

A Hybrid Approach for Flexible Case Modeling and Execution

Marcin Hewelt^(✉) and Mathias Weske

Hasso Plattner Institute Potsdam, Potsdam, Germany
{marcin.hewelt,mathias.weske}@hpi.de

Abstract. While the business process management community has concentrated on modelling and executing business processes with a known structure, support for processes with a high degree of variability performed by knowledge workers is still not satisfactory. A promising approach to overcome this deficiency is case management. Despite of the work done in the area of case management in recent years, there is no accepted case handling formalism that features a well defined semantics. This paper introduces a novel approach to case management, which is based on dynamically combining process fragments as required by knowledge workers. An operational semantics defines the meaning of case models in detail, using states of data objects and enablement conditions of process fragments.

Keywords: Case management · Business process management

1 Introduction

Business process management concepts and techniques have been successfully applied in a variety of domains to document, analyze, automate and optimize business processes. Processes with a predefined structure are well supported by today's technology. However, this is not true for processes with a high degree of variability, which are conducted by knowledge workers. As a result, data-driven and goal-oriented business processes with a high degree of variability are not well supported.

Increasing the flexibility of business processes has been one of the main drivers in the development of the BPM field, for instance in areas like flexible process management, process variants, declarative and object-centric approaches [12]. Based on these works, the area of case management centers around the concept of cases and knowledge workers [2]. Case Management is not a new concept, but IT support for knowledge work is still limited to specific domains and implemented in an ad-hoc manner.

Case management received attention in the industry [14], however, only few research publications deal with this topic. According to the literature review of Hauder et al. [6] a “solution that aims to support knowledge workers needs to balance between structured processes for repetitive aspects of knowledge work

and unstructured processes.” While case management is an important approach that complements traditional process management, there is no agreed operational semantics for case management. Therefore, Hauder et al. [6] identify the proposal of a case management theory as one of the key challenges.

In this paper we introduce a case management approach, based on [8] that provides an operational execution semantics for cases. Case models are specified by a number of process fragments, which are structured pieces of work that are dynamically combined during case execution based on data objects and their states. At runtime this results in a multitude of valid execution paths from case instantiation to case termination, suiting the flexible nature of knowledge work.

The rest of this paper is structured as follows. We review related work in Sect. 2. The conceptual framework for case management is presented in Sect. 3, followed by the operational semantics in Sect. 4. We discuss our approach in Sect. 5 and then conclude.

2 Related Work

The case handling approach [2] observes that classical WfMS are too rigid for knowledge workers and relaxes the control-flow of processes. Cases, data objects, and activities are the central concepts of case handling. Activities can write a subset of the case’s data objects and while some are optional, other data objects are mandatory to complete the activity. Activities can be skipped, when their mandatory data objects are already defined, e.g. by a previous activity. Similarly to the presented approach, case states in [2] depend not only on control-flow, but also on case data. Their approach is formalized by giving generic lifecycles for activities and data objects, as well as event-condition-action rules that describe the execution semantics.

Artifact- or object-centric approaches to process modeling shift the focus from the control-flow to the data perspective. Data objects are considered first-order citizens in the modeling methodology.

The business artifacts approach [4] considers both data and process aspects and represents key business entities as business artifacts. These artifacts have an attached lifecycle specifying states of interest an artifact can be in as well as permissible state transitions, which are realized by services (corresponding to tasks in workflow approaches). In addition, artifacts have an associated information model specifying their attributes similar to a database schema. In [3] the authors present an artifact-centric design methodology, which involves the steps of artifact identification, lifecycle and information model design, service specification and association, operationalization of the logical specification into so-called conceptual flows, optimization of those flows, and implementation. The specification of services includes input and output artifacts as well as which attributes can be written, pre-conditions and effects (post-conditions) of service execution. The association of services to state transitions of artifacts as well as ordering of services is achieved by event-condition-action (ECA) rules.

Triggering events might be external messages, attribute and lifecycle state changes of artifacts, begin and termination of services, and requests by case workers. The condition is specified as a first-order logic formula, although the authors do not clarify over which domain. Possible actions of rules are the performing of services and state changes of artifacts. In addition, rules include constraints on the performers of the service, e.g. requiring certain capabilities. ECA rules prove a powerful and flexible formalism that can simulate both procedural and declarative modeling styles.

Although the methodology is very elaborated it begs the question of rule maintainability. The ECA rules, which contain the business logic to us seem hard to manage and especially modify. The lack of a visual representation has been amended with the introduction of conceptual flows. However, they use an ad-hoc notation, which is harder to understand than for example the BPMN standard. Additionally, it seems that once the flow is optimized and implemented, the flexibility existing on the BOM level is reduced.

The case management modeling and notation (CMMN) is a OMG specification released in 2014 [11] based on the Guard-Stage-Milestone (GSM) approach. CMMN's case plan models structure cases into several stages that are guarded by sentries waiting for certain events to occur and conditions to be fulfilled to enter the stage. The sentries' formulae determine case behavior, however, they are not part of the graphical model. Stages contain tasks that can be repetitive, mandatory or optional, and performed in arbitrary order as long as they are not dependent on another task to terminate. Data is represented as a single case file with multiple items that can be anything from a XML document to a folder hierarchy. The downside of this generic definition is that case data is treated essentially as a black box, and is not used to make automated decisions in the process. Additionally, the graphical presentation of CMMN models to us seems hard to understand compared to BPMN, although only an user study could support this argument.

