let $D = (C, c_c)$ be a domain model, $l(C)$ the set of corresponding OLCs, and F a set of fragments defined over D. A case model is defined by tuple $M = (F, D, \mathfrak{T})$ where $\mathfrak{T} = \{t_1, t_2, \ldots, t_n\} \subseteq 2^{O(C)}$ is a set of termination conditions. The case can terminate if for any $t_i \in \mathfrak{T}$ all data object conditions $o \in t_i$ are satisfied.

3.2 Petri Nets

Just from looking at a case model, it is hard to determine all modeled variants and dependencies. Petri nets, in contrast, have clear execution semantics allowing an automated formal analysis [45]. They are commonly used to model concurrent systems and suit the setting of case management well. While a general Petri net is a simple bipartite graph consisting of nodes divided into places and transitions and arcs, we consider labeled Petri nets assigning labels to places and transition. Using the labels, we can reflect the relationships between the original case model and its Petri net formalization.

Definition 6 ((*Labeled*) *Petri Net*) Let P be a set of places and T a set of transitions. A Petri net is a tuple $N = (P, T, \rightarrow)$ where $\rightarrow \subseteq (P \times T) \cup (T \times P)$ connects the places with the transitions and vice versa. A labeled Petri net is a tuple $N = (P, T, \rightarrow, \lambda)$ where $(P, T, \rightarrow)$ is a Petri net and $\lambda : P \cup T \rightarrow (\Sigma \cup \{\perp\})$ is a function assigning a label or the null value $(\perp)$ to each transition and place.

In an instantiated Petri net, the current state is given by the position of tokens—this is called a marking. Each place can hold an arbitrary amount of tokens, which are indistinguishable. A transition consumes a token of each place from which an arc leads to the transition and produces a token on the places directly succeeding the transition. The firing of a transition changes the Petri net's state. In labeled Petri nets, a meaning is assigned to tokens via the label of places. Process models usually contain control-flow and data-flow. Similar to [3], we distinguish between data places and control flow places (see Definition 7).

Definition 7 (*Petri Net w. Control Flow and Data Places*) Following [43], we define a Petri net with control flow and data places as a tuple $N = (P, T, \rightarrow, \lambda, \sigma)$ where

- $(P, T, \rightarrow, \lambda)$ is a labeled Petri net, and
- $\sigma : P \rightarrow \{CF, D\}$ is a function that assigns each place a status where
  - CF denotes control flow places, and
  - D denotes data places.

4 A Formal Execution Semantics for Case Models

To verify case models using model checking, we must assign formal semantics to case models. Therefore, we provide a translational semantics mapping case models to Petri nets. We assume that all state changes of data objects performed by activities are reflected in the corresponding OLC. This property is called *object life cycle conformance*. Furthermore, inputs and outputs of an activity can be combined arbitrarily.

Activities belong to fragments and are connected by control and data flow. Different fragments are loosely coupled through data objects. The loose coupling and traditional control flow allows us to build the case model's Petri net representation bottom-up: we formalize activities followed by control flow dependencies to derive fragments. Next, we can translate the termination conditions before assembling the case model's formalization from the fragments and the termination condition.

Figure 4 provides an overview of the algorithms mapping a case model to a Petri net. The case model is provided as input to the overarching algorithm `translateCaseModel`. The algorithm translates each of the case model's fragments by calling `translateFragment`, and the termination condition by calling `translateTermination`

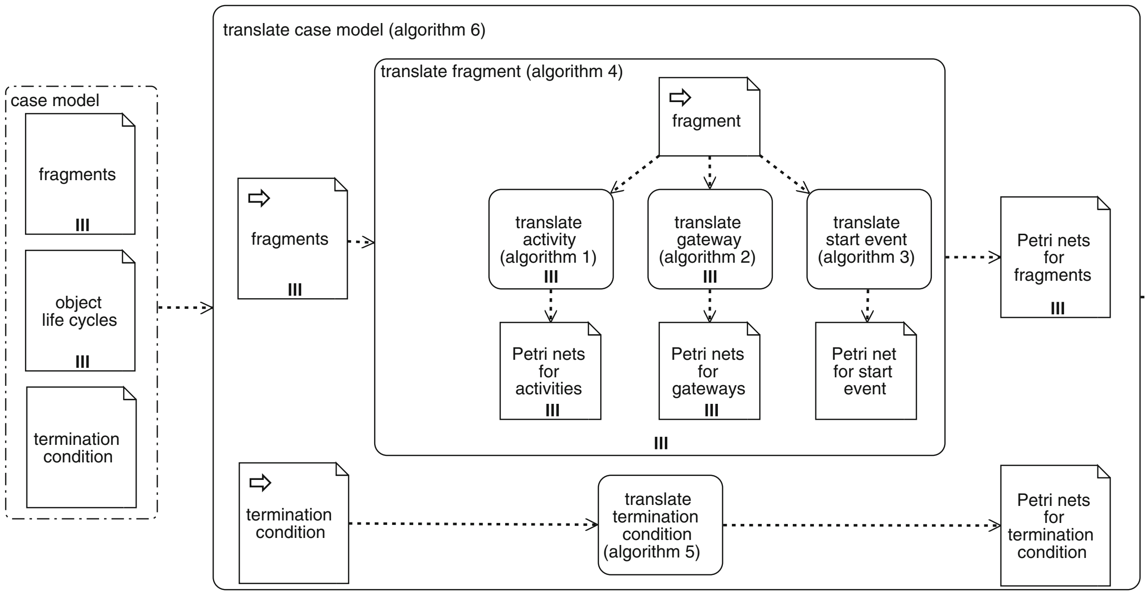

translate case model (algorithm 6)
translate fragment (algorithm 4)
case model
fragments
object life cycles
termination condition
fragments
fragment
translate activity (algorithm 1)
translate gateway (algorithm 2)
translate start event (algorithm 3)
Petri nets for activities
Petri nets for gateways
Petri net for start event
Petri nets for fragments
Petri net for case model
termination condition
translate termination condition (algorithm 5)
Petri nets for termination condition

Fig. 4 Overview of the algorithms presented in this section and the corresponding call hierarchy. As shown in the figure, most functions can run concurrently and synchronization only occurs at the level of the calling function, e.g., all calls to `translateFragment` can run concurrently, the calling instance of `translateCaseModel` waits for all outputs, our algorithms work sequentially to ease comprehension

`Condition`. A fragment is formalized by looking at individual control flow nodes, activities, gateways, and the start event. These can be mapped individually and independently before being assembled to a bigger Petri net representing the fragment. The Petri nets for all fragments are combined with the one for the termination condition to derive the final output: the Petri net representation of the whole case model. In the reminder of this section, we look at each of the mentioned mapping in more detail.

Mapping activities. We start by formalizing an activity (cf. Algorithm 1). To execute an activity, it has to be enabled by control flow, and its data precondition must be satisfied. We add a place with label `act[enabled]` for each activity `act` (line 5). A token on this place enables the activity (by control flow). Furthermore, we explicitly consider the *running* state of activities. Therefore, we add a place labeled `act[running]` (line 6).

To enter the running state, an activity reads and binds the data objects of *one* input set. Upon termination, these objects are either released or updated. A Petri net transition has one set of places from which it consumes tokens; thus, the mapping has one transition for each of the activity's input sets $inputSet \in i(act)$ (see Definition 4) labeled `begin act,inputSet` (line 8). The transition's pre-set contains the `act[enabled]` place (line 9) as well as a place for each element in the input set (lines 14–16). Such a place has a label `obj[state]` reflecting that an object `obj` is in a certain `state`. The transition's post-set contains `act[running]` but also a place `inputSet[bound]2act` indicating that the chosen input set is now bound (line 10), i.e., it cannot be used by another activity. Furthermore, transitions for initial activities, those without a predecessor node, put a token back on `act[enabled]` to allow new fragment instances to be started (lines 11–12).

Note each input set can be combined with each output set, since we assume that all contained state transitions are valid according to the object life cycles. Thus, we need to create transitions terminating activities for the cross product of input sets and output sets. Given an input set `inputSet` and an output set `outputSet`, we create a transition labeled `act,inputSet, outputSet` (line 19). The transition terminates the activity consuming a token from `act[running]` and releases the objects in the input set, reflected in the consumption of a token from `inputSet[bound]2act` (line 20). The place `input Set[bound]2act` asserts that only terminating transitions for the bound input set can fire. Consequently, all bound objects are either released or updated: the transition produces tokens for the data objects that have been read and those that are created (lines 21–23). Algorithm 1 duplicates some data places, which we merge in a later stage.

