Computing (2016) 98:345–373
DOI 10.1007/s00607-015-0442-0

Synthesizing data-centric models from business process models

Rik Eshuis · Pieter Van Gorp

Received: 11 January 2014 / Accepted: 19 January 2015 / Published online: 14 February 2015

Abstract Data-centric business process models couple data and control flow to specify flexible business processes. However, it can be difficult to predict the actual behavior of a data-centric model, since the global process is typically distributed over several data elements and possibly specified in a declarative way. We therefore envision a data-centric process modeling approach in which the default behavior of the process is first specified in a classical, imperative process notation, which is then transformed to a declarative, data-centric process model that can be further refined into a complete model. To support this vision, we define a semi-automated approach to synthesize an object-centric design from a business process model that specifies the flow of multiple stateful objects between activities. The object-centric design specifies in an imperative way the life cycles of the objects and the object interactions. Next, we define a mapping from an object-centric design to a declarative Guard-Stage-Milestone schema, which can be refined into a complete specification of a data-centric BPM system. The synthesis approach has been implemented and tested using a graph transformation tool.

Keywords Data-centric BPM · UML · Case management · Model transformation

Mathematics Subject Classification 68U01 · 68U31

R. Eshuis (✉) · P. Van Gorp
Eindhoven University of Technology, P.O. Box 513, 5600 MB Eindhoven, The Netherlands
e-mail: h.eshuis@tue.nl

P. Van Gorp
e-mail: p.m.e.v.gorp@tue.nl

1 Introduction

The classic way to model business processes is to specify atomic activities and their ordering in a flowchart-like, imperative process model. In recent years, data-centric modeling paradigms have increasingly grown popular in research and industry as alternative to the classic, activity-centric paradigm. Data-centric modeling approaches aim to have a more integrated perspective on business processes by treating data elements as first-class citizens next to process elements [22,29]. Moreover, data-centric modeling approaches support the specification and execution of semi-structured, knowledge-intensive business processes [40], which are more difficult to support using classic process modeling.

These two paradigms are often positioned as alternatives, each having their own modeling techniques and implementation technologies. Data-centric approaches use for instance state machines [21,22,29,46] or business rules [7] as modeling techniques, while activity-centric approaches use process flow models such as UML activity diagrams [41] or BPMN [30], where each modeling technique is supported by dedicated engines.

In this paper, we aim to combine the strengths of both approaches. An activity-centric model shows clearly the behavior of the process, while in a data-centric model the actual behavior is difficult to predict, either since the global process is distributed over different data elements or since the behavior is specified in a declarative way, for instance in the Guard-Stage-Milestone (GSM) approach [7]. Conversely, data-centric approaches enable more flexible ways of performing business processes than activity-centric approaches, which are typically rigid [33].

Given these strengths of both approaches, we envision the following high-level design method for designing data-centric BPM systems. Initially, classic process modeling techniques are used to specify the main "default" scenarios of a process, allowing users to understand and define exactly what the intended behavior is. In a later stage, a data-centric approach is used to actually realize a data-centric BPM system. The backbone of the data-centric BPM system is the behavior specified by the default scenarios. In addition, extra behavior is specified for anticipated exceptional circumstances, for instance by specifying business rules that are not in the main scenarios. This allows actors to perform the process in the prescribed way for default scenarios, but respond in a flexible way to exceptional circumstances not covered by the default scenarios, which is one of the strengths of data-centric process management [33].

To support this way of working, a process model specifying the default scenarios needs to be translated into a data-centric model. Such translations are currently lacking in the literature, as we elaborate in Sect. 7.

This paper outlines a semi-automated approach for creating a data-centric design from a business process model that specifies the main scenarios in which business objects interact. The approach uses synthesis rules that relate process model constructs to object life cycle constructs. The resulting object-centric design can be used as starting point for a data-centric process implementation. In particular, we show how the constructed object life cycles can be translated to GSM schemas [19]. The constructed GSM schemas can be refined with additional "non-default" behavior such as exceptions.

While the last phase of our proposed approach uses GSM schemas, the first phase uses UML [41], since UML offers a coherent notation for specifying both activity-centric and data-centric models. To specify business process models that reference objects, we use UML activity diagrams with object flows. We use UML state machines (statecharts) to model communicating object life cycles, which specify an object-centric design. The key concepts of the GSM model have been incorporated in the emerging OMG standard on case management [5]. Since our mapping consumes UML activity diagrams as input, and produces GSM schemas expressible in CMMN via UML state machines, it is very relevant to the OMG ecosystem of standard modeling languages. The synthesis rules defined on UML activity diagrams are directly applicable to any imperative process modeling language that uses a similar semantics for object flows as UML. As we explain in Sect. 7, the object-flow semantics of BPMN [30] differs from UML [41].

The remainder of this paper is structured as follows. Section 2 summarizes a previously developed approach for synthesizing one object life cycle from a business process model that specifies different states of the object. This paper extends that approach to the case of multiple objects of different types that interact with each other. Section 3 presents synthesis rules that define coordination logic between multiple object life cycles based on object-flow constraints from the activity diagram. Section 4 presents synthesis rules that define execution logic of object life cycles based on control-flow constraints from the activity diagram. Section 5 presents a mapping of the resulting object-centric designs to GSM schemas. Section 6 discusses a prototypical reference implementation of the rule-based translation from activity diagrams to object-centric designs. Section 7 presents related work and Sect. 8 ends the paper with conclusions.

This paper revises and expands a workshop paper [10]. New parts of this paper are a mapping from object-centric designs to GSM schemas and a discussion of a prototypical reference implementation based on graph-transformations.

2 Preliminaries

We assume readers are familiar with UML activity diagrams [41] and UML state machines [41]. Figure 1 shows an activity diagram of an ordering process that we refer to in the sequel. In previous work [9,11], we outlined an approach to synthesize a single object life cycle, expressed as a hierarchical UML state machine, from a business process model, expressed in a UML activity diagram. This paper builds upon that approach. To make this paper self-contained, we briefly summarize our previous results here and explain the extensions made in this paper.