Declarative process modeling languages follow a different approach. Instead of explicitly specifying the ordering of activities, they use a set of constraints between activities, like precedence or non-coexistence, to exclude possible behaviors. The language framework DECLARE [1] expresses these constraints as LTL formulae over finite traces which are transformed into finite automata to check their satisfaction. In general, constraints are more flexible, i.e. more execution paths are permitted at runtime, however, constraints might also conflict leaving no valid behavior. Therefore, DECLARE verifies the models for dead activities and conflicting constraints. The authors mention that declarative approaches are not suited for prescriptive, strict processes, and become illegible when many constraints have to be expressed. Additionally, the approach in [13] does not handle process data. However, current research on declarative process modeling [9] includes basic support for task data.

3 A Hybrid Framework for Case Management

As the presented approach combines aspects of object-centric models with BPMN, it exhibits quite a few interrelated concepts. We will first provide an overview of this concepts using a sample scenario before we give their formal definitions and discuss operational semantics.

3.1 Overview of Concepts

In our approach, business scenarios are captured in a *case model* that consists of (a) a domain model, (b) a set of object lifecycles, (c) a set of process fragments, and (d) a goal state. A case model is instantiated into a *case*, which represents the scenario at runtime and hence exhibits the notion of *case state* that changes over time, mainly through knowledge workers performing activities. Cases are similar to process instances in traditional workflow systems, however, contrary to those, cases are made up of several fragment instances, as well as data objects.

As a running example we will consider the organization of university seminars. This scenario clearly qualifies as knowledge work, as it is variant-rich, goal-oriented, data-driven, and its course unfold over time. A case is usually started during the semester break by finding a suitable theme and assigning teaching staff responsible for organizing the seminar.

Data and Lifecycles. The *domain model* is part of the case model. It defines the business objects relevant for the scenario as a set of data classes and their associations in an UML diagram. Each data class is a named entity that has a set of attributes, which can assume values from a specified domain. For each data class we can specify a data object creation policy that determines whether data objects are created automatically during case instantiation. One of the data classes is designated the *case class* and as such the root class of any associations. In our example the seminar is the case class, as it holds references to the other data classes. Other relevant domain objects in the scenario are the seminar topics, student enrollments, and teaching staff, as is shown in Fig. 1. As topics should be explicitly suggested by staff members during case execution, no data objects of this type are created during instantiation. For the sake of presentation, we will not consider domain objects like presentations given for and by the students, and papers or software artifacts handed-in by the students.

Each data class in the domain model has an associated object lifecycle (OLC) that specifies valid behavior of its instances, i.e. data objects. Object lifecycles are state transition systems consisting of states and state transitions, as well as initial and final states. Whenever a data object is instantiated, an instance of the associated OLC is created. At runtime, each data object is in exactly one of the states defined by the lifecycle, while different objects of the same class can be in different states. Valid states for a seminar object, i.e. an instance of the data class ‘Seminar’, are for example **in planing**, **prepared**, or **grades submitted**, as depicted in Fig. 2.

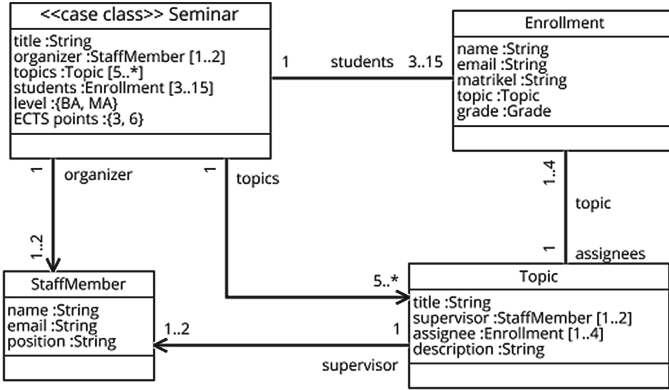


Fig. 1. Domain model for the seminar organization scenario

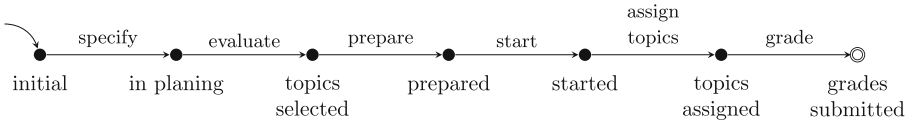


Fig. 2. Lifecycle for data class ‘Seminar’

Fragments and Activities. The concept of splitting process models into smaller fragments and combing them dynamically during runtime is the main difference compared to traditional workflow approaches. A process fragment describes a structured part of a business scenario, and thus defines the possible behavior of a case only in unison with the other fragments. Each fragment, just like usual BPMN process models, consists of events, gateways, activities, and data objects¹. However, our formalization encompasses only a subset of the BPMN specification [10]. Like data classes, fragments have an associated lifecycle that controls the behavior of their instances. Fragment instantiation is controlled by a pre-condition requiring a data object to be in a certain state, and can occur multiple times for the same case, such that multiple instances of the same fragment can be present at the same time and run in parallel.

Our exemplary scenario encompasses several fragments. Fragment **setup**, depicted in Fig. 3, decides on the format of the seminar, asks staff members for topic proposals, and selects among them after the proposal deadline. Fragment **topic proposal**, shown in Fig. 4 can be instantiated multiple times while the seminar is in state **in planing** to create and propose new topics. Due to space limitations the other fragments dealing with student enrollment and assignment, preparation of presentations and student supervision, as well as grading are not depicted in this paper.

¹ We need to distinguish between the BPMN modeling construct named data objects used in fragments and the instances of a data class present at runtime. The former represent the latter in the model.