Consider the activity *decide treatment*. It has one input set requiring the *patient file* in state *diagnosed* and the *x-ray* in state *created*. Furthermore, it has two output sets as it will create a *report* object in state *required* and either update the state of the *patient file* to *medication* or to *surgery*. The corresponding Petri net is shown in Fig. 5. It has two transitions representing the termination, which additionally to the output set put a token on `x-ray[created]`, because the object

Algorithm 1: Function translateActivities

```
input  : a fragment (A,G,s,$\xrightarrow{cf}$,read,write)
output: a mapping activityNets, which assigns each activity a Petri net
1  activityNets = new Map();
2  for $act \in A$ do
3      pn = new EmptyPetriNetWithControlFlow();
4      for $inputSet \in i(act)$ do
5          enablementPlace = pn.createCfPlace().withLabel(act[enabled]);
6          runningCfPlace = pn.createCFPlace().withLabel(act[running]);
7          bindingPlace = pn.createDataPlace().withLabel(inputSet[bound]2act);
8          startingTransition = pn.createTransition().withLabel(begin
           act,inputSet);
9          startingTransition.addToPreset(enablementPlace);
10         startingTransition.addToPostset(runningCfPlace, bindingPlace);
11         if $|getIncomingCfFor(act)|=0$ then
12             startingTransition.addToPostset(enablementPlace);
13         end
14         for $(obj, state) \in inputSet$ do
15             dataObjectPlace = pn.createDataPlace().withLabel(obj[state]);
16             startingTransition.addToPreset(dataObjectPlace);
17         end
18         for $outputSet \in o(a)$ do
19             terminatingTransition =
               pn.createTransition().withLabel(terminate
               act,inputSet,outputSet);
20             terminatingTransition.addToPreset(runningCfPlace,
               bindingPlace);
21             for $(obj, state) \in (outputSet \cup \{(o,s) \in inputSet : \nexists\, (o,s')$ in
               $outputSet\})$ do
22                 dataObjectPlace =
                   pn.createDataPlace().withLabel(obj[state]);
23                 terminatingTransition.addToPostset(dataObjectPlace);
24             end
25         end
26     end
27     activityNets.put(act, pn);
28 end
29 return activityNets;
```

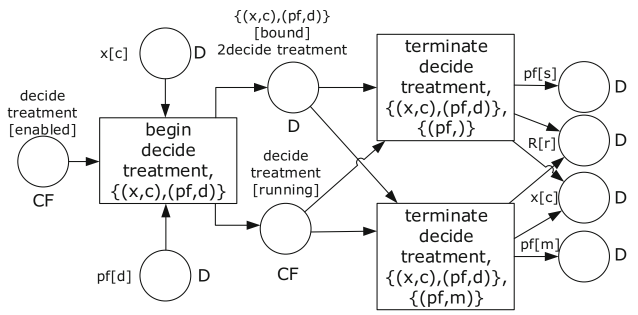


Fig. 5 Petri net formalization of the activity decide treatment. The following abbreviations are used: x for x-ray, pf for patient file, R for report, c for created, d for diagnosed, m for medication, s for surgery, and r for required. The Petri net is created by Algorithm 1

is bound upon start and released upon termination of *decide treatment*.

Translate gateway. Fragments may contain exclusive gateways. An exclusive gateway triggers one of its outgoing control flow arcs whenever one of its incoming control flow arcs got triggered. Given a gateway `gw`, we create a place labeled `gw[enabled]` (Algorithm 2, line 4). A token on this place represent a triggered incoming control flow. Furthermore, we add a transition for each outgoing control flow. We label it `gw, target` where `target` is the target node of said control flow (line 6). Each of these transitions has the place `gw[enabled]` in its pre-set (line 7).

Algorithm 2: Function translateGateways

```
input  : a fragment (A,G,s,$\xrightarrow{cf}$,read,write)
output: a map gatewayNets, which assigns each gateway a Petri net
1  gatewayNets = new Map();
2  for $gw \in G$ do
3      pn = new EmptyPetriNetWithControlFlow();
4      enablementCfPlace = pn.createCfPlace().withLabel(gw[enabled]);
5      for $(gw,successor) \in getOugoingControlFlow(gw)$ do
6          transition = pn.createTransition().withLabel(gw,successor);
7          transition.addToPreset(enablementCfPlace);
8      end
9      gatewayNets.put(gw, pn);
10 end
11 return gatewayNets;
```

Translating start events. Start events create new cases and may produce data objects. As such they can have different output sets similar to an activity. Algorithm 3 maps start events to Petri nets. If a fragment has a start event ($s \neq \perp$), we create a transition for each output set (line 6). If an object `obj` in state `state` is in the output set, a corresponding data place is added to the net and the transition's post set (lines 7–9).

Algorithm 3: Function translateStartEvent

```
input  : a fragment (A,G,s,$\xrightarrow{cf}$,read,write)
output: null or a Petri net representing the start event
1  if $s = \perp$ then
2      return $\perp$;
3  end
4  pn = new emptyPetriNetWithControlFlow();
5  for $outputSet \in o(s)$ do
6      startEventTransition = pn.createTransition().withLabel(s,outputSet);
7      for $(obj,state) \in outputSet$ do
8          dataObjectPlace = pn.createDataPlace().withLabel(obj[state]);
9          startEventTransition.addToPostset(dataObjectPlace);
10     end
11 end
12 return pn;
```

Translating process fragments. Each activity, gateway, and start event belongs to a fragment. We formalize a fragment by first translating individual elements (see Algorithms 1, 2, and 3) before connecting them according to the control flow.

Algorithm 4 shows the procedure for translating a whole fragment. After translating all control flow nodes (lines 3–5), the algorithm iterates over the sequence flow. It connects transitions representing the source node to the enablement place of the target node, where the target is either an activity or a gateway (lines 8–24). If the source node is a start event, the algorithm selects all transitions representing the event and adds the enablement place of its successor to the transitions' post-sets (lines 19–22). If the source node is an activity, it does the same for all the transitions representing the termination of the activity (lines 9–13). If it is a gateway, however, it takes the single transition representing the control flow arc that starts in the gateway and ends in the target (lines 14–18).

Algorithm 4: Function translateFragments

```
   input  : a set F of fragments
   output: a map fragmentNets, which maps fragments to their Petri net
           formalization
1  fragmentNets = new Map();
2  for $(A,G,s,\xrightarrow{cf},read,write) \in F$ do
3      activityNets = translateActivities(A,G,s,$\xrightarrow{cf}$,read,write);
4      gatewayNets = translateGateways(A,G,s,$\xrightarrow{cf}$,read,write);
5      startEventNet = translateStartEvent(A,G,s,$\xrightarrow{cf}$,read,write);
6      fragmentNet = combineNets(activityNets, gatewayNets, startEventNet);
7      for $(source,target) \in \xrightarrow{cf}$ do
8          enablementPlace =
           fragmentNet.findPlaceWhere(label=target[enabled]);
9          if $source \in A$ then
10             for transition ∈ fragmentNet.findTransitionWhere(label LIKE
               'terminate source*') do
11                 transition.addToPostSet(enablementPlace);
12             end
13         end
14         if $source \in G$ then
15             for transition ∈
               fragmentNet.findTransitionWhere(label=source,target) do
16                 transition.addToPostSet(enablementPlace);
17             end
18         end
19         if $source = s$ then
20             for transition ∈ fragmentNet.findTransitionWhere(label LIKE
               's*') do
21                 transition.addToPostSet(enablementPlace);
22             end
23         end
24     end
25     fragmentNets.put((A,G,s,$\xrightarrow{cf}$,read,write), fragmentNet);
26 end
27 return fragmentNets;
```

Figure 6 depicts the formalization of the fragment *medication*. In the fragment, the activity *choose medication* is followed by *administer prescription*. The Petri net contains four transitions for the start and termination of the activities, respectively. When the Petri nets for the individual activities are concatenated, data places with the same label are merged into one (line 15).[1] The exit place of the first activity's net is merged with the one of the latter.

See Fig. 6 for a formalization of the fragment **f4** medication. Its first activity, *choose medication*, requires a data object *patient file* in state *medication*. The Petri net has a transition `begin choose medication, {(pf,m)}` that consumes a respective token and binds the data object by producing a token on the binding place. Furthermore, it consumes a token from the control flow place `choose medication[enabled]`. Since *choose medication* is the first activity of the fragment, a token is also produced on the place to allow more instances to run concurrently. After the transition fired, a token on `choose medication[running]` reflects that the activity is currently executed. On termination (represented by a subsequent transition), the *patient file* is released, a *prescription* is *created*, and the control flow is progressed. The fragment continues with the activity *administer medication*.

Mapping the termination conditions. The set of termination conditions contains data conditions. Whenever one of the conditions is satisfied, the case can be closed. Similar to the start of an activity, we need to represent the different conditions by different transitions. For each elemental termination condition, we create one transition (Algorithm 5, line 5) and the corresponding data places in the transition's pre-set (lines 8–11). The pre-sets and post-sets of all these transitions have a common control flow place with the label `terminationConditions[enabled]` (lines 2 and 5) and `terminationConditions[fired]` (lines 3 and 6), respectively.

Algorithm 5: Function translateTerminationConditions

```
   input  : a set of termination conditions $\mathfrak{T}$
   output: a labeled Petri net pn with control flow
1  pn = new EmptyPetriNetWithControlFlow();
2  enablementCfPlace =
   pn.createCfPlace().withLabel(terminationConditions[enabled]);
3  firedCfPlace = pn.createCfPlace().withLabel(terminationConditions[fired]);
4  for $condition \in \mathfrak{T}$ do
5      transition =
       pn.createTransition().withLabel(terminationCondition,condition);
6      transition.addToPreset(enablementCfPlace);
7      transition.addToPostset(firedCfPlace);
8      for $(obj,state) \in condition$ do
9          inputPlace = pn.createDataPlace().withLabel(obj[state]);
10         transition.addToPreset(inputPlace);
11     end
12 end
13 return pn;
```

Mapping the case model. Given a case model, we translate all fragments and the termination condition (see Algorithm 6, lines 1 and 2). Next, we combine all retrieved Petri nets into one (line 3): all transitions and control flow places, including corresponding arcs, are added to the combined Petri net. Furthermore, we partition the data places of all Petri nets according to their label. For each partition, we add one place to the combined Petri net. This place has all the outgoing and incoming arcs of the places in the corresponding partition. In the resulting Petri net, there is exactly one place for each combination of a data object and its state. The transitions representing activities that consume or write the corresponding data object configuration access this place.