Input to the synthesis approach is an activity diagram with object nodes. Each object node references the same object but in a distinct state, identified by the modeler. An activity can have object nodes as input or output. An activity can only start if all its input object nodes are filled, and upon completion it fills all output object nodes [41]. If an activity has multiple input or output object nodes that reference the same object, the object is in multiple states at the same time; then the object life cycle contains parallelism. Each parallel state specifies a partial view on the global state of the object life cycle [11]. Parallelism may lead to interference if concurrent tasks write the same

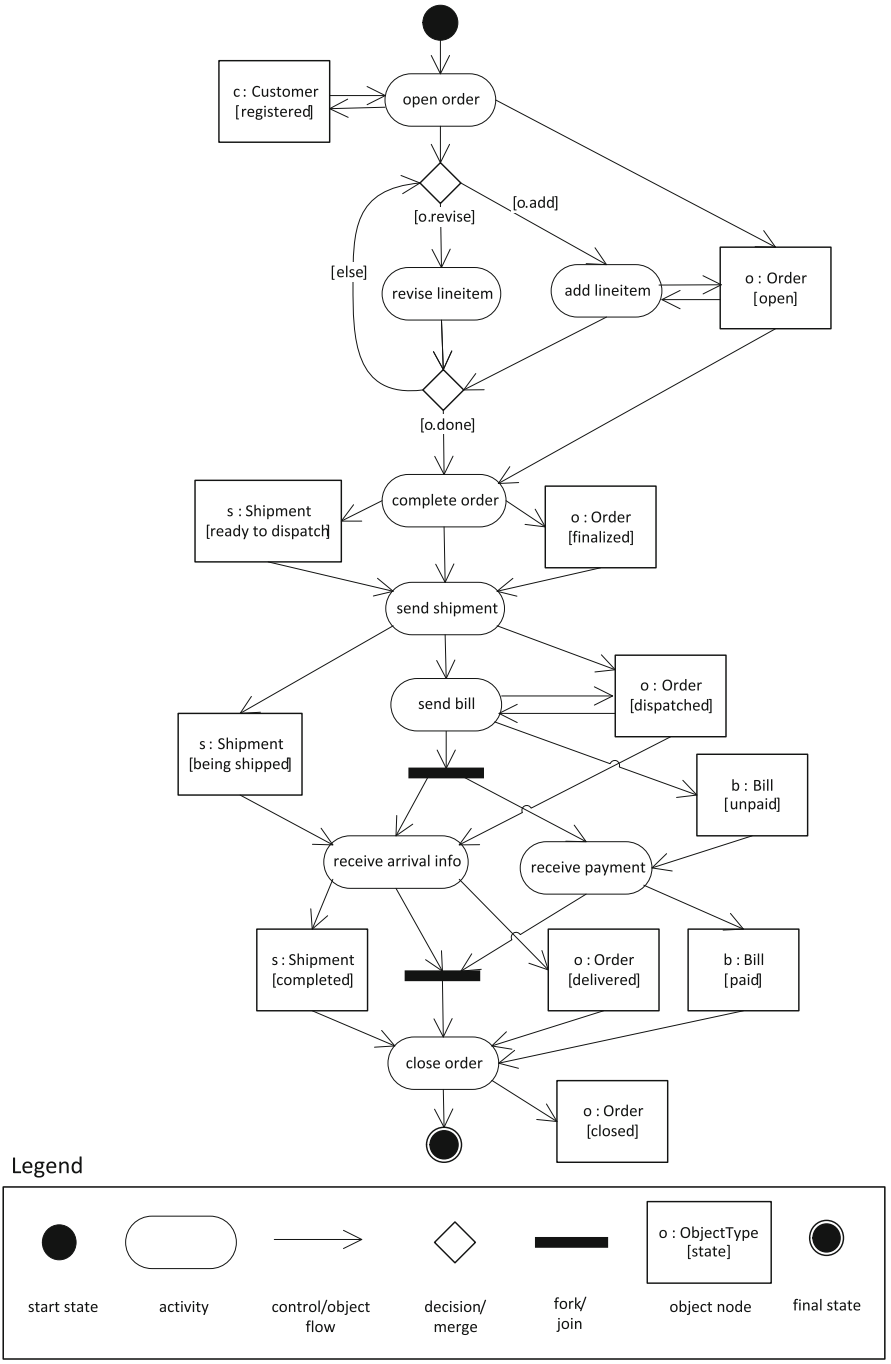

Fig. 1 Activity diagram of ordering process

data element. In Sect. 4.6 we discuss how to deal with such interference. A single object node is exclusive: if multiple activities require an input object from the same object node, only one activity can read (consume) the object.

The synthesis approach consists of two steps. First, irrelevant nodes are filtered from the activity diagram. Object nodes are relevant, but activities are not. Control nodes are only relevant if they influence the object nodes. Second, the filtered activity diagram is converted into a UML state machine by constructing a state hierarchy that ensures that the behavior of the state machine induced by the state hierarchy is equivalent to the behavior of the filtered activity diagram. Each state machine has a start state (black dot) and one or more final states (bull's eyes), which have no explicit counterpart in the activity diagram.

A limitation of that approach [9,11] is that it assumes that all object nodes in an activity diagram belong to the same object. If an activity references multiple different objects with different types, for each object a different version of the activity diagram specific to that object can be created. While this ensures that for each object a state machine skeleton can be created, the generated state machine skeletons are not yet executable, lacking coordination and execution logic.

This paper removes that limitation by defining coordination and execution synthesis rules that materialize state machine skeletons synthesized using the earlier defined approach [9].

Figure 2 shows for some objects their life cycles that are generated using the approach proposed in this paper. The earlier defined synthesis approach [9] is used to construct the skeletons of the life cycles, so the nodes and edges without labels. The approach defined in this paper in Sects. 3 and 4 adds annotation to the skeletons. Also, it refines some atomic states into composite states. We revisit the example in Sect. 4.6.

Activity diagrams also support a pin-style modeling of object flows: each activity can have attached input and output pins that model input and output objects, respectively, of the activity. The basic pin style is equivalent to the object node style [41]: each object node translates into an output pin for each activity that produces objects for the object node and into an input pin for each activity that consumes objects from the object node. However, the general pin-style notation is more expressive, since it allows the specifications of alternative pin sets, called parameter sets [41]. The results in this paper carry over to the basic pin-style notation.

3 Synthesis rules for coordination logic

The translation from UML activity diagrams to communicating state machines that specify object life cycles is based on synthesis rules. Each synthesis rule translates a basic substructure in an activity diagram to a basic substructure of a state machine. The state machines constructed by applying the synthesis rules form an object-centric design, in which the objects collaboratively execute the process.