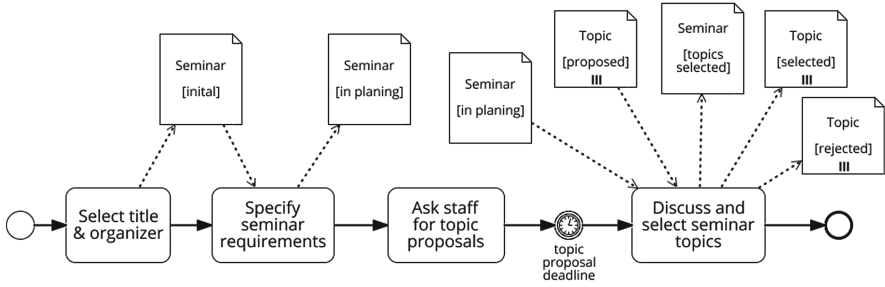


Fig. 3. Fragment for seminar setup

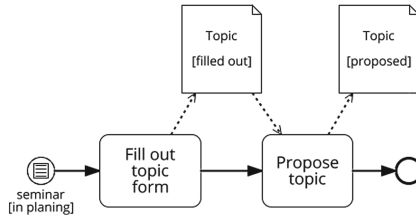


Fig. 4. Fragment for topic proposals

While individual fragments are usually straightforward, their interplay allows for complex behavior. The set of fragments cannot be considered fixed, as case workers might find new ways to deal with certain situations, ways, that were not considered during design of the case model. Therefore, our approach allows to add fragments during runtime. Let us consider that we would like to support cancellation of students. The case worker might add a fragment that specifies how to deal with such a situation.

As usual, activities are the basic units by which work is performed, mainly by creating and manipulating data objects. Activities are only enabled, when their data pre-condition is met, i.e. when a set of specified data objects is in a certain state. Activity instances follow a similar lifecycle like the one defined in [15]. They are instantiated once the fragment they are part of is instantiated.

Termination of a case is defined differently than for workflows, because cases contain multiple fragment instances and data objects. Keeping in mind the goal-orientation of case management, we state that a case is finished, when certain data objects have reached a desired state. In our example the case is terminated, when grades have been submitted to the university. In general, several data objects are involved in the formulation of the termination condition.

3.2 The Domain Model

To formalize our approach we have to formally define the concepts introduced in the last section, i.e. domain model, object life cycle, and fragment. This will be

achieved in this section, while the next will instantiate these model-level concepts and discuss the notions of case state and progress.

Definition 1 (Domain Model). *The domain model is a tuple $\mathcal{D} = (DC, AT, D_c, pol, class, dom)$, where $DC := \{D_1, \dots, D_k\}$ is a set of data classes, $AT := \{\alpha_1, \dots, \alpha_l\}$ is a set of typed attributes, and $D_c \in DC$ is a mandatory and unique data class, called the case class. The function $pol : DC \rightarrow \{true, false\}$ specifies, whether data objects of a data class should be created during case instantiation. The function $class : AT \rightarrow DC$ maps each attribute to exactly one data class it belongs to. The function $dom(\alpha_i) : AT \rightarrow \{Integer, Float, String, Boolean\} \cup DC$ specifies the domain of an attribute, e.g. String or a data class $D_i \in DC$.*

The domain model specifies data classes that represent business entities relevant for the scenario. Basically, a data class is a named set of typed attributes that represents a domain element. Data classes can be associated with each other, however, in this paper we refrain from formalizing associations and multiplicities. Domain models can be expressed as UML class diagrams. The exemplary scenario, shown in Fig. 1, defines data classes ‘Seminar’, ‘Topic’, ‘Enrollment’, and ‘StaffMember’.

3.3 Lifecycles

Definition 2 (Lifecycle). *A lifecycle L is a labeled transition system represented by the tuple $(Q, \Sigma, q, \Omega, \rightarrow)$, where Q is a set of states, Σ is a set of actions, $q \in Q$ is the unique initial state, $\Omega \subseteq Q$ is the set of final states, and $\rightarrow \subseteq Q \times \Sigma \times Q$ is the transition relation.*

Lifecycles specify valid states and permissible state transition of model elements. Our framework defines five generic lifecycles L_C, L_F, L_A, L_G and L_E , that are independent of a concrete scenario and used for cases, fragments, activities, gateways, and BPMN events respectively. L_C , the case lifecycle, depicted in Fig. 5, for example determines how case instances behave at runtime. The activity lifecycle L_A , depicted in Fig. 7, governs the behavior of all activity instances. How exactly the interplay of different instances works, is described in Sect. 4 when we discuss the operational semantics of case models. Graphically, lifecycles are represented by the usual notation for state transition systems.

The fragment lifecycle L_F , shown in Fig. 6, is very similar to the case lifecycle, with exception of the **enabled** state, which indicates whether the data precondition of the fragment is fulfilled. Slightly more complicated is the activity lifecycle depicted in Fig. 7. To reach the **enabled** state, activity instances must be both control-flow-enabled (denoted by action **cfe**), and data-flow-enabled (action **dfe**). Because the data objects references by the activities’ data preconditions can change their state, an activity instance can be data-flow-disabled (action **dfd**) again. Activity instances can also be skipped in some situations, e.g. when the follow a XOR gateway, by performing the action **skip**.