Next, we must add additional places and arcs to connect the start events to the fragments. First, we add a control flow place with label `i` (line 4). The place is added to the pre-

[1] The function `combinePetriNets` is like box calculus operations which combine smaller Petri nets to larger ones [6].

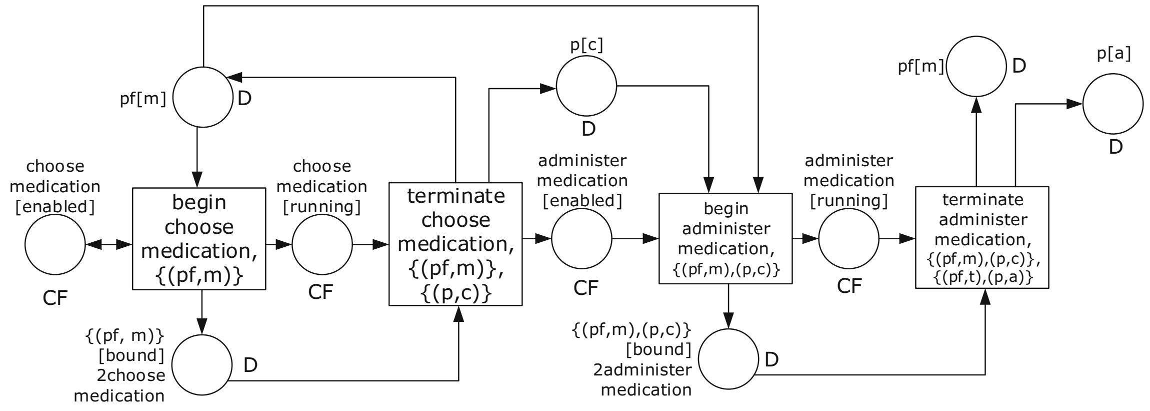

pf[m]
D
p[c]
D
pf[m]
D
p[a]
D
choose medication [enabled]
CF
begin choose medication, {(pf,m)}
choose medication [running]
CF
terminate choose medication, {(pf,m)}, {(p,c)}
administer medication [enabled]
CF
begin administer medication, {(pf,m),(p,c)}
administer medication [running]
CF
terminate administer medication, {(pf,m),(p,c)}, {(pf,t),(p,a)}
{(pf, m)} [bound] 2choose medication
D
{(pf,m),(p,c)} [bound] 2administer medication
D

Fig. 6 Petri net representation of the fragment *medication* is derived by sequentially combining the Petri nets for the activity *choose medication* and *administer medication*. We use the same abbreviations as in Fig. 5 and additionally *a* for *administered* and *t* for *treated*

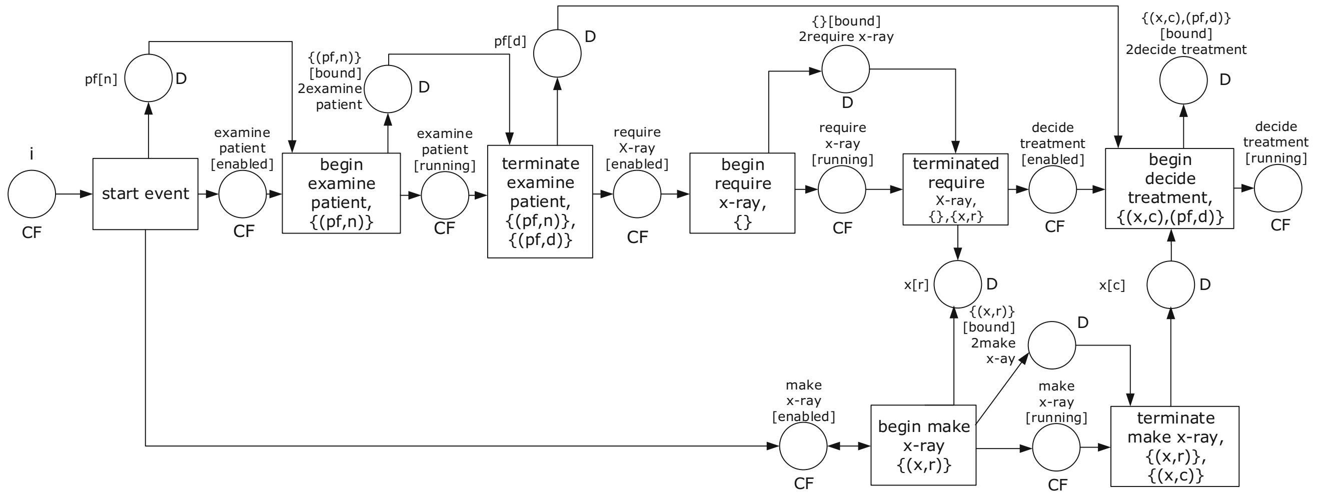

i
CF
start event
pf[n]
D
examine patient [enabled]
CF
{(pf,n)} [bound] 2examine patient
D
begin examine patient, {(pf,n)}
examine patient [running]
CF
pf[d]
D
terminate examine patient, {(pf,n)}, {(pf,d)}
require X-ray [enabled]
CF
begin require x-ray, {}
{}[bound] 2require x-ray
D
require x-ray [running]
CF
terminated require X-ray, {},{x,r}
decide treatment [enabled]
CF
{(x,c),(pf,d)} [bound] 2decide treatment
D
begin decide treatment, {(x,c),(pf,d)}
decide treatment [running]
CF
x[r]
D
{(x,r)} [bound] 2make x-ay
D
x[c]
D
make x-ray [enabled]
CF
begin make x-ray {(x,r)}
make x-ray [running]
CF
terminate make x-ray, {(x,r)}, {(x,c)}

Fig. 7 Petri net representation of parts of the fragment *diagnosis* and the fragment *x-ray* after processing them according to Algorithm 6

set of transitions representing a start event (lines 5 and 8). This place is the initial place. It holds one token in the initial marking. Furthermore, we must enable the fragments when the the start event fires. We take all the enablement control flow places for initial activities (line 6). Next, we add the places to the post-set of each transition representing a start event (lines 9–11).

Figure 7 depicts a part of the emergency handling case model formalized as a Petri net. It contains the start event; when the respective transition fires, the first activities of all fragments are enabled. Furthermore, it contains the fragment *x-ray* and parts of the fragment *diagnosis*. Both fragments can run concurrently. Due to mutual dependencies, it is necessary that they run in an interleaved manner: first the patient needs to be examined (fragment *diagnosis*), then an X-ray must be requested (fragment *diagnosis*), the X-ray must be made (fragment *x-ray*), before the diagnosis can continue (fragment *diagnosis*, activity *decide treatment*).

Algorithm 6: Mapping case models to Petri Nets

```
input  : a case model $M = (F, D, mathfrakT)$
output: a Petri net $N = (P, T, \to, \lambda, \sigma)$
1  fragmentNets = translateFragments(F);
2  terminationConditionNet = translateTerminationConditions(mathfrakT);
3  caseModelNet = combinePetriNets(fragmentNets, terminationConditionNet);
4  initialPlace = caseModelNet.createCfPlace().withLabel(i);
5  startEventTransitions = findStartEventTransitionsIn(caseModelNet);
6  enablementPlaces =
   findEnablementPlacesOfInitialActivitiesIn(caseModelNet);
7  for $transition \in startEventTransitions$ do
8      transition.addToPreset(initialPlace);
9      for $place \in enablementPlaces$ do
10         transition.addToPostset(place);
11     end
12 end
```

Adding Fragments to the case model. Case models can be adapted at run-time by adding new fragments. Adding a fragment does not require a full re-run of the formalization. We translate the new fragment separately (using Algorithm 4) from the remaining case model. The resulting Petri net can be combined with the existing model using the function combinePetriNets. This allows to adapt the formalization incrementally as new fragments are added to a case.

5 Compliance Checking Framework

In general, our frameworks follows a model checking approach to verify a case model against compliance rules (see Fig. 8). A model checker extracts a state space from a formal model and aligns it with a formal specification (i.e., given as temporal logic expressions). More specifically, we take a case model (see definitions in Sect. 3) and formalize its behavior using Petri nets (see mapping in Sect. 4). Furthermore, we specify compliance rules as temporal logic formulas (using CTL*) with respect to the Petri net. A proposition corresponds to a token in the net. A model checker extracts the Petri net's state space and verifies the formula. If the property does not hold, the model checker returns a counterexample in the form of a trace violating the property. In this section, we detail how the proposed framework uses this approach to both design-time compliance checking as well as run-time compliance checking.

5.1 Design-Time Compliance Checking

At design-time the model checking process can prove that a case model adheres to compliance rules. To do so, the complete behavior, this means all possible traces, must be considered. Therefore, the fully specified case model is checked. However, case models describe multi-variant processes and the state space may grow infeasible large or may even be unbounded.

Fragments can be executed arbitrarily often, in sequence or concurrently. However, the state space is only bound if every place of the Petri net is bound. This means there must be an upper limit for the number of data objects and fragment instances. Consequently, for any fragment, there can only be a finite number of instances per case. Furthermore, fragments creating new object instances can only be executed finitely often.

Some case models satisfy this property naturally: in order to start a new fragment instance certain data objects must be in specific states. The first activity of such a fragment may invalidate this condition. Hence, it disables the fragment so that no further instances can be spawned.

More often, however, parts of the model are unbound, or the knowledge limiting the number of instances is not reflected in the model. In such cases, the Petri net mapping from Sect. 4 cannot be used for model checking. We need to adapt the formal model in order to limit the state space to a finite number of states. To this end, our framework limits the number of instances for each fragment. A finite number of instances can only create a finite number of objects leading to an upper bound for the resulting Petri net.