In this section, we focus on synthesis rules that define coordination logic. Each synthesis rule for coordination logic is defined on a substructure in an activity diagram that has a single activity and multiple objects connected to the activity by object flows. One of the objects, called the coordinator, is responsible for orchestrating the other

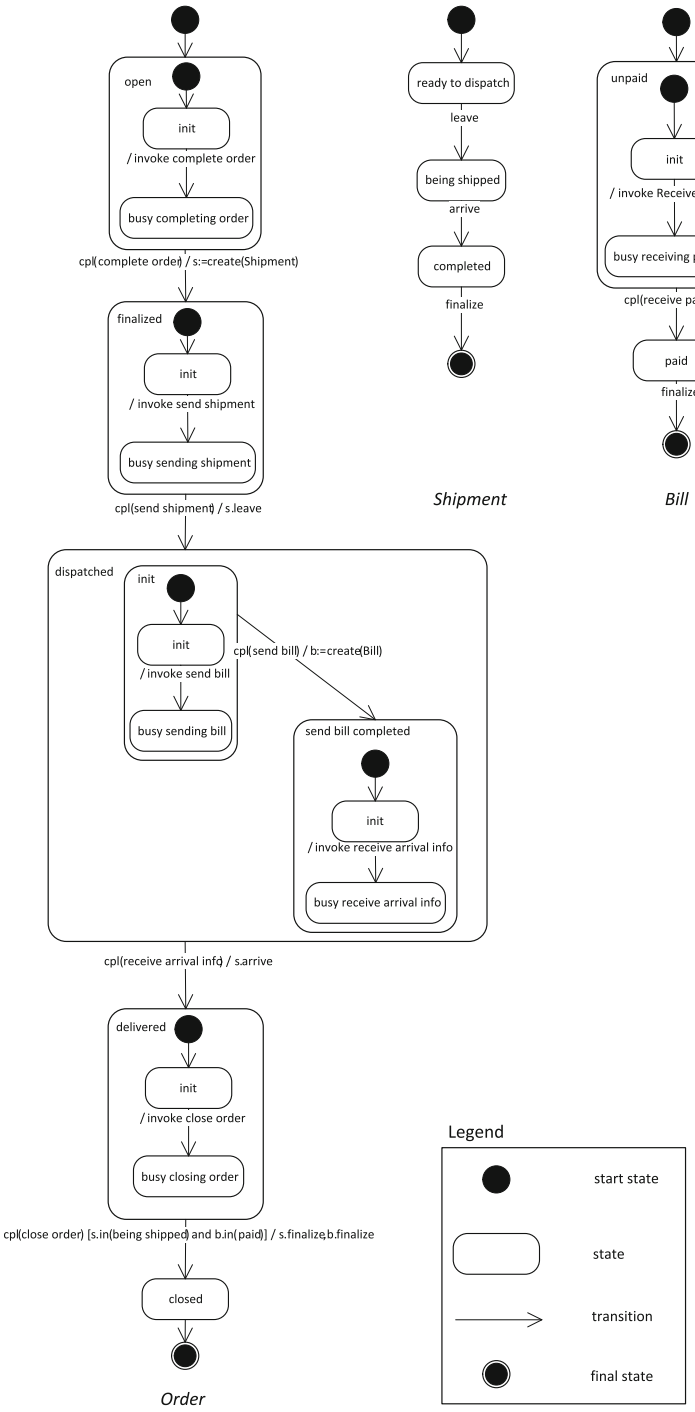

Fig. 2 State machines for objects from ordering process

Table 1 Basic relations between activity A and object $:O$ in activity diagrams

Relation	Object flow from A to $:O[s_1]$	Object flow from $:O[s_2]$ to A	Extra constraint
A creates $:O$	x		
A finalizes $:O$		x	
A accesses $:O$	x	x	
A read-accesses $:O$	x	x	$s_1 = s_2$
A update-accesses $:O$	x	x	$s_1 \neq s_2$

objects, called participants, such that the behavior specified in the substructure is jointly realized by the objects in the object-centric design. Applying coordination-logic rules results in abstract object-centric designs that are not executable, since they do not contain external event triggers or task invocations. Section 4 presents synthesis rules for execution logic, the application of which results in executable object-centric designs.

3.1 Basic notions

Let A be an activity (action node) and let $o{:}O[s]$ be an object node representing object o of type (class) O where o is in state s. Node $:O$ represents an anonymous object of type O with an unspecified state. Activity A and object $:O$ can be related by object flows in five possible ways, as specified in Table 1. The last two relations are specializations of the third (*access*) relation, leaving four basic relations. Note that if there is an object flow between A and $:O$, then exactly one of these four relations holds, so the four basic relations are complete.

We define for each basic relation between A and $:O$ a corresponding coordination-logic synthesis rule, in which the object $:O$ plays the role of participant. Each coordination-logic synthesis rule also requires that A has exactly one coordinator object, different from $:O$, that is responsible for coordinating the behavior of $:O$ in the object-centric design. Performing A typically causes a state change with the coordinator, so A updates the state of the coordinator. Any object that A updates can play the role of coordinator. Alternatively, if A does not update any object, any object that is read by A can become coordinator. Otherwise, there is no coordinator. We consider this as a design error that can be detected automatically and fixed via additional user input. If A accesses multiple objects, the user has to decide which object is responsible for coordinating $:P$. Alternatively, from a process model with object flows a priority scheme on object types can be automatically derived, for instance based on the dominance criterion [21]. The object type with the highest priority can be made the default coordinator of $:P$.

In the sequel, we present synthesis rules for coordination logic. We assume that each activity has one coordinator, which is either derived automatically from the activity diagram or designated by the user. To simplify the presentation, we assume that the coordinator changes state. We explain the impact of considering a coordinator that does not change state at the end of this section.