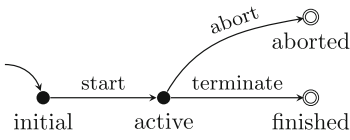


Fig. 5. Lifecycle of a Case

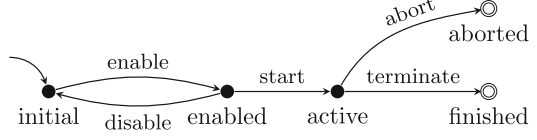


Fig. 6. Lifecycle of a Fragment

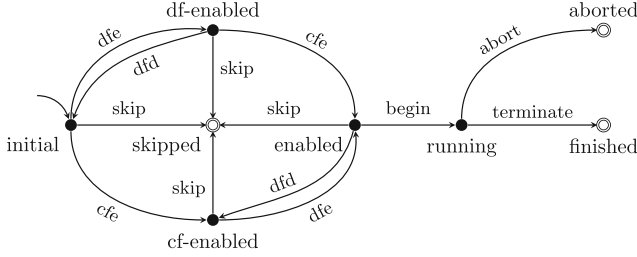


Fig. 7. Lifecycle of an Activity

The gateway lifecycle in Fig. 8 specifies that gateways can be **opened**, **closed**, or **skipped**. Finally, the valid states of BPMN events are defined by the event lifecycle in Fig. 9. Events can either occur directly, e.g. a blank start or an end event, or they are **waiting** for some external trigger to occur.

In contrast to these generic lifecycles, each data class $D_i \in DC$ in the domain model has its own associated scenario-specific lifecycle $lc(D_i) = L_{D_i}$. The function lc associates lifecycles to elements of the case model, not only to data classes, but also to cases, fragments, activities, gateway, events, and their lifecycles as we will see later. We denote the set of scenario-specific data class lifecycles as L_{DC} for a domain model $\mathcal{D} = (DC, AT, D_c, pol, class, dom)$.

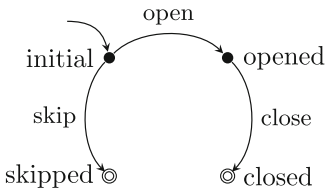


Fig. 8. Lifecycle of a Gateway

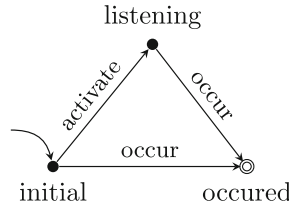


Fig. 9. Lifecycle of an Event

For example, the lifecycle of data class ‘Topic’ from our introductory scenario is depicted in Fig. 10. For each topic that is prepared in the preparation phase for a seminar, one data object is created as instance of the data class ‘Topic’. All these data objects ‘topic A’, ‘topic B’, etc. follow the same lifecycle $L_{Topic}, Topic \in DC$. However, different topics might be in different states, e.g. while ‘topic A’ is **proposed**, ‘topic B’ might be already **selected**.

3.4 Process Fragments

For a domain model \mathcal{D} we define a set $COND$ of terms called *data object state conditions*. An atomic condition is of the form $D[q]$, where $D \in DC$ is a data class and $q \in Q$ is a state in D 's associated lifecycle $lc(D) = (Q, \Sigma, q, \Omega, \rightarrow)$. Any combination of atomic conditions written in disjunctive normal form is also a term $\in COND$. Data objects state conditions are evaluated in the context of a case state. An exemplary data object state condition for our exemplary scenario is `Seminar[in planing]`.

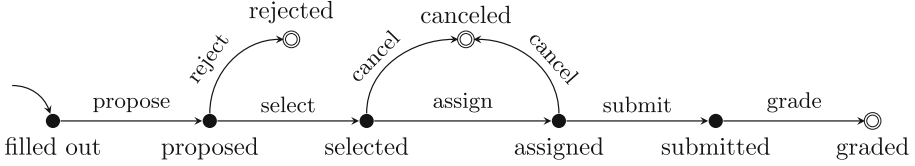


Fig. 10. Lifecycle for data class ‘Topic’

Definition 3 (Process Fragment). Let $\mathcal{G} := \{G_{<}^{\wedge}, G_{>}^{\wedge}, G_{<}^{\times}, G_{>}^{\times}\}$ be a set of gateway types, representing AND split, AND join, XOR split, and XOR join. Let $\mathcal{D} = (DC, AT, D_c, pol, class, dom)$ be a domain model as defined in Definition 1.

Then a process fragment F is a tuple $(N_A, N_E, N_G, N_D, Cf, Df, \gamma, \delta, \Delta)$, where

- N_A, N_E, N_G, N_D are disjunctive sets of activity nodes, event nodes, gateway nodes, and data nodes,
- $N = N_A \cup N_E \cup N_G \cup N_D$ is the set of all nodes,
- $Cf \subseteq (N \setminus N_D) \times (N \setminus N_D)$ is the control-flow and
- $Df \subseteq (N_D \times N_A) \cup (N_A \times N_D)$ is the data-flow relation,
- $\delta : N_D \rightarrow \bigcup_{\forall i} (D_i \times Q_i)$ maps each data node to a pair of data class $D_i \in DC$ and one of its states,
- $\gamma : N_G \rightarrow \mathcal{G}$ assigns a gateway type to each gateway node,
- $\Delta \in COND$ is the fragment’s data pre-condition.

We use the following usual notions.