For each fragment, we add a place with a fixed number of tokens. The transitions representing the respective fragment's first activity consume a token from the place. Whenever a new instance of the fragment is created, the first activity is executed and the number of tokens is decremented. If no token is left, the transitions cannot fire anymore; hence, no more fragment instances can be spawned. The resulting state space is finite.

Real-world processes usually accomplish a certain business goal. Therefore, cases should not run indefinitely. However, a rich set of process variants which may partly be unknown at run-time require a flexible approach such as frag-ment-based case management (fCM). Nevertheless, limiting the number of fragment instances is in most cases a reasonable assumption; while preventing infinite loops and data object creation, a sufficiently high bound still captures all possible execution orders. Our approach validates these orders and, thus, returns correct results despite the bound (see Sect. 6 for details).

5.2 Run-Time Compliance Checking

Usually, case models remain under-specified at design-time. Knowledge workers decide on the next activity to be executed (out of the set of enabled activities). Furthermore, fCM allows adapting the specification by adding new fragments to a running case. Both the decisions and adaptations made at run-time may impact the case's compliance. Therefore, compliance violations at design-time may be acceptable if the knowledge workers prevent them through intelligent decisions at run-time. At the same time, adding new fragments may introduce flaws, which are absent in the base model.

Compliance can be assessed at design-time, at run-time, or a posteriori. Run-time compliance monitors the state of running cases and makes statements about the cases' compliance in the past, present, and sometimes in the future. Our framework supports and distinguishes between two scenarios:

1. Run-Time Detection: The first scenario allows compliance violations during design-time, but we want to detect situations in which compliance violations become unavoidable at run-time.
2. Run-Time Extensions: In the second scenario, knowledge workers may extend the case model and potentially invalidate existing results from design-time compliance checking.

Run-Time Detection. Checking processes models can detect and prevent compliance violations. However, especially multi-variant processes may become rather complex when

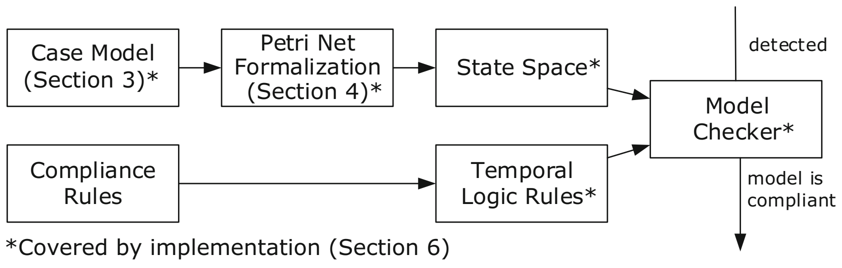


Fig. 8 High-level view on the compliance-checking framework. The process is specified in a case model, formalized as a Petri net, and verified against (temporal logic) compliance rules cf. [31])

compliance is incorporated at design-time. An under-specified model may allow both valid and invalid behavior. In this case, it is up to the user (i.e., the knowledge worker) to choose paths that do not lead to a compliance violation.

If multiple activities are enabled, knowledge workers decide on the next one. Thereby, they perform a set of decision tasks that drive the case. If the model allows compliance violations, the knowledge workers must keep the case compliant and choose appropriate next actions. Automated approaches can support the knowledge worker at run-time.

Compliance auditing can detect violations of running processes: past states are recorded and investigated with respect to compliance rules. Generally, it is desirable to prevent compliance violations or to detect them as early as possible to take appropriate counter measures. We apply a hybrid approach that uses the case model as well as state information of the running case. Therefore, we (i) formalize the model as a Petri net and (ii) induce the current state of the case, before (iii) we verify the compliance.

The model is formalized as described in Sect. 4. However, we do not include any tokens. Tokens are added in the second step; we consider the state of the running case: for each instance of a data object, we add a token to the respective place. Furthermore, we add tokens for enabled and running activities and gateways.

Considering the enabled activities, we can use a model checker to investigate whether certain actions lead to compliance violations at run-time. If, by performing a certain activity, a violation becomes unavoidable, the knowledge worker can abort the activity or adapt the process model to remain compliant. Consider a rule for the emergency ward that requires the report to be finalized before discharging the patient. Assuming that the fragments would allow a violation of the rule, the physicians would be informed when they start the activity “discharge patient” and the report is not in state final. If compliance violations are possible but avoidable, the potential future violation is presented to the knowledge worker (and can be avoided).

Run-Time Extensions. Case management allows knowledge workers to adapt the process at run-time. Regarding this, fCM allows adding new fragments to an already running case. Before conducting a surgery a detailed diagnosis is made, including an X-ray. However, the health of the patient might be in a critical state. A surgery may be necessary. A new fragment can be added that allows to conduct a surgery if two physicians independently decide on its necessity.

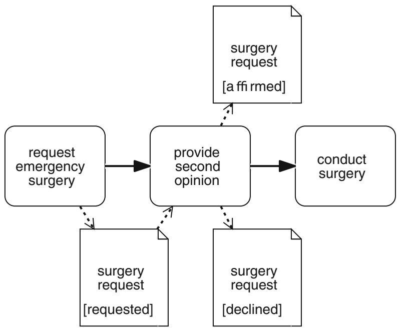

surgery request [affirmed]
request emergency surgery
provide second opinion
conduct surgery
surgery request [requested]
surgery request [declined]

Fig. 9 Extension to the emergency ward case model: The fragment allows to perform an emergency surgery if two physicians agree on it

Such an addition can change whether a case model is compliant. The example fragment in Fig. 9 shows the emergency surgery: if a physician reports an emergency, a second opinion is consulted before conducting the surgery. However, the fragment allows conducting the surgery no matter the second physicians’ opinion. This represents a compliance violation which does not exist in the base model.

In order to verify the adapted model, we need to formalize it, add the case’s state, and check it. From the Petri net formalism, we know that fragments are composed concurrently. In order to check the compliance for the adapted model, we (i) have to add the new fragment to the formalism and (ii) introduce the current state as a marking. There are two differences to the approach presented in the section “Run-Time Detection.” First, we obviously use the updated model rather than the original one. Secondly, it may be necessary to verify the model against rules that held for the original (at design-time), since introduced changes may affect the compliance of the case model.

We showed in Sect. 4 that case models are modular: fragments can be formalized independently of one another and the results can be composed into a larger Petri net. Hence, it is sufficient to formalize the newly created fragment and add to the existing formalization as (a concurrent sub-net). The addition will never restrict the behavior but may extend it.

This extension may lead to a compliance violation previously absent.

6 Proof of Concept and Evaluation

In this section, we present our proof of concept implementation and an evaluation. The evaluation is twofold: it consists of an application to a real world use case as well as an empirical analysis of our implementation's run-time performance. The first part evaluates the effectiveness. It demonstrates that our approach performs as expected. The second part shows that compliance checking can be performed within reasonable time and that our approach is, therefore, feasible.

6.1 Prototypical Implementation

The proposed formal behavioral model for fragment-based case management (fCM) is based on generating a Petri net for a given case model. This Petri net can then be used to check compliance of the originating model. We applied this approach by integrating a prototypical compliance checking component into an existing system for modeling and executing fCM models.

Architecture Overview. The existing fCM system comprises a modeler for fCM case models called *Gryphon*[2,3] and the case execution engine *Chimera*[4] [19] shown in the system architecture in Fig. 10. In the usual workflow, case models are designed in Gryphon. Once they are deemed complete, they are deployed to Chimera, where they can be executed. If a case model needs to be adapted, it can be changed in Gryphon and then re-deployed to Chimera. Following our presented approach of mapping fCM case models to Petri nets, we decided to use the *Low Level Analyzer (LoLA)* [48] as model checker for the generated Petri nets. LoLA can perform (among others) model checks on Petri nets using rules specified as linear temporal logic (LTL) and computation tree logic (CTL) formulae.

To extract and analyze the state space of fCM case models with LoLA, they need to be translated to Petri nets. We implemented the compliance checking logic—including a mapper translating fCM models to Petri nets—in the fCM execution engine Chimera, cf. system architecture in Fig. 10. Placing the translation and compliance checking component in the execution engine instead of the modeler offers the advantage that run-time state information of case instances can be accessed. Still, a user interface for the process designer to the compliance checker is integrated in the Gryphon modeler further presented in the next paragraph. Afterward, the

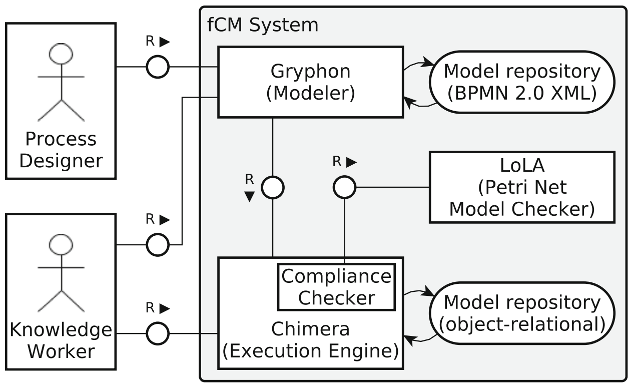

fCM System
Process Designer
Knowledge Worker
R ▶
R ▶
R ▶
R ▼
R ▶
Gryphon (Modeler)
Model repository (BPMN 2.0 XML)
LoLA (Petri Net Model Checker)
Compliance Checker
Chimera (Execution Engine)
Model repository (object-relational)

Fig. 10 The *fCM system* comprising a modeler, an execution engine and a model checker

execution of a compliance request is explained, and finally, the realization of checking run-time extensions is presented.

The project page[5] contains more information and a demonstrating screencast.