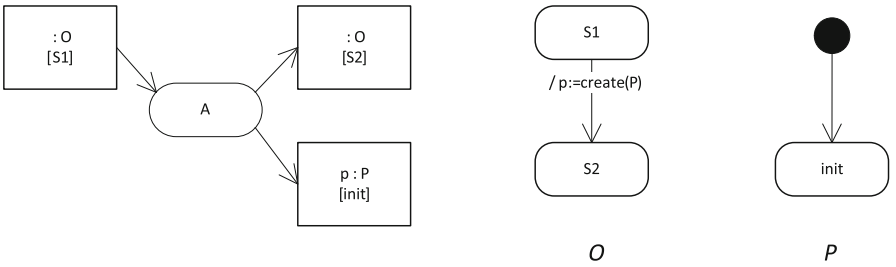

Fig. 3 Synthesis rule for creation

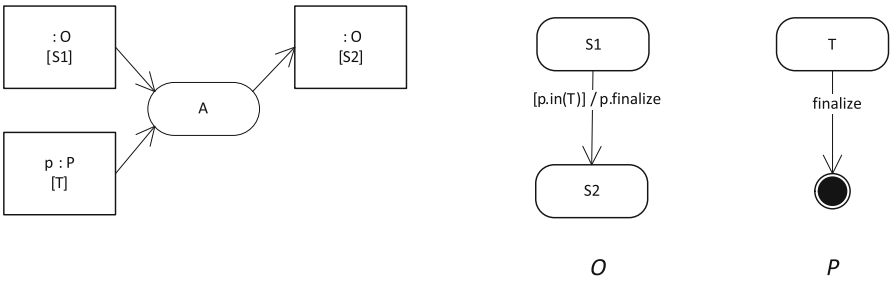

Fig. 4 Synthesis rule for finalization

3.2 Creation

We consider the case that activity A creates an object of type P under coordination of object $:O$. Figure 3 specifies the synthesis rule for creation. The activity diagram specifies that by performing A, the state of $:O$ moves from $S1$ to $S2$ and that an object $p{:}P$ is created whose first state is $init$. In the corresponding object life cycle, coordinator $:O$ creates an object p of type P with action $create(P)$ when moving from $S1$ to $S2$. Coordinator $:O$ is also responsible for executing task A and waiting for its completion, but these details are defined by the synthesis rule for tasks in Sect. 4.

3.3 Finalization

Activity A finalizes object $p{:}P$ under coordination of object $:O$ by moving $p{:}P$ into its local end state, which has no successor state. Finalization does not mean that the object is destroyed. The finalization is realized in the object life cycles (Fig. 4) by letting coordinator $:O$ send a special event $finalize$ to $p{:}P$, provided $:P$ is indeed in the expected state T, modeled with guard condition $p.in(T)$. The $finalize$ event moves the life cycle of $p{:}P$ to the local end state. It might be that the life cycle of $p{:}P$ has multiple end states in different parallel branches; in that case, the other parallel branches of the life cycle can still continue after this branch has been finalized.

3.4 Read-access

If activity A reads object $p{:}P$ under coordination of $:O$, then $:P$ does not change state. This means no transition is required in the object life cycle of $p{:}P$. However, the state

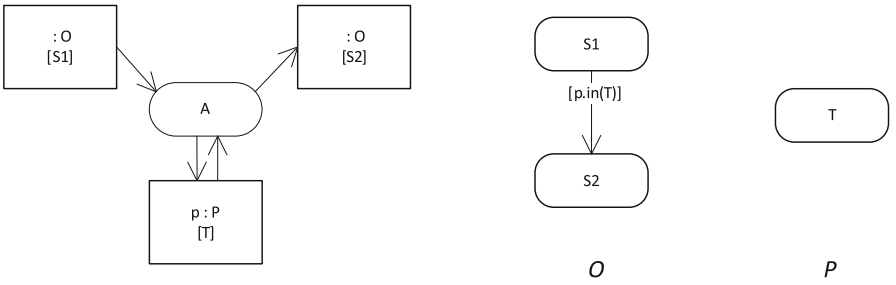

Fig. 5 Synthesis rule for read-access

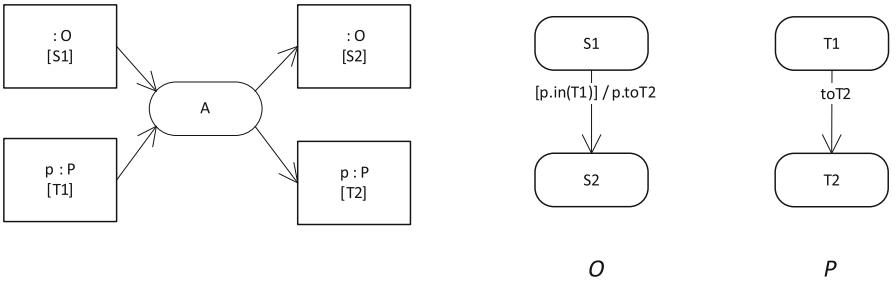

Fig. 6 Synthesis rule for update-access

of P is a precondition for $:O$ to change state, since A can only start if objects $:O$ and $p{:}P$ are present. Therefore, in Fig. 5 the coordinator state machine for $:O$ can only change state if the current state of $p{:}P$ is T.

3.5 Update-access

If activity A updates object $p{:}P$ under coordination of $:O$, then both $:O$ and $p{:}P$ move to a new state. The state change of $p{:}P$ is triggered by $:O$. But coordinator $:O$ may only generate an event for $p{:}P$ if p has already reached the state that is precondition to A. Figure 6 shows that $:O$ generates an event $toT2$ that triggers $p{:}P$ to move to its next state, but only if $p{:}P$ is currently in state T, since otherwise A cannot start.

3.6 Discussion

We next discuss two issues for the translation.

3.6.1 Coordinators with read-access

As explained in Sect. 3.1, the synthesis rules for coordination logic assume the coordinator changes state when activity A is performed. If the coordinator does not change state, the rules need to be slightly adjusted. Denote the coordinator state with S. All the state machines for $:O$ in the coordination rules can be adjusted by letting states $S1$ and $S2$ become substates of composite state S. For instance, Fig. 7 shows the synthesis

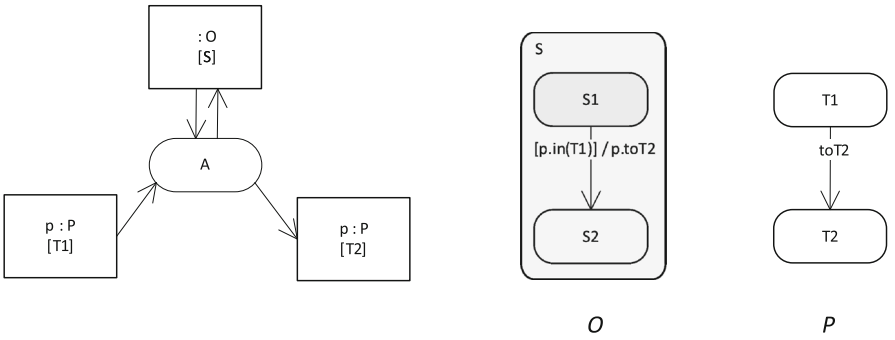

Fig. 7 Synthesis rule for update-access using coordinator with read-access

rule for update-access, modified from Fig. 6, in case the coordinator uses a read-access for A.