- ${}^{\circ}N$ (resp. N°) denotes the first (resp. last) element of an ordered set N
- $\bullet A := \{B \mid (B, A) \in Cf\}$ denotes the set of A 's preceding nodes
- $A \bullet := \{B \mid (A, B) \in Cf\}$ denotes the set of A 's successive nodes
- $\blacksquare A := \{\delta(D) \mid (D, A) \in Df\}$ denotes the data pre-condition of an activity node, consisting of pairs of a data class and one of its lifecycle states.
- Similarly, $A \blacksquare$ denotes the data post-conditions of an activity node
- $\blacksquare F := \Delta$ is used to refer to the data pre-condition of the fragment F .

The notion of a *process fragment* formalizes certain aspects of usual BPMN process models, closely following the BPMN specification [10]. However, we focus on the parts relevant for this paper and exclude constructs like pools and lanes, complex gateways, sub-processes, boundary events, as well as message flows.

The BPMN specification provides basic modeling constructs for data modeling, see [10, Section 10.4.1], that are used in fragments to represent data pre- as well as post-conditions of activities. Following [7] we refer to these model elements as *data nodes*. Each data node in the fragment model stands for one data object at runtime that is required for the activity to be enabled (pre-condition) resp. that is produced when the activity terminates (post-condition).

Definition 4 (Well-formed Process Fragment). *A process fragment $F = (N_A, N_E, N_G, N_D, Cf, Df, \gamma, \delta, \Delta)$ is called well-formed, if the following propositions are true.*

- (a) *The first and last nodes regarding the order Cf are unique, i.e. $|\circ N| = |N^\circ| = 1$*
- (b) *The last node is an event node $N^\circ \in N_E$*
- (c) *The first node is an activity node $\circ N \in N_A$*
- (d) *Activity and event nodes have exactly one predecessor and one successor, i.e. $\forall X \in (N_A \cup N_E) \setminus (\circ N \cup N^\circ), |\bullet X| = |X \bullet| = 1$*

As the first and last elements are unique according to (a), we use $\circ N, N^\circ$ to refer to the start activity, respectively end event of a fragment. Similarly, because of (d) we overload to notion of $\bullet A$ and $A \bullet$ to refer to the unique element rather than the set when talking about activity or event nodes $X \in N_A \cup N_E$. According to (c) start events are not part of the formalization. They are used in the graphical presentation of fragments to indicate the data pre-condition of a fragment.

Now, that all its components are defined, we define the notion of a case model, which captures the essence of a business scenario.

Definition 5 (Case Model). *A case model $(\mathcal{D}, \mathcal{L}, \mathcal{F}, \mathcal{A}, tc, lc)$ consists of a domain model \mathcal{D} , a set of lifecycles $\mathcal{L} := \{L_C, L_F, L_A, L_G\} \cup \{lc(D_i) = L_{D_i} \mid D_i \in DC\}$, a set of well-formed process fragments \mathcal{F} , a set of activities \mathcal{A} , a termination condition $tc \in COND$, and a function lc assigning a lifecycle to the other elements.*

4 Operational Semantics of Case Models

This section formally specifies the operational semantics of case models, i.e. their behavior at runtime. To perform a case model it needs to be instantiated into a case that is in an initial state. This is described in Sect. 4.1, while Sect. 4.2 explains how the case state can change according to case progress rules.

4.1 Case State and Instantiation

Cases reside on the instance level and consist of fragment, activity, gateway, and event instances, as well as data objects, i.e. instances of data classes defined in the domain model. Each of these elements is at any time in exactly one lifecycle state and values are assigned to each data object attribute. A case is in flux, new data objects are created, the states of activity instances change, the case finally terminates, however, the case identifier stays the same. Each snapshot in this series is referred to as a *case state*, which is formally defined as follows.

Definition 6 (Case State). *Given a case model $M = (\mathcal{D}, \mathcal{L}, \mathcal{F}, \mathcal{A}, tc, lc)$ multiple cases c_1, c_2, \dots can be instantiated forming the set of cases $Cases_M$. At any time a case $c \in Cases_M$ is in a certain case state S . A case state S is a tuple (I, in, cs, val) , where $I = FI \cup AI \cup GI \cup EI \cup DO$ is a set of instances, partitioned into fragment, activity, gateway, and event instances, as well as data objects, $in \subset (I \times I)$ defines the inclusion among instances and data objects, cs maps instances and data objects, including the case c itself, to their lifecycle states, and $val : (DO \times AT) \rightarrow dom(AT)$ assigns values to data object attributes. The (infinite) set of all possible states of instances for a case model M is denoted as $States_M$.*

The inclusion relation *in* gives rise to a directed, acyclic graph called *case graph*. It is rooted in the case identifier $c \in Cases_M$ and specifies which activity instances belong to which fragment instances, as well as which data objects are bound to which activity instance.

New cases can be either manually instantiated by a knowledge worker or automatically when an external event occurs. Instantiation of a case model creates a new case instance, one instance for each fragment, and instances for all activity, gateway, and event nodes in every fragment. The initial state of these instances is determined by their associated lifecycles. For some instances, lifecycle transitions occur during instantiation, e.g. an activity instance belonging to the first activity node of a fragment² will be control-flow-enabled by the engine. Activity, gateway, and event instances are in inclusion relation with their respective fragment instance and depending on the data object creation policy of a data class, data objects are created in their initial state.