User interface. Our proof of concept implementation has a user interface in Gryphon that allows checking CTL* formulae (formalized compliance rules) against a designed case model. Compliance checking requests are forwarded to a linked and running Chimera instance. The overall result (yes/no/unknown) and a witness state or witness path, if present, are then displayed to the user in Gryphon.

Let us consider the exemplary constraints "Every patient will eventually be discharged" and "Per patient, a maximum of one Xray is created", which are formalized to the queries

```
F ({Discharge patient} = 1)
```

and

```
G ({Xray[created]} <= 1)
```

respectively.[6] The first property requires that a state satisfying the termination condition can always be reached, this property is also called *weak termination*. The second property is more domain specific. It limits the number of X-rays per patient, which is a reasonable safety measure to limit the exposure to radiation. When checking these two example constraints, Gryphon reports that the first formula is satisfied, while the second is not. For the first query, no witness path is produced, since it is always satisfied. For the second one, a witness path is shown that lists the corresponding transition of the *Make Xray* activity twice, once during the *Diagnosis* fragment, and once during the *Surgery* fragment. More

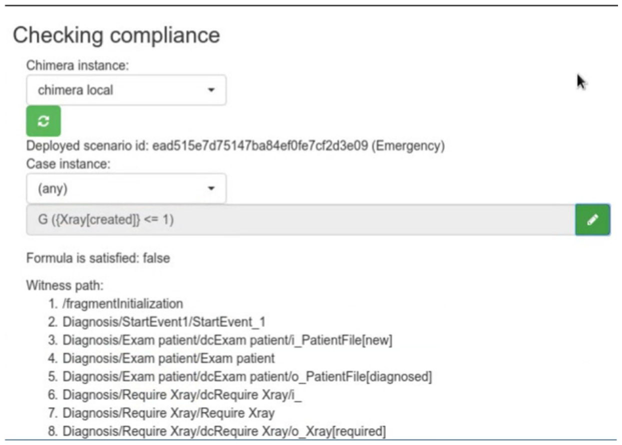
2 https://github.com/bptlab/gryphon.

3 Gryphon is based on Camunda's BPMN.io: https://bpmn.io/.

4 https://github.com/bptlab/chimera.

5 https://bptlab.github.io/ds2020-data-driven-case-management-compliance/.

6 The constraints are not raised from reality but reasonable for the given scenario.



Checking compliance

Chimera instance:

chimera local

Deployed scenario id: ead515e7d75147ba84ef0fe7cf2d3e09 (Emergency)

Case instance:

(any)

G ({Xray[created]} <= 1)

Formula is satisfied: false

Witness path:

1. /fragmentInitialization
2. Diagnosis/StartEvent1/StartEvent_1
3. Diagnosis/Exam patient/dcExam patient/i_PatientFile[new]
4. Diagnosis/Exam patient/Exam patient
5. Diagnosis/Exam patient/dcExam patient/o_PatientFile[diagnosed]
6. Diagnosis/Require Xray/dcRequire Xray/i_
7. Diagnosis/Require Xray/Require Xray
8. Diagnosis/Require Xray/dcRequire Xray/o_Xray[required]

Fig. 11 Cropped screenshot of the Gryphon modeler showing the result of a compliance check

specifically, also the selected pre- and post-condition sets are listed with the corresponding consumed and produced data objects in their respective states as shown in Fig. 11. Thus, the two tokens on *Xray[created]* can be found in the witness path.

Execution of a compliance request. Compliance checking requests sent from Gryphon to Chimera contain a CTL* formula, individually serialized fragments of the case model, and optionally a case instance ID if run-time compliance checking is selected. When Chimera receives such a compliance checking request, multiple tasks are performed. First, the referenced case model is retrieved from the model repository and translated to a Petri net, using the aforementioned mapping. Secondly, the CTL* formula is rewritten, so that identifiers of fCM model elements and their specific states are identified and internally replaced with the identifiers of the corresponding places/transitions in the Petri net. Third, a marking is generated for the Petri net: for design-time compliance checking, an initial marking is generated, while for run-time compliance checking, the run-time state of the referenced case instance is used to generate a marking that reflects the current execution state. The Petri net, its marking, and the rewritten compliance formula are then sent to LoLA to perform the actual model checking step. Upon completion, the result is parsed. If a witness state/path is provided by LoLA, the referenced identifiers (pointing to places or transitions in the Petri net) are translated back to meaningful identifiers in the case model.

Run-time extensions. A compliance request of Gryphon to Chimera contains the serialized case model's fragments. The fragments are used by Chimera to compare them to the fragments of the deployed case model. If additional fragments are encountered in the request, a check for a run-time extension is executed. First, the deployed case model is translated to a Petri net and a mapping for the current case instance state is generated in the fashion of run-time compliance checking. Next, a new sub-net is generated for the newly added fragment, and an initial marking is generated. Since both fCM models and the translated Petri nets are modular in nature, the sub-net for the newly added fragment is integrated into the Petri net. In contrast to run-time compliance checking of a deployed model, we consider two different sources, the deployed model and the adapted model, which we merge into a single formal model rather than considering only the deployed model. This way, it is possible to check if adding a new fragment to a running case instance violates a given compliance rule.

6.2 Evaluation

Model-based compliance checking promises to detect violations at design-time. This is desirable for companies since the model can be corrected before it is implemented. In particular, process models with a flexible behavior, such as fCM, may contain compliance violations that remain unnoticed by humans. Therefore, semi-automatic compliance checking presents a value to companies. In this section, we evaluate the feasibility and efficiency of our approach by (i) applying it to a computer-aided translation use case and (ii) by empirically evaluating the performance of our implementation.

6.2.1 Application to a Document Translation Service Use Case

We selected a professional translation service use case that uses computer-aided translation (CAT) tools. Information systems are widely applied in translation management because they improve the translators' efficiency and the translations' quality. However, required functionality is often distributed among multiple services [55]. The translator can use and manually connect these tools as desired. Production case management can be used to efficiently coordinate work that involves the different systems while maintaining the flexibility required by the translators.

Evaluation goal and method. By applying our compliance checking approach to a real-world use case, we aim to evaluate the effectiveness. Therefore, we elicit and design a process model, verify the model against a set of compliance rules, and iterate the model until it is compliant.

We consulted a booklet by European commission [55] to get a first impression of the use case. Afterward, we conducted interviews with a translator to derive a case model describing the CAT process. The translator's experience is based on her work at a french translation agency and a German online retailer using the CAT tool SDL Trados. We validated the model with the translator in a separate session. Next, we checked whether the model is compliant. Therefore, we interviewed the expert again to elicit a set of compliance rules related to the process logic. The set is limited to a few rules relevant for the use case to illustrate the effectiveness of our approach and might be incomplete. The rules have been formalized manually. Surprisingly, our initial model—although validated—was not compliant. In two additional iterations, the model was adapted to satisfy all elicited compliance rules. The final model was again validated with the translator.

Translation is knowledge-intensive work mostly driven by the translator's experience. However, there are crucial constraints and desirable guidelines to deliver good translations. In an interview, we gathered a list of compliance rules that should be satisfied by each translation. We then translated the rules to temporal logic formulas using LTL. All rules and their translation are listed in Table 1 in the column "rules".

We verified the process model against each rule limiting each fragment to five instances and measured the performance (as end-to-end latency). Note that in this case, five instances are enough to capture all possible sequences of activities. In fact, most fragments will be executed at most once. Increasing the limit, e.g., to 100, has no significant impact on the performance. However, this is due to the peculiarities of this case model. For the impact of the limit on different case models see Sect. 6.2.2.

Resulting fCM model. Figure 12 depicts the fragments of the case model. For every job, the translator decides whether to accept or reject it (**f1**). Every accepted job needs to be translated. The translator can optionally query the translation memory (TM) to automatically insert known translations for text blocks (**f3**). Next, the job is translated to one or multiple segments by executing the fragment *segmentation* (**f2**). The TM is a data base that stores past translation and that is used to (a) reduce the work necessary to translate a new text and to (b) stay consistent with regard to past translations. Regardless whether **f3** has been executed or not, the translation can be continued. The translation is started **f4**, segments are translated in one or multiple steps **f5**, before the translation is finalized **f4**. Once the translation is complete, the translator must update the TM. This can be done semi-automatically, by 1. aligning the translation to the original text (**f6**), 2. verifying the alignment (**f6**), and 3. using the alignment to update the TM (**f7**). Alternatively, the translator can update the TM manually (**f8**).

Compliance checking results. Since the Petri net is merely an internal representation of the system, we do not provide a translation for the CAT example here. A user can only influence the case model and limit the number of fragment instances. The performance measures for compliance checking the CAT example are reported in the column *duration* of Table 1. We adapted the model twice to satisfy all compliance rules.

In the initial model (v1),[7] the manual translation consisted only of the activity *translate segment* and not of fragment **f4**. Furthermore, each segment was a separate data object created by repeatedly executing a fragment *create segment*. Rule R2 requires that *translate segment* once for each segment (as often as *create segment*). However, fCM has no mechanism to quantify data objects. Each condition is evaluated on a single object. Thus, it was possible to execute *translate segment* only once even in presence of multiple segments. In version v2, all segments were merged by a single object created in a single step and translated in one or multiple steps. The translator verified, that the segments are created in a single activity whose result cannot be used before all segments have been created. The activity, however, can be paused and resumed.