3.6.2 Multiple participants

In the discussion of the synthesis rules, we tacitly assumed that for each activity A there is only one participant. In practice, however, there can be multiple participants. For instance, in Fig. 1 activity close order has coordinator o:Order and two participants s:Shipment and b:Bill. In the case, the effects on the coordinator life cycle that are obtained by applying the rule to each participant separately, must be combined in the coordinator life cycle by joining all the guards and actions introduced for each participant. For instance, in Fig. 2 the transition from dispatched to delivered for coordinator o:Order contains a conjunction of the guard conditions obtained by applying the finalization rule for both participants b:Bill and s:Shipment. Similarly, the transition generates two events that are also the result of applying the finalization rule to each participant.

4 Synthesis rules for execution logic

Synthesis rules for coordination logic only capture the object-flow constraints from activity diagrams. To capture control-flow constraints, we define synthesis rules for execution logic. Each execution-logic synthesis rule translates a substructure which contains a basic control-flow construct (task, decision, merge, fork, join) into an object-centric design. These constructs are similar to the standard workflow patterns used in any process language [2].

4.1 Task

A task is invoked in an activity node. A typical distinction is between manual, automated, and semi-automated tasks [1]. For this paper, we only consider automated tasks; we expect the generated designs can be adapted easily to other task types. Figure 8 shows how a task invocation can be specified in an object-centric design. The coordina-

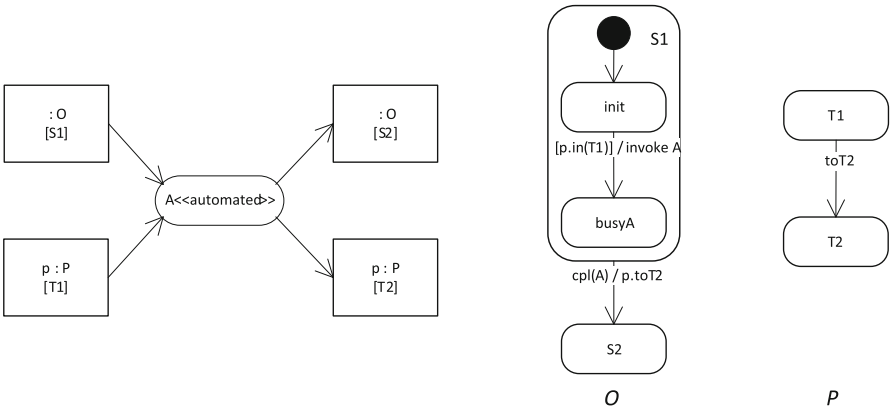

Fig. 8 Synthesis rule for tasks

tor $:O$ is responsible for invoking task A; there its precondition state $S1$ is decomposed into two states, where $busy\ A$ denotes that activity A is being executed. The busy state can be decomposed to model a more complex task life cycle, for instance allowing suspension of a task [6]. However, we assume tasks are successful, so task abortion is not considered. Upon completion, the coordinator moves to $S2$ and informs the other object $p{:}P$ that it has to move to new state $T2$. In Fig. 8, the underlying coordination-logic rule is update-access, but the synthesis rule for tasks can be combined with any coordination-logic rule or be used independently if only a single object is involved. We discuss in Sect. 4.6 how the synthesis rule is applied to the example in Fig. 1.

4.2 Decision

An object node can have multiple outgoing flows. This represents exclusive (choice) behavior: exactly one of the outgoing flows is taken if the object node is active. The actual decision is taken in the control flow, represented by a decision node.

Figure 9 shows the synthesis rule for decision nodes. In the activity diagram, state $S1$ is precondition to both A and B; upon completion of either A or B object $:O$ moves to $S2$ or $S3$. The object life cycle mimics this behavior. Upon entering state $S1$ of $:O$, either A or B is invoked, where A can only be invoked if $p{:}P$ is in state $T2$. If A is invoked, then also $p{:}P$ must change state (update-access rule). As in the case of the previous rule, the precondition state $S1$ is hierarchical. In this case, $S1$ contains the decision logic to decide between A or B; note that this decision logic comes from the control flow in the activity diagram.

4.3 Merge

Similar to the previous case, an object node with multiple incoming object flows represents exclusive behavior: if one of the object flows is activated, the object node is entered. Again, the actual behavior is governed by control flow. The resulting synthesis rule for merge nodes shown in Fig. 10 is symmetric to the rule for decision nodes.

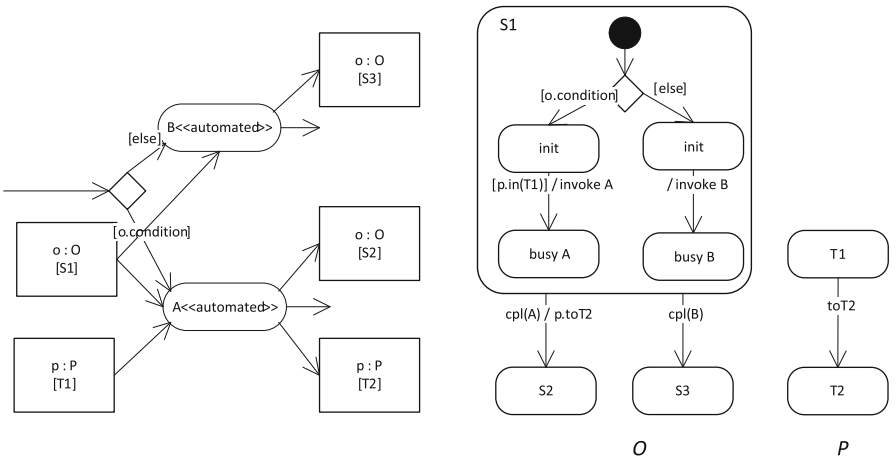

Fig. 9 Synthesis rule for decision nodes

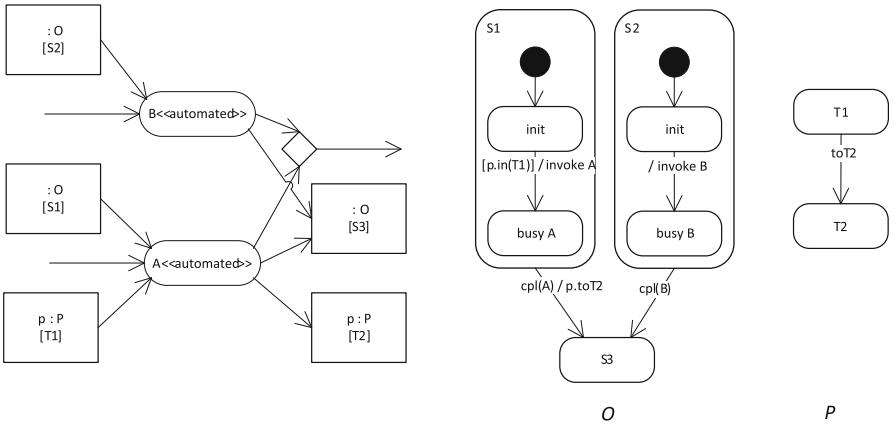

Fig. 10 Synthesis rule for merge nodes

Note that in the object life cycle for $o{:}O$ states $S1$ and $S2$ are exclusive, so they cannot be active in parallel at the same time.