For our exemplary scenario the initial state is $S_0 = (I_0, in_0, cs_0, val_0)$, where $I_0 = FI_0 \cup AI_0 \cup GI_0 \cup EI_0 \cup DO_0$ and FI_0 contains one instance for the setup fragment (f_1) and one for the topic proposal fragment (f_2). AI_0, GI_0, EI_0 contain instances for all activity, gateway, and event nodes respectively. Those instances are related to their fragment instance via the *in* relation. DO_0 is empty, because data objects of class ‘Seminar’ and ‘Topic’ have to be created explicitly during the case according to the data object creation policy. Because DO_0 is empty, there are no attributes to which val_0 could assign values. The state $cs(f_1)$ of

² When it is clear from the context, we will speak of the first activity, when we mean the activity instance belonging to the first activity. Bear in mind, that cases are on instance level.

the setup fragment instance is *enabled*, while $cs(f_2) = \textit{initial}$, because the data precondition $\blacksquare F_2 = \textit{Seminar}[\textit{in planing}]$ is not fulfilled. The activity instance of “select title & organizer” is enabled, because it is the first activity of a fragment and has no data pre-condition. All other activity instances are either in state *initial* or *df-enabled*, depending on whether they have a data pre-condition.

For cases that are automatically started due to an external event, it would be useful to consider input data for the case derived from the event. To achieve this, data objects would need to be created with attribute values assigned according to the triggering event. However, the mapping of external events to data objects is beyond the scope of the basic formalism.

4.2 Case Progress

Knowledge workers progress a case by performing activities, in addition automatically performed system activities, as well as external events can drive the case’s progress. These state changes, called global transitions, are governed by a set of rules that together define the operational semantics of our approach. Because a case c consists of many component instances – fragment, activity, gateway, and event instances, as well as data objects – its state S is compounded of the component instances’ states, which are captured by their associated lifecycle and changed through lifecycle transitions. However, these transitions cannot occur on their own in isolation, but only when triggered by a rule due to a global transition. The following definition introduces triggering of lifecycle transitions.

Definition 7 (Lifecycle transitions). *Let $M = (\mathcal{D}, \mathcal{L}, \mathcal{F}, \mathcal{A}, tc, lc)$ be a case model and $S = (I, in, cs, val) \in States_M$ be the state of a case c of M . Let further $i \in I$ be an instance with associated lifecycle $lc(i) = (Q, \Sigma, q, \Omega, \rightarrow)$ in state $cs(i) = q_s$, $q_s \in Q$. We write $action(i)$ to denote the triggering of lifecycle transition $(q_s, action, q_t) \in \rightarrow$. This lifecycle transition results in changing the instance’s state from $cs(i) = q_s$ to $cs'(i) = q_t$.*

Take for example fragment instance f in state $cs(f) = \textit{active}$ and activity instance $a, (a, f) \in in$ in state $cs(a) = \textit{running}$. Let us assume further that the activity node, a is an instance of, is the final activity node in its fragment. Although the fragment lifecycle L_F allows a transition (*active, terminate, finished*), this transition is taken only when the end event of that fragment occurs. When the case worker terminates the final activity through the frontend, several lifecycle transitions are executed by the engine. First, the lifecycle state of the terminated activity instance changes. As a result, the succeeding end event performs the lifecycle transition *occur*, which triggers the lifecycle transition *terminate* of the fragment instance. As a result of the global transition the case is in state S' with $cs'(a) = cs'(f) = \textit{finished}$.

To ease definition of progress rules we define some helper functions. $type : DO \rightarrow DC$ maps data objects to their data class. $node$ maps activity, gateway, and event instances to the node they are instances of. $frag : FI \rightarrow \mathcal{F}$ maps fragment instances to the fragment they are instances of. The notion of pre/post-set of nodes is extended to instances, i.e. $i \in AI, i \bullet = \{x \mid node(x) \in node(i) \bullet \wedge$

$(i, f), (x, f) \in in\}$, analogously for $\bullet i$. Also, data pre/post-conditions of activity nodes are extended to instances, i.e. $a \in AI, a^\blacksquare = node(a)^\blacksquare$.

Data object state conditions as well as data expressions are evaluated in the context of a case state, yielding either true or false. An atomic condition $D[q] \in COND$ holds true in a state S , if and only if there exists a data object of type D that is in state q , i.e. $\exists d \in DO : type(d) = D \wedge cs(d) = q$. Compound formulae are evaluated according to the usual rules of conjunction and disjunction. Data pre-conditions of activities are evaluated in a similar fashion.

Definition 8 (Fulfilled data pre-conditions). *Let $M = (D, \mathcal{L}, \mathcal{F}, \mathcal{A}, tc, lc)$ be a case model and $S = (I, in, cs, val) \in States_M$ be the state of a case c of M . Let further $a \in AI$ be an activity instance of activity node $node(a) = A$, and $\blacksquare A = \{(D_1, q_1), \dots, (D_n, q_n)\}$ be the data pre-condition of A . A subset $B = \{b_1, \dots, b_n\} \subseteq DO$ fulfills the data pre-conditions of A in state S , if $type(b_i) = D_i$ and $cs(b_i) = q_i$. If $\blacksquare A = \emptyset$ the empty set fulfills the data pre-conditions. B is said to be unbound, if $\forall b \in B, \nexists x \in I : (b, x) \in in$.*

An unbound subset of data objects that fulfills the data pre-conditions of an activity instance, can be bound to that instance, once it becomes control-flow-enabled and is started by the user. In our running example, in the case state after terminating activity “select title & organizer” the data object set $\{sem\}$ with $type(sem) = Seminar$ is unbound and fulfills the pre-condition of activity “specify requirements”. This leads to the first progress rule.