The new version v2, contained a new flaw: The translation was still represented by a single activity that can be executed repeatedly to start, continue, and finalize the translation. This was done by using different input and output-sets. However, fCM supports no mechanism to indicate that eventually a certain output set is chosen. Consequently, traces in which the translation was never finalized were possible. This violates rule R2. We resolved the issue by splitting the translation into two fragments and three activities. Fragment *f5* can be executed repeatedly to translate segments, but the execution is framed by the activities *start translation* and *finalize translation*, which are executed exactly once in each case. While *translate segment* can still be executed arbitrary often, we set an upper limit for the number of fragment instances (e.g., 5). Not later than this number is reached, *finalize translation* can and will be executed. The final model (see Fig. 12) complies to all rules.

6.2.2 Empirical Evaluation of Performance

One of the major concerns when it comes to model-based compliance checking is feasibility [54]. Model checking for Petri nets is, in general, an EXPSPACE problem [14]. The concurrent execution and flexibility of fragment-based case models leads to an exponential growth of the state space, called *state explosion*. Therefore, the questions whether our approach's performance is feasible or not arises. In the following, we provide empirical results from the verification of case models of different complexity. We investigate the case model, the size of the resulting Petri nets, the complexity of its behavior (size of the state space), as well as the performance of checking compliance.

For our empirical performance evaluation, we use the example in Fig. 3 as well as two extensions of the model. With the extension, physicians can adapt and change the *report* arbitrary often before it is considered to be final. This

[7] Initial model at https://bptlab.github.io/ds2020-data-driven-case-management-compliance/#the-cat-example-model.

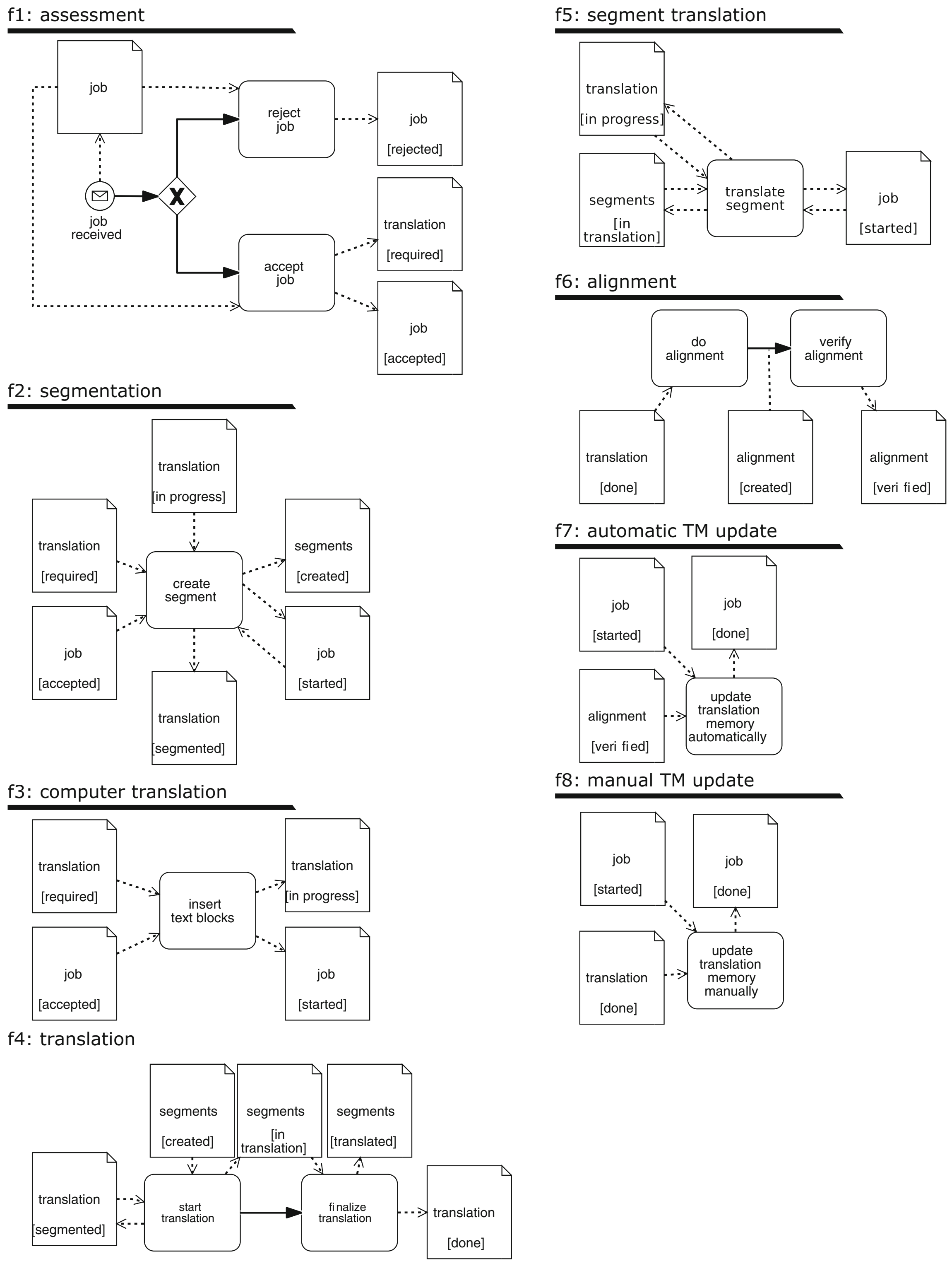


Fig. 12 Fragments of a case model describing the process of computer-aided translation

Table 1 Compliance rules for the CAT example in both natural language and temporal logic as well as performance measures; the number of instance for each fragment is limited to 5

No.	Rule	Time (in ms)
R1	If a job gets rejected, no further actions are possible	232
	G((job[rejected]>0)→**G**(accept job[running]=0 AND translate segment[running]=0...))	
R2	A case cannot be closed before all segments are translated	164
	G(**X**(terminationConditions[fired]>0)→(segments[in translation]=0 AND segment[created] = 0))	
R3	The TM cannot be updated automatically if the alignment is unverified	129
	G((alignment[created]>0)→**X**(update TM automatically[running]=0))	
R4	Segment can only be created until the translation is started	176
	G((translation[in progress]>0)→(create segment[running]=0))	
R5	If a case gets accepted, the TM must eventually be updated	145
	G((job[accepted]>0)→**F**(update TM automatically[running]>0 OR update TM manually[running]>0))	

The elicited rules are incomplete and limited to those related to the process model

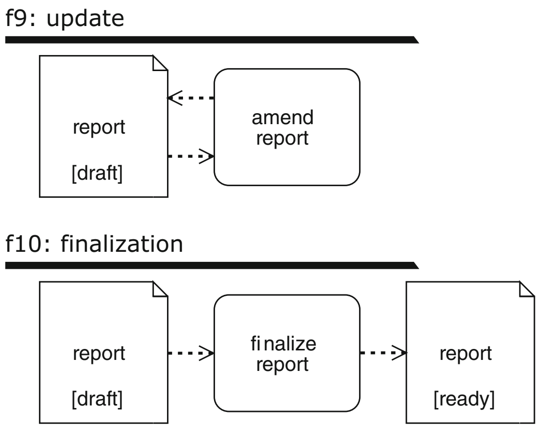


Fig. 13 Additional fragments for the emergency ward process

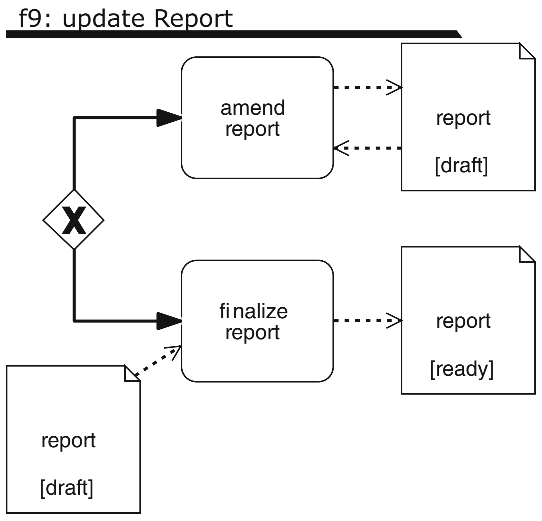


Fig. 14 Additional fragment offering the decision to amend or finalize the report

potential infinite loop increases the complexity of the model. We add two activities to manifest the change: In the first version, we add a single fragment (see Fig. 14) that starts with a decision (exclusive gateway) followed by the two activities on alternative branches. In the second version, we add two separate *fragments* (see Fig. 13).

The complexity of the case model variants and their Petri net formalization is given in Table 2. We assess the complexity by looking at the structural elements. For the case model, we count the fragments, activities, gateways, data classes/data objects, and data states. For the Petri net, we use the number of places and transitions. However, rather than looking at the structural complexity, we are interested in the behavior.