4.4 Fork

So far, we have seen only sequential state machines that do not contain any parallelism. However, an object can be in multiple states at the same time [16,41], each state being a partial view of the global state. In activity diagrams, parallelism inside an object life cycle for object type O is introduced by an activity node that outputs multiple object nodes belonging to O. Since activity nodes activate all outgoing object flows, all output object nodes are filled. Each output object node for O then represents a parallel branch in the life cycle for O. For examples, we refer to earlier work [9].

In the control flow of activity diagrams, parallelism is introduced by a fork node. Figure 11 shows how the resulting synthesis rule for fork nodes is specified. The

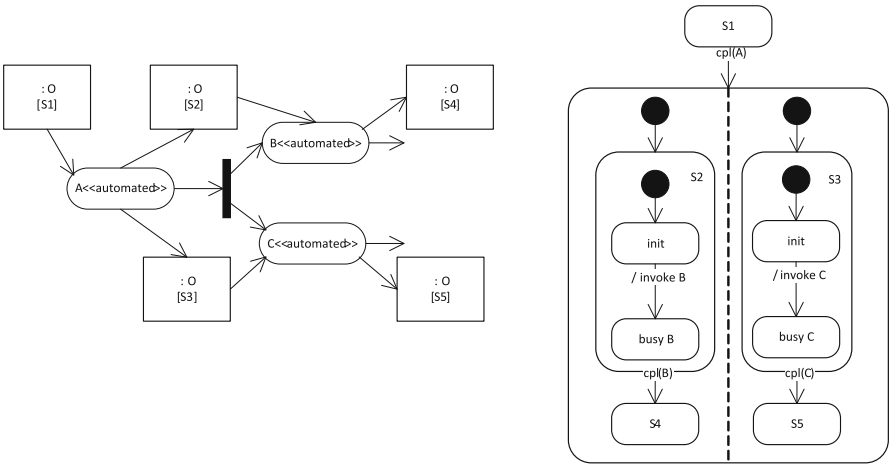

Fig. 11 Synthesis rule for fork nodes

state hierarchy is constructed using the approach we developed previously [9]. The two concurrent states, separated with a dashed line, model the two parallel branches started upon completion of A. The state hierarchy for $S2$ and $S3$ derives from the synthesis rule for tasks. The state hierarchy for $S1$, also due to the task synthesis rule, is not shown to simplify the exposition.

4.5 Join

The rule for join nodes is symmetric to the one for fork nodes (Fig. 12). Complicating factor is to decide where in the life cycle task C should be invoked. Activity C synchronizes the parallel branches in the activity diagram. Similarly, the parallel branches in the object life cycle for $:O$ are only left if activity C completes. This implies that in the life cycle for $:O$ task C needs to be invoked in one of the parallel branches (since activity C is invoked only once in the activity diagram). But if C is invoked in a branch, the other parallel branch must be in the state that is precondition to C. Which parallel branch is chosen to invoke C is arbitrary. In the figure, the task rule for C is applied to $S2$. This means that C can only be invoked if the other parallel branch is in state $S4$.

4.6 Discussion

We highlight the most interesting issues of the translation defined in this section and the previous one.

4.6.1 Interference

The synthesis rules for fork and join nodes introduce concurrency inside a business object, which may lead to interference if two concurrent tasks write the same data

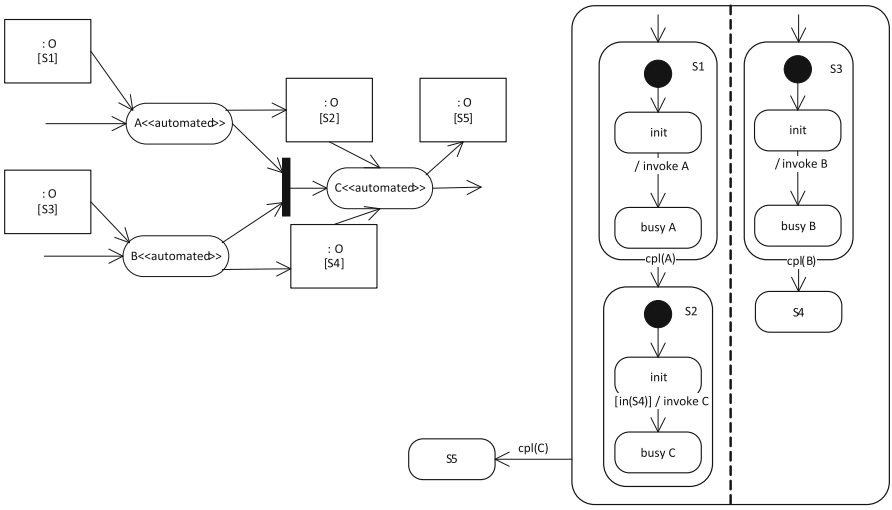

Fig. 12 Synthesis rule for join nodes

element. We do not handle interference in the approach, but interference can be prevented in different ways. First, orthogonal states can be required to access distinct data elements. However, this may be too restrictive. Alternatively, if orthogonal states do access shared data elements, nested transactions [15] can be used by creating for each state a transaction and nesting a transaction t inside another transaction t' if the state of t is descendant of the state of t' in the state machine. Nested transactions allow for relaxed isolation between concurrent tasks in workflow management [45].

4.6.2 Order example

To obtain the state machines in Fig. 2 from the activity diagram in Fig. 1 all four rules for coordination logic are required. Next, the task synthesis rule is used. The decision, merge, fork, and join control nodes in Fig. 1 all reference multiple objects and therefore do not fit in any individual object life cycle. Consequently, the synthesis rules for these control nodes are not applicable for this example.