Rule 1 (Activity Start). *Let $M = (D, \mathcal{L}, \mathcal{F}, \mathcal{A}, tc, lc)$ be a case model and $S = (I, in, cs, val) \in States_M$ be the state of a case c of M . Let $a \in AI$ be an activity instance in state $cs(a) = enabled$, let $f \in FI$ be a fragment instance with $(a, f) \in in$, and $B \subseteq DO$ be an unbound set of data objects, potentially empty, that fulfills the data pre-conditions of a .*

Then S can make a global transition to $S' = (I', in', cs', val)$ with

- (a) $in' = in \cup \{(b_i, a) \mid b_i \in B\}$, i.e. data objects are bound to activity instance a
- (b) cs' is defined by the following lifecycle transitions: $begin(a)$ and
- (c) If f is not yet active, i.e. $cs(f) = enabled$,
 - (i) $start(f)$, i.e. start the fragment instance f
 - (ii) $FI' = FI \cup f'$ where f' is a new fragment instance with $frag(f') = frag(f)$, initialized as described in Sect. 4.1

According to Rule 1(c) a fragment instance f stays in state **enabled** until one of its activity instances begins execution, only then it makes the lifecycle transition $start(f)$. At the same time a new fragment instance f' of fragment F is created in its initial state (although $enable(f')$ will be performed when $\blacksquare f'$ is empty or fulfilled). This allows to create new fragment instances at the moment they are required, ensuring that arbitrarily many instances are available.

Activity Termination with User Input. Running activities can be terminated by knowledge workers when they finished working on that activity, leading to a new global case state. If the activity manipulates data objects, i.e. its data post-set

is non-empty, users can enter attribute values for those data objects through a form. This input determines the valuation of the attributes of those data objects. We formalize the input as a valuation function defined for all bound data objects. When an activity terminates, the state of data objects in its data post-set is changed according to the model and all of its successors are triggered.

Rule 2 (Activity Termination). *Let M, S be defined as before. Let $a \in AI$ be an activity instance in state $cs(a) = running$ included in fragment instance $f \in FI$, i.e. $(a, f) \in in$. Let $B = \{b \mid (b, a) \in in\}$ be the data objects bound to a and val_{in} be the valuation function provided by the user.*

Then S can make a global transition to $S' = (I, in', cs', val')$ with

- (a) *cs' is defined by the following lifecycle transitions: $terminate(a)$ and*
 - (i) *a 's successor is triggered, i.e. if $a\bullet \in AI$ then $cf-enable(a\bullet)$, if $a\bullet \in GI$ then $open(a\bullet)$, if $a\bullet \in EI$ then $occur(a\bullet)$*
 - (ii) *The lifecycle states of all bound data objects $b \in B$ are changed according to the data post-conditions of $node(a)$.*
 $type(b) = D_i \wedge (D_i, qt) \in a\blacksquare \wedge (cs(b), action, qt) \in \rightarrow_i \implies action(b)$
 - (iii) *new data objects d are created in the state $cs'(d) = q_0$,*
 $DO' = DO \cup \{d \mid type(d) = D \wedge (D, q_0) \in a\blacksquare \setminus a\blacksquare\}$
 - (iv) *Attribute value assignment val' is pieced together from the previous valuation val and the user input val_{in} .*
 - (v) *$terminate(c)$, if the termination condition is fulfilled*
- (b) *$in' = in \setminus \{(b, a) \mid b \in B\}$, i.e. data objects in B are released*

Following this rule cases terminate immediately, once their termination condition becomes true. On the other hand, nothing prevents the knowledge worker to continue working on a case that is terminated. The frontend should display that the termination condition is fulfilled and offer the possibility to close the case.

Rule 3 (Event Occurrence). *When an event $e \in EI$ occurs in a state S , it triggers its successor $x = e\bullet$, by performing the appropriate lifecycle transition.*
 $occur(e) \implies cf-enable(x), x \in AI \vee open(x), x \in GI \vee occur(x), x \in EI$

Rule 4 (Gateway Behavior). *Let M, S be defined as before and let $g \in GI$ be a gateway instance.*

- (a) *When a XOR split opens it triggers its successors, i.e. $open(g) \wedge \gamma(g) = G_{<}^x \implies cf-enable(x_i), x_i \in AI \vee open(x_i), x_i \in GI \vee occur(x_i), x_i \in EI$, for all $x_i \in g\bullet$*
- (b) *A XOR split closes and skips all alternatives when one activity begins, i.e. $begin(x_i) \wedge \gamma(g) = G_{<}^x \implies close(g) \wedge skip(x_j), x_j \neq x_i$*
- (c) *When AND splits and XOR joins open, they trigger their successors and close, i.e. $open(g) \wedge \gamma(g) \in \{G_{<}^{\wedge}, G_{>}^x\} \implies close(g) \wedge (cf-enable(x_i), x_i \in AI \vee open(x_i), x_i \in GI \vee occur(x_i), x_i \in EI)$, for all $x_i \in g\bullet$*
- (d) *An AND join closes when its last predecessor terminates, i.e. $terminate(y_i) \wedge \forall y_j \in \bullet g \setminus \{y_i\} : cs(y_j) = finished \wedge \gamma(g) = G_{>}^{\wedge} \implies close(g) \wedge (cf-enable(x_i), x_i \in AI \vee open(x_i), x_i \in GI \vee occur(x_i), x_i \in EI)$*

If there exist paths from a XOR split to a XOR join without activities in between, activities on alternative paths are called optional. Optional activities are enabled, when the XOR split opens, but have to be skipped explicitly.

Rule 5 (Fragment Termination). *If the end event of a fragment instance occurs, that fragment instance terminates.*

$$\text{occur}(N^\circ) \wedge (N^\circ, f) \in \text{in} \implies \text{terminate}(f)$$

Application of the rules to an initial case state S_0 yields the structure of all reachable case states. However, as rule applicability depends on user input and selection of activity bindings, the state space of a case can grow tremendously.