Therefore, we compare their behavioral complexity in the size of state space and its impact on compliance checking. For each variant, we check whether each case will eventually terminate:

`F(terminationConditions[fired] > 0).`

Table 2 Complexity of the case model and Petri nets given as the number of fragments, activities, gateways, data objects, data states, as well as places and transitions (from left to right)

Case model variant	$\|F\|$	$\|A\|$	$\|G\|$	$\|C\|$	Q_C	$\|P\|$	$\|T\|$
Original	6	12	0	4	13	28	54
Decision	7	14	1	4	14	36	62
Fragments	8	14	0	4	14	32	61

We measure the end-to-end/roundtrip time for the variants and different bounds for the number of fragment instances. These tests were performed on a consumer laptop using an Intel Core i5 430M processor and 8 GiB of RAM, running a 64-bit Debian 10 with kernel 4.9.0. When measuring the

Table 3 Number of reachable states and performance measurements for different variants and limits

Case model	Limit	$\|States\|$	Satisfied	Roundtrip	Model checking
Original (Fig. 3)	1	43	✓	34 ms	31 ms
Decision (Fig. 14)	1	87	×	157 ms	29 ms
Fragments (Fig. 13)	1	78	✓	31 ms	35 ms
Original (Fig. 3)	5	467	✓	127 ms	115 ms
Decision (Fig. 14)	5	8515	×	27 ms	31 ms
Fragments (Fig. 13)	5	3010	✓	836 ms	785 ms
Original (Fig. 3)	10	2167	✓	3334 ms	2853 ms
Decision (Fig. 14)	10	132, 540	×	29 ms	30 ms
Fragments (Fig. 13)	10	25, 305	–	Aborted	62,116 ms
Original (Fig. 3)	20	12, 317	–	Aborted	145,291 ms
Decision (Fig. 14)	20	2, 729, 665	×	28 ms	34 ms
Fragments (Fig. 13)	20	268, 795	–	Aborted	Out of memory

roundtrip time,[8] we configured LoLA to cancel after 60 seconds to prevent it from exhausting available resources (main memory). Additionally to the roundtrip, we measured the time required for the compliance checking without a time limit (separate run).

6.3 Discussion

In our evaluation, the performance of our approach ranges from around 30 ms to cases which cannot be verified at all. The result depends on the size of the state space that is investigated, which in turn depends on the complexity of the case model and on the assumptions made. While models that allow few repetitions and little concurrency of fragments (Sect. 6.2.1) can be verified with relaxed assumption, i.e., a fixed but high number of fragment instances, others are complex and require strict assumptions (Sect. 6.2.2).

In most cases, however, even a low boundary of allowed fragment instances—strong assumption—still leads to the same result as a relaxed assumption. Any limit, be it high or low, can change the result of the compliance checking. Looking at the emergency example that may update or finalize the report based on a decision (using an exclusive gateway), we see that it may not reach a termination condition. This happens if the report is only updated but never finalized. However, assuming the Petri net is fair,[9] each case will eventually terminate properly if no other assumptions, such as limiting the instances, are made. This cannot be assessed by the model checker. Thus, our assumption limiting the number of fragment instances can invalidate the result of the compliance checking, but will lead to a correct result in most cases.

[8] Roundtrip time: how long does it take from sending a request in the frontend to receiving the response?

[9] The fairness property states that every transition that can fire will eventually fire.

We evaluated the usefulness by applying our approach to the real world case of a computer aided-transition (Sect. 6.2.1). This evaluation was limited to a single use case. Still, the application demonstrates that fCM models may contain subtle errors, even though we, experts in fCM modeling, created the models. Our approach helped us to design a model conforming to all compliance rules.

However, the model checker may run out of memory for very large and highly concurrent models. Additionally, an organization usually has a set of compliance rules. While checking an individual rule is fast, evaluating many rules may be too slow for an interactive approach. Thus, our approach may not be applicable to all case models and all organizations. However, adaptations are possible, such as an asynchronous batch mode testing a set of rules and reporting the result.

In general, our model checking approach can handle the complexity of real world case models under certain assumptions. The interactive mode allows verifying rules and adapt the model right away if necessary. This proves to be effective in detecting and correcting errors in the model.

7 Conclusion

Traditional compliance checking allows companies to verify processes against compliance rules. However, they lack support for flexible processes. We presented an approach to check flexible processes modeled using the fragment-based case management approach. Our work allows checking process specifications and running instances against compliance rules.

Our approach enables organizations to model flexible processes in an activity-centric and mostly imperative manner, but at the same time, assert its integrity with declarative compliance rules. This is desirable since processes are commonly specified in activity-centric models (e.g., using the de facto industry standard BPMN) while being subject to declarative business rules [47]. Therefore, we present a Petri net formalization for case models, which extends existing formalization for process models. Using a model checker and optionally run-time information, users can verify their case models against rules.

A single fragment in fragment-based case management (fCM) is similar to a single process model in BPMN. This allowed us to build on existing compliance checking methods [2]. It also allows to extend our approach to handle a set of BPMN process models that are connected through shared data or communication.

However, our approach has some limitations. For one, it does not support users eliciting, modeling, and managing compliance rules. This limitations can be resolved by integrating our framework in a compliance management application. Furthermore, the rules are limited to CTL* rules regarding the order of activities, the running state of activities, and data object states. Many compliance, however, require detailed information about time and data attributes, which we do not support since they are not included in the model. Other approaches show how information about data attributes can be incorporated in processes models, e.g., by defining pre- and post-conditions of activities [8], constraints for states [41], or decisions [17,29]. However, including attribute-level information would increase the size of the state space drastically.

Model checking is a computationally expensive task. Although not encountered during evaluation, the state space of case models can grow to an infeasible size, especially if it contains many concurrent fragments and despite the counter measure presented in this paper. Parallelization and state-space reduction technique may improve the model verification further. Furthermore, model checkers usually report a violated property through a counter example or witness path. However, there may be more than one explanation for a violation. LoLA, the model checker used in our framework, is no exception: only one explanation is provided [48]. This may lead to a situation in which a violation is resolved but the property may still not hold. Iterative verification is necessary but may be tedious.

Finally, we tested the effectiveness of our approach with the help of a use case, but so far we did not test the usability of our approach and tool. In this regard, the approach requires further improvement: visual languages for compliance rules should be incorporated [4,30] and the overall user experience should be improved. After such improvements, future work may evaluate the usefulness, e.g., by using the framework presented in [11]. Furthermore, we focused on single case models and isolated cases. However, many knowledge-intensive processes are coupled: they share data, are subject to global constraints, or show batch processing behavior. These aspects have been subject to research investigating instance-spanning [15,33] constraints and batch processing [44]. In future work, we plan to extend our approach to handle multiple cases including shared data objects and batch behavior.

Funding Open Access funding enabled and organized by Projekt DEAL.

References

1. Awad A, Decker G, Weske M (2008) Efficient compliance checking using bpmn-q and temporal logic. In: Dumas M, Reichert M, Shan MC (eds) Business process management. Springer, Berlin, pp 326–341. https://doi.org/10.1007/978-3-540-85758-7_24
2. Awad A, Sakr S (2012) On efficient processing of BPMN-q queries. Comput Ind 63:867–881. https://doi.org/10.1016/j.compind.2012.06.002
3. Awad A, Weidlich M, Weske M (2009) Specification, verification and explanation of violation for data aware compliance rules. In: Service-Oriented Computing, 7th International Joint Conference, ICSOC-ServiceWave 2009, Stockholm, Sweden, November 24–27, 2009. Proceedings, pp 500–515. https://doi.org/10.1007/978-3-642-10383-4_37
4. Awad A, Weidlich M, Weske M (2011) Visually specifying compliance rules and explaining their violations for business processes. J Vis Lang Comput 22:30–55. https://doi.org/10.1016/j.jvlc.2010.11.002
5. Belardinelli F, Lomuscio A, Patrizi F (2012) Verification of GSM-based artifact-centric systems through finite abstraction. In: Service-oriented computing—10th international conference, ICSOC 2012, Shanghai, China, November 12–15, 2012. Proceedings, pp 17–31. https://doi.org/10.1007/978-3-642-34321-6_2
6. Best E, Devillers RR, Hall JG (1992) The box calculus: a new causal algebra with multi-label communication. In: Advances in Petri Nets 1992, The DEMON Project. Springer, Berlin, pp 21–69. https://doi.org/10.1007/3-540-55610-9_167
7. Borrego D, Barba I (2014) Conformance checking and diagnosis for declarative business process models in data-aware scenarios. Expert Syst Appl 41(11):5340–5352. https://doi.org/10.1016/j.eswa.2014.03.010
8. Borrego D, Eshuis R, López MTG, Gasca RM (2013) Diagnosing correctness of semantic workflow models. Data Knowl Eng 87:167–184. https://doi.org/10.1016/j.datak.2013.04.008
9. Burattin A, Maggi FM, Sperduti A (2016) Conformance checking based on multi-perspective declarative process models. Expert Syst Appl 65:194–211. https://doi.org/10.1016/j.eswa.2016.08.040
10. Combi C, Oliboni B, Weske M, Zerbato F (2018) Conceptual modeling of processes and data: connecting different perspectives. In: Conceptual modeling—37th International Conference, ER 2018,

Xi'an, China, October 22–25, 2018, Proceedings, pp 236–250. https://doi.org/10.1007/978-3-030-00847-5_18