To derive the state machines in Fig. 2, first the activity diagram is copied and sliced per object type. Next, for each object type a flat state machine skeleton is synthesized using the approach defined in earlier work and summarized in Sect. 2. Then the coordination-logic rules are applied iteratively, which result in annotation on transitions of the state machines. Next, the synthesis rule for tasks is applied iteratively, which results in decomposition of the states that have been created for the object nodes in Fig. 1.

The decomposition of state dispatched in Fig. 2 is more complex than the other states because of activity send bill, where the coordinator o:Order uses a read-access rather than update-access. Since send bill creates object bill, the synthesis rule for creation needs to applied. Since the coordinator uses a read-access, a slightly modified version of the rule in Fig. 3 is applied: $S1$ and $S2$ of coordinator $:O$ are substates of the composite state S of $:O$, similar to the rule defined in Fig. 7. State init inside dispatched

in Fig. 2 resembles state $S1$ of object $:O$ while send bill completed resembles $S2$. Next, state send bill completed is decomposed in order to invoke task receive arrival info, which needs to be executed after send bill.

4.6.3 Multiple start states

If an object life cycle has multiple start states that are created by parallel activities, the synthesis approach will create multiple create actions. This results in multiple objects rather than a single object with parallelism.

We consider multiple start states as a design error: two parallel activities that create the same object $:O$ suggest on the one hand that $:O$ is created synchronously while on the other hand the two activities execute independently from one another. This error can be repaired by merging the activities that create the object. Another option is to extend the object life cycle with a new, unique start state from which the old start states can be reached.

4.6.4 Multiple variables

The translation does not consider two different objects that reference the same object type but have different states. One option is to view such objects as two orthogonal aspects of the same object life cycle. Another option is to view the objects as independent subclasses of the object type, each subclass having its own life cycle. We defer further analysis of such models to future work.

5 From state machines to GSM schemas

We define a mapping from state machines constructed by the rules in Sects. 3 and 4 to GSM schemas, a recent proposal for declarative, artifact-centric business process management [7,19]. We introduce first the basics of GSM schemas, next the mapping, and finally the most salient features of the mapping.

5.1 GSM schemas

The core ingredients of a GSM schema are stages and milestones [19]. A milestone specifies an operational business objective. A stage specifies a business activity performed to achieve a milestone. If a stage has multiple milestones, at most one of them can be achieved at a time [19]. The GSM version we consider in this paper has been published in citations [12,39] and allows milestones that are disconnected from stages. A GSM schema contains the following kind of rules:

- for each stage *guard rules* that specify when the stage is opened and *termination rules* that specify when the stage is closed; and
- for each milestone *achieving rules* that specify when the milestone gets achieved and *invalidation rules* that specify when the milestone gets invalidated.

Each rule has the syntax **ON** e **IF** $condition$ where e is an event and $condition$ is a Boolean expression, possibly true. Both the **ON**-part and the **IF**-part are optional.

Each change of a stage or milestone generates an event that can be referenced in rules. Event prefixes + and −indicate the kind of change: for milestone m event $+m$ occurs if m gets achieved and $-m$ occurs if m gets invalidated, and for stage S event $+S$ occurs if S gets opened while $-S$ occurs if S gets closed.

5.2 Mapping

To simplify the definition, we first identify two types of states for state machines constructed using the mapping defined in Sects. 3 and 4.

- In a *busy state* the system waits for an activity to complete. Each busy state does not contain any other states and has a label starting with "busy".
- An *object state* corresponds directly to an object node in the activity diagram. An object state either contains no other states or is composite, without having any parallel branches.

For instance, in the state machine of Bill (Fig. 2), state busy receiving payment is a busy state, while composite state unpaid and state paid are object states. The other states are neither busy nor object states.

By definition, a busy state is not an object state and vice versa. Based on this classification, we map state machines to GSM schemas. Object states are similar to milestones while busy states are similar to stages. We exploit this correspondence in the mapping defined below. However, not all properties of a generated state machine can be captured in a GSM schema. By construction, a busy state is inside an object state (cf. Fig. 8). In GSM schemas, however, a stage cannot be nested inside a milestone, and therefore the state machine hierarchy cannot be encoded directly in a GSM schema.

We now define a mapping from state machines as constructed using the rules defined in Sects. 3 and 4 to GSM schemas. We define the set of stages and milestones as well as the rules that specify when the stages and milestones change status.

5.2.1 Stages

Each busy state in which task T is performed, maps into a stage S_T in which T is launched. By construction (cf. Fig. 8), the parent of the busy state is an object state that maps into a milestone m, defined below. The guard of S_T becomes "**ON** $+m$". If the busy state follows a decision (cf. Fig. 9), the guard of S_T is appended with "**IF** *cond*", where *cond* is the condition of the decision branch in the state machine that leads to the busy state for T. The termination rule of S_T becomes "**ON** $cpl(T)$".

For example, consider busy state busy completing order in Fig. 2, which is contained in object state open. We create a stage complete order, whose guard rule is **ON** +open, for milestone open, and whose terminating rule is **ON** cpl(complete order).

5.2.2 Milestones

We define two sets of milestones with achieving and validating rules. For the first set, each state s that has an incoming transition t triggered by a completion event $cpl(T)$,

where T is a task, maps to a milestone m_s. State s is either an object state (cf. Fig. 8) or a state containing parallel branches (cf. Fig. 11). An achieving rule "**ON** $-S_T$" is defined for the milestone m_s, so when the stage performing T is closed, the milestone is achieved.

For example, for object state finalized in Fig. 2 a milestone finalized is created, whose achieving rule is **ON** −complete order, since the incoming transition of state finalized has trigger cpl(complete order).

If s has an outgoing transition with trigger event $cpl(T)$ for some task T, then the target state of the transition maps into a milestone m, as defined in the previous paragraph. The invalidating rule of m_s becomes "**ON** $+m$". For example, the invalidating rule for milestone open becomes **ON** +finalized. Otherwise, if s has no outgoing transition with trigger event $cpl(T)$ but is child of an object state for which milestone m is created, milestone m_s becomes invalid if parent milestone m becomes invalid, so the invalidating rule of m_s becomes "**ON** $-m$". For instance, for state send bill completed in Fig. 2 milestone send bill completed is created, whose parent milestone is dispatched. Therefore the invalidating rule for send bill completed becomes **ON** −dispatched. As another example, for the milestones m_{S2} and m_{S4} created for Fig. 12, the invalidating rule becomes in both cases **ON** $-m_c$, where m_c is the milestone created for composite state c in the figure. In all other cases, no invalidating rule is specified for m_s.