5 Discussion

In this section we explore whether and how our approach eases modeling and execution of flexible processes. The central idea of our approach is to model business scenarios as a set of small fragments and use data object states to combine them at runtime. The alternative would be to express the complex flows that ensue through dynamic fragment combination in one BPMN model. Imagine, the scenario allows to cancel a seminar before the semester started. A BPMN model would necessitate many gateways to allow for cancelation at the right places in the model and hence would become too large to be manageable.

Our fragment approach makes it easy to add fragments and keeps the fragments simple, because fragment combination is based on data objects instead of gateways. This fits naturally for flexible processes in knowledge work, where different courses of action can be expressed by different fragments. One could add a fragment for canceling seminars after they have started, another one for dealing with students who quit their enrollment. While the fragments are much simpler there can be quite many of them and the flows resulting from their combination pose a threat for model comprehension.

Therefore, it is essential for our approach to answer questions about possible flows, e.g. how can I reach a case state satisfying the termination condition. The formalization lays the foundation for this kind of formal analysis of case models. Without the presented operational semantics for cases, analysis techniques would not be able to generate the state space. Thus, the provided formalization is a precondition for verification of cases, e.g. to find deadlocks and reachable states.

Finally, the formal background of our approach helped in implementing a prototypical engine for executing case models [5]. The implementation closely follows the formal definitions by using state machines to control the state of instances and implementing the progress rules.

6 Conclusion

Driven by the deficits of traditional process management technology in supporting knowledge intensive processes, since about a decade there is interest in case management. As discussed in the related work section, several approaches have been presented with different assumptions, notations, and limitations.

In the approach presented in this paper, we have tried to balance the structured parts of cases with the unstructured, flexible ones. We did so by following a hybrid approach in which process fragments expressed in BPMN support the structured part, while enablement conditions based on data objects and their states support the variability aspects.

To validate the approach in general and the operational semantics in particular, they have been prototypically implemented in a software system called Chimera. Initial user tests show the appropriateness of the modeling approach and the effectiveness of the defined execution semantics. However, a thorough empirical analysis involving a formal user study is not in the scope of this paper. On the other hand, this paper provides the technical results of our research which provide the basis for a future empirical evaluation.

References

1. van der Aalst, W.M.P., Pesic, M., Schonenberg, H.: Declarative workflows: balancing between flexibility and support. *Comput. Sci. Res. Dev.* **23**(2), 99–113 (2009). <http://link.springer.com/article/10.1007/s00450-009-0057-9>
2. van der Aalst, W.M.P., Weske, M., Grünbauer, D.: Case handling: a new paradigm for business process support. *Data Knowl. Eng.* **53**(2), 129–162 (2005)
3. Bhattacharya, K., Gerede, C.E., Hull, R., Liu, R., Su, J.: Towards formal analysis of artifact-centric business process models. In: Alonso, G., Dadam, P., Rosemann, M. (eds.) *BPM 2007. LNCS*, vol. 4714, pp. 288–304. Springer, Heidelberg (2007)
4. Cohn, D., Hull, R.: Business artifacts: a data-centric approach to modeling business operations and processes. *IEEE Data Eng. Bull.* **32**(3), 3–9 (2009)
5. Haarmann, S., Podlesny, N., Hewelt, M., Meyer, A., Weske, M.: Production case management: a prototypical process engine to execute flexible business processes. In: *Proceedings of the BPM Demo Session*, pp. 110–114 (2015)
6. Hauder, M., Pigat, S., Matthes, F.: Research challenges in adaptive case management: a literature review. In: *Enterprise Distributed Object Computing Conference Workshops and Demonstrations (EDOCW)*, pp. 98–107. IEEE (2014)
7. Meyer, A.: Data perspective in business process management. Dissertation, Universität Potsdam (2015)
8. Meyer, A., Herzberg, N., Puhmann, F., Weske, M.: Implementation framework for production case management: modeling and execution. In: *Enterprise Distributed Object Computing (EDOC)*. IEEE (2014)
9. Montali, M., Chesani, F., Mello, P., Maggi, F.M.: Towards data-aware constraints in DECLARE. In: *Proceedings of the 28th Annual ACM Symposium on Applied Computing*, pp. 1391–1396 (2013)
10. Object Management Group: Business Process Model and Notation (BPMN), Version 2.0.2 (2013). <http://www.omg.org/spec/BPMN/2.0.2/>
11. Object Management Group: Case Management Model and Notation (CMMN) (2014). <http://www.omg.org/spec/CMMN/1.0>
12. Reichert, M., Weber, B.: *Enabling Flexibility in Process-Aware Information Systems*. Springer, Heidelberg (2012)
13. Schonenberg, H., Weber, B., van Dongen, B.F., van der Aalst, W.M.P.: Supporting flexible processes through recommendations based on history. In: Dumas, M., Reichert, M., Shan, M.-C. (eds.) *BPM 2008. LNCS*, vol. 5240, pp. 51–66. Springer, Heidelberg (2008)

14. Swenson, K.D.: Mastering the Unpredictable - How Adaptive Case Management Will Revolutionize the Way that Knowledge Workers Get Things Done. Meghan-Kiffer Press, Tampa (2010)
15. Weske, M.: Business Process Management, 2nd edn. Springer, Heidelberg (2012). <http://link.springer.com/10.1007/978-3-642-28616-2>