11. Davis FD (1989) Perceived usefulness, perceived ease of use, and user acceptance of information technology. MIS Q 13(3):319–340
12. Di Ciccio C, Marrella A, Russo A (2015) Knowledge-intensive processes: characteristics, requirements and analysis of contemporary approaches. J Data Semant 4(1):29–57
13. Dijkman RM, Dumas M, Ouyang C (2008) Semantics and analysis of business process models in BPMN. Inf Softw Technol 50(12):1281–1294
14. Esparza J (1996) Decidability and complexity of petri net problems—an introduction. In: Lectures on Petri Nets I: Basic Models, Advances in Petri Nets, the volumes are based on the Advanced Course on Petri Nets, held in Dagstuhl, September 1996, pp 374–428. https://doi.org/10.1007/3-540-65306-6_20
15. Fdhila W, Gall M, Rinderle-Ma S, Mangler J, Indiono C (2016) Classification and formalization of instance-spanning constraints in process-driven applications. In: Business process management—4th international conference, BPM 2016, Rio de Janeiro, Brazil, September 18–22, 2016. Proceedings, pp. 348–364. https://doi.org/10.1007/978-3-319-45348-4_20
16. Gonzalez P, Griesmayer A, Lomuscio A (2012) Verifying GSM-based business artifacts. In: 2012 IEEE 19th international conference on web services, Honolulu, HI, USA, June 24–29, 2012, pp. 25–32. https://doi.org/10.1109/ICWS.2012.31
17. Haarmann S, Batoulis K, Weske M (2018) Compliance checking for decision-aware process models. In: Business process management workshops—BPM 2018 international workshops, Sydney, NSW, Australia, September 9–14, 2018, Revised Papers, pp 494–506. https://doi.org/10.1007/978-3-030-11641-5_39
18. Haarmann S, Batoulis K, Weske M (2019) Compliance checking for decision-aware process models. In: Business process management workshops. Springer, pp 494–506. https://doi.org/10.1007/978-3-030-11641-5_39
19. Haarmann S, Podlesny N, Hewelt M, Meyer A, Weske M (2015) Production case management: a prototypical process engine to execute flexible business processes. In: BPM (Demos), pp 110–114
20. Hashmi M, Governatori G, Lam HP, Wynn MT (2018) Are we done with business process compliance: state of the art and challenges ahead. Knowl Inform Syst 1–55
21. van Hee KM, Sidorova N, van der Werf JMEM (2013) Business process modeling using petri nets. Trans Petri Nets Other Model Concurr 7:116–161
22. Hewelt M, Weske M (2016) A hybrid approach for flexible case modeling and execution. In: Lecture notes in business information processing. Springer, Berlin, pp 38–54. https://doi.org/10.1007/978-3-319-45468-9_3
23. Hewelt M, Wolff F, Mandal S, Pufahl L, Weske M (2018) Towards a methodology for case model elicitation. In: Enterprise, business-process and information systems modeling. Springer, pp 181–195
24. Hildebrandt TT, Mukkamala RR (2010) Declarative event-based workflow as distributed dynamic condition response graphs. In: Proceedings third workshop on programming language approaches to concurrency and communication-cEntric Software, PLACES 2010, Paphos, Cyprus, 21st March 2010, pp 59–73. https://doi.org/10.4204/EPTCS.69.5
25. Hoare CAR (1978) Communicating sequential processes. Commun ACM 21(8):666–677
26. Holfter A, Haarmann S, Pufahl L, Weske M (2019) Checking compliance in data-driven case management. In: Business process management workshops—BPM 2019 international workshops, Vienna, Austria, September 1–6, 2019, Revised Selected Papers, pp 400–411. https://doi.org/10.1007/978-3-030-37453-2_33
27. Hull R, Damaggio E, Fournier F, Gupta M, III FFTH, Hobson S, Linehan MH, Maradugu S, Nigam A, Sukaviriya P, Vaculín R (2010) Introducing the guard-stage-milestone approach for specifying business entity lifecycles. In: Web services and formal methods—7th international workshop, WS-FM 2010, Hoboken, NJ, USA, September 16–17, 2010. Revised Selected Papers, pp 1–24. https://doi.org/10.1007/978-3-642-19589-1_1
28. Johnson RE (1997) Frameworks = (components + patterns). Commun ACM 40(10):39–42. https://doi.org/10.1145/262793.262799
29. Knuplesch D, Ly LT, Rinderle-Ma S, Pfeifer H, Dadam P (2010) On enabling data-aware compliance checking of business process models. In: Conceptual modeling—ER 2010, 29th international conference on conceptual modeling, Vancouver, BC, Canada, November 1–4, 2010. Proceedings, pp 332–346. https://doi.org/10.1007/978-3-642-16373-9_24
30. Knuplesch D, Reichert M, Ly LT, Kumar A, Rinderle-Ma S (2013) Visual modeling of business process compliance rules with the support of multiple perspectives. In: International conference on conceptual modeling. Springer, pp 106–120
31. Kunze M, Weske M (2016) Behavioural models–from modelling finite automata to analysing business processes. Springer, Berlin
32. Künzle V, Reichert M (2011) Philharmonicflows: towards a framework for object-aware process management. J Softw Maint 23(4):205–244. https://doi.org/10.1002/smr.524
33. Leitner M, Mangler J, Rinderle-Ma S (2012) Definition and enactment of instance-spanning process constraints. In: Web information systems engineering—WISE 2012—13th international conference, Paphos, Cyprus, November 28–30, 2012. Proceedings, pp 652–658. https://doi.org/10.1007/978-3-642-35063-4_49
34. Li Y, Deutsch A, Vianu V (2017) VERIFAS: a practical verifier for artifact systems. Proc VLDB Endow 11(3):283–296
35. Ly LT, Knuplesch D, Rinderle-Ma S, Göser K, Pfeifer H, Reichert M, Dadam P (2010) Seaflows toolset—compliance verification made easy for process-aware information systems. In: Information systems evolution—CAiSE Forum 2010, Hammamet, Tunisia, June 7–9, 2010, Selected Extended Papers, pp 76–91. https://doi.org/10.1007/978-3-642-17722-4_6
36. Ly LT, Rinderle-Ma S, Knuplesch D, Dadam P (2011) Monitoring business process compliance using compliance rule graphs. In: On the move to meaningful internet systems: OTM 2011—confederated international conferences: CoopIS, DOA-SVI, and ODBASE 2011, Hersonissos, Crete, Greece, October 17–21, 2011, Proceedings, Part I, pp 82–99. https://doi.org/10.1007/978-3-642-25109-2_7
37. Meyer A, Pufahl L, Fahland D, Weske M (2013) Modeling and enacting complex data dependencies in business processes. In: Daniel F, Wang J, Weber B (eds) Business process management. Springer, Berlin, pp 171–186
38. Milner R (1999) Communicating and mobile systems: the pi-calculus. Cambridge University Press, Cambridge
39. Object Management Group (OMG) (2014) Business Process Model and Notation (BPMN). OMG Document Number formal/13-12-09. Version 2.0.2
40. (2017) Object Management Group (OMG): Case Management Model and Notation (CMMN). OMG Document Number formal/16-12-01. https://www.omg.org/spec/CMMN/About-CMMN/. Version 1.1
41. Pérez-Álvarez JM, López MTG, Eshuis R, Montali M, Gasca RM (2020) Verifying the manipulation of data objects according to business process and data models. Knowl Inf Syst 62(7):2653–2683. https://doi.org/10.1007/s10115-019-01431-5
42. Pesic M, Schonenberg H, van der Aalst W (2007) DECLARE: full support for loosely-structured processes. In: Proceedings of the 11th IEEE international enterprise distributed object computing conference, p 287. IEEE Computer Society, Washington, DC, USA. http://portal.acm.org/citation.cfm?id=1317532.1318056
43. Pommereau F (2009) Algebras of coloured Petri nets and their applications to modelling and verification. Habilitation à diriger des

recherches, Université de Paris-Est/Créteil. https://hal.archives-ouvertes.fr/tel-02309973
44. Pufahl L, Weske M (2019) Batch activity: enhancing business process modeling and enactment with batch processing. Computing 101(12):1909–1933
45. Reisig W (2011) Petri Nets. Springer, Berlin. https://www.ebook.de/de/product/19303359/wolfgang_reisig_petri_nets.html
46. Sackmann S, Kuehnel S, Seyffarth T (2018) Using business process compliance approaches for compliance management with regard to digitization: evidence from a systematic literature review. In: International conference on business process management. Springer, pp 409–425
47. Santoro FM, Slaats T, Hildebrandt TT, Baião FA (2019) Dcr-kipn a hybrid modeling approach for knowledge-intensive processes. In: Conceptual modeling—38th international conference, ER 2019, Salvador, Brazil, November 4–7, 2019, Proceedings, pp 153–161. https://doi.org/10.1007/978-3-030-33223-5_13
48. Schmidt K (2000) Lola a low level analyser. In: Nielsen M, Simpson D (eds) Application and theory of Petri Nets 2000. Springer, Berlin, pp 465–474
49. Semmelrodt F, Knuplesch D, Reichert M (2014) Modeling the resource perspective of business process compliance rules with the extended compliance rule graph. In: Enterprise, business-process and information systems modeling—15th international conference, BPMDS 2014, 19th international conference, EMMSAD 2014, Held at CAiSE 2014, Thessaloniki, Greece, June 16–17, 2014. Proceedings, pp 48–63. https://doi.org/10.1007/978-3-662-43745-2_4
50. Slaats T, Mukkamala RR, Hildebrandt T, Marquard M (2013) Exformatics declarative case management workflows as DCR graphs. In: Business process management. Springer, pp 339–354
51. Solomakhin D, Montali M, Tessaris S, De Masellis R (2013) Verification of artifact-centric systems: decidability and modeling issues. In: Service-oriented computing—11th international conference, ICSOC 2013, Berlin, Germany, December 2–5, 2013, Proceedings, pp 252–266. https://doi.org/10.1007/978-3-642-45005-1_18
52. Sporleder T (2016) Fragment-based case management: specification and translational semantics. Master's thesis, Hasso Plattner Institute, University of Potsdam, Germany
53. Swenson KD (2012) Case management: contrasting production vs. adaptive. How knowledge workers get things done, pp 109–116
54. Tosatto SC, Governatori G, van Beest N (2019) Checking regulatory compliance: will we live to see it? In: Business process management—17th international conference, BPM 2019, Vienna, Austria, September 1–6, 2019, Proceedings, pp 119–138. https://doi.org/10.1007/978-3-030-26619-6_10
55. European Commission for Translation, D.G.: Translation tools and workflow
56. Weske M (2019) Business process management—concepts, languages, architectures, 3rd edn. Springer, Berlin. https://doi.org/10.1007/978-3-662-59432-2