For the second set, each object state s that has no incoming transition triggered by a completion event maps into a milestone m_s. Figure 11 contains two such object states: S2 and S3, which are both the initial states of a composite state that is entered by a transition triggered by a completion event. Next, S2 and S3 are in parallel. By definition of the first set of milestones, such a composite state c maps to milestone m_c. In that case, the achieving rule of m_s becomes "**ON** $+m_c$", so if m_c gets achieved, also m_s gets achieved.

Alternatively, the object state s can have an incoming transition triggered by a generated (non-completion) event e (cf. Fig. 6). In that case, the achieving rule for milestone m_s becomes "**ON** e". For example, for the Shipment object in Fig. 2 the achieving rule for milestone being shipped becomes **ON** leave.

Otherwise, the object state is the start state of the life cycle and the achieving rule is trivially achieved by rule "**IF** true".

The definition of the invalidating rule for milestone m_s is similar to the one for milestones from the first set.

5.2.3 Example

Figures 13 and 14 show the stages and milestones and their rules for the Order artifact of Fig. 2 that are obtained by applying the translation defined above. Milestone send bill completed has a different type of invalidating rule than the other milestones: the corresponding state send bill completed does not have an outgoing transition triggered by a completion event, but is contained inside an object state for which milestone dispatched is created. All milestones are from the first set, except open which belongs to the second set.

Stages	Guard rules	Terminating rules
complete order	**ON** +open	**ON** cpl(complete order)
send shipment	**ON** +finalized	**ON** cpl(send shipment)
send bill	**ON** +dispatched	**ON** cpl(send bill)
receive arrival info	**ON** +send bill completed	**ON** cpl(receive arrival info)
close order	**ON** +delivered	**ON** cpl(close order)

Fig. 13 Stages with their rules for Order artifact

Milestones	Achieving rules	Invalidating rules
open	**IF** true	**ON** +finalized
finalized	**ON** -complete order	**ON** +dispatched
dispatched	**ON** -send shipment	**ON** +delivered
send bill completed	**ON** -send bill	**ON** -dispatched
delivered	**ON** -receive arrival info	**ON** +closed
closed	**ON** -close order	

Fig. 14 Milestones with rules for Order artifact

5.3 Discussion

We next discuss the most salient features of the translation.

5.3.1 Refining

The constructed GSM schema is a skeleton that is to be further refined by adding data attributes as well as additional stages, milestones and rules, for instance to incorporate human-centric behavior. To illustrate, suppose the company of the order process wishes to allow that a customer cancels a finalized order that has not yet been paid by rejecting a sent bill. This means that an external cancellation event can occur which needs to be responded to. Modeling the response by extending the global process model of Fig. 1 results in a complex diagram with a lot of additional edges. In the GSM schema, only a few local changes are required to model the response: extending the life cycle of the Bill and Order artifacts with additional milestones that model cancellation plus new rules that specify the intended response if the cancellation event occurs.

5.3.2 Creation and communication

The GSM approach does not support explicit create and communication actions. Rather, GSM tasks are responsible for creating new artifacts and generating events that are sent to other artifacts [20]. The tasks in the object-centric design of Fig. 2 do not have this functionality. Consequently, the GSM schema for Order of which the core elements are specified in Figs. 13 and 14 is less detailed than the object-centric model for Order in Fig. 2, at the expense of more complex GSM task implementations to realize the creation of artifacts and generation of events. For instance, a Shipment object is created by the Order object in the design of Fig. 2, but in a GSM implementation task Complete order should be responsible for creating the Shipment artifact instance and returning an ID for this instance. Likewise, the generation of events leave, arrive and finalize by Order for Shipment in Fig. 2 must be implemented by GSM tasks.

5.3.3 Direct vs. indirect mapping

A natural question is whether a direct mapping from activity diagrams to GSM schemas would not be preferable. An obvious mapping would be to translate every activity with n outgoing object nodes into a stage with n milestones attached. However, the meaning of an activity in UML [41] is that it simultaneously produces n different objects, while milestones attached to a stage are exclusive [19], i.e., at most one milestone at a time can be true. Another mismatch is that an activity can produce different types of objects (artifacts), while a stage that encapsulates a task is always local to one artifact. These two mismatches between activity diagrams and GSM schemas considerably complicate defining a direct mapping from activity diagrams to GSM schemas.

Using an intermediate state machine model helps to resolve the two mismatches as follows. The synthesis rules for fork and join nodes in Sects. 4.4 and 4.5 introduce composite states containing parallel branches; these states translate into milestones. For instance, the composite state in Fig. 11 translates into a milestone m, since it has an incoming transition triggered by a completion event. Milestone m belongs to the stage in which task A is launched. The object states S2 and S3 translate into milestones from the second set, which get achieved if the milestone m gets achieved; they do not belong to any stage. This resolves the first mismatch. The second mismatch is resolved by the synthesis rule for tasks, which allocates an activity to the coordinator of the activity.

6 Implementation

To evaluate the feasibility of the approach, we implemented the synthesis rules in a prototype. For the implementation, we build upon previous work on the transformation of Petri nets to UML state machines [43]. In turn, that line of work builds upon GrGen.NET [14], a generic graph transformation infrastructure not specific to BPM. In this section, we describe the overall architecture of our implementation. Appendix provides a gentle introduction to the source code, to stimulate the reader to leverage our open source materials.

Figure 15 provides a dynamic view on the architecture of our implementation. Each thick arrow represents a step in the transformation process. At the most abstract level, the figure shows that the human modeler only interacts with a professional UML editor; all processing steps are automated.

Step ① represents the step of exporting the input UML activity diagram to a standard format. Our current implementation supports XMI based on the UML metamodel in version 2.2 from the Object Management Group. All example models have been edited using MagicDraw 16.6 but other tools conforming to the standard can be used too. Steps ② to ⑨ are fully automated. Step ② consists of a simple syntactic translation. The output of this step is an input script for GrGen.NET. In Step ③, this script is executed by the GrGen.NET shell interpreter. This results in a graph representation of the UML activity diagram. The graph is typed according to the various elements from the UML metamodel. The type graph is used to check among others whether cardinality constraints are satisfied in the input model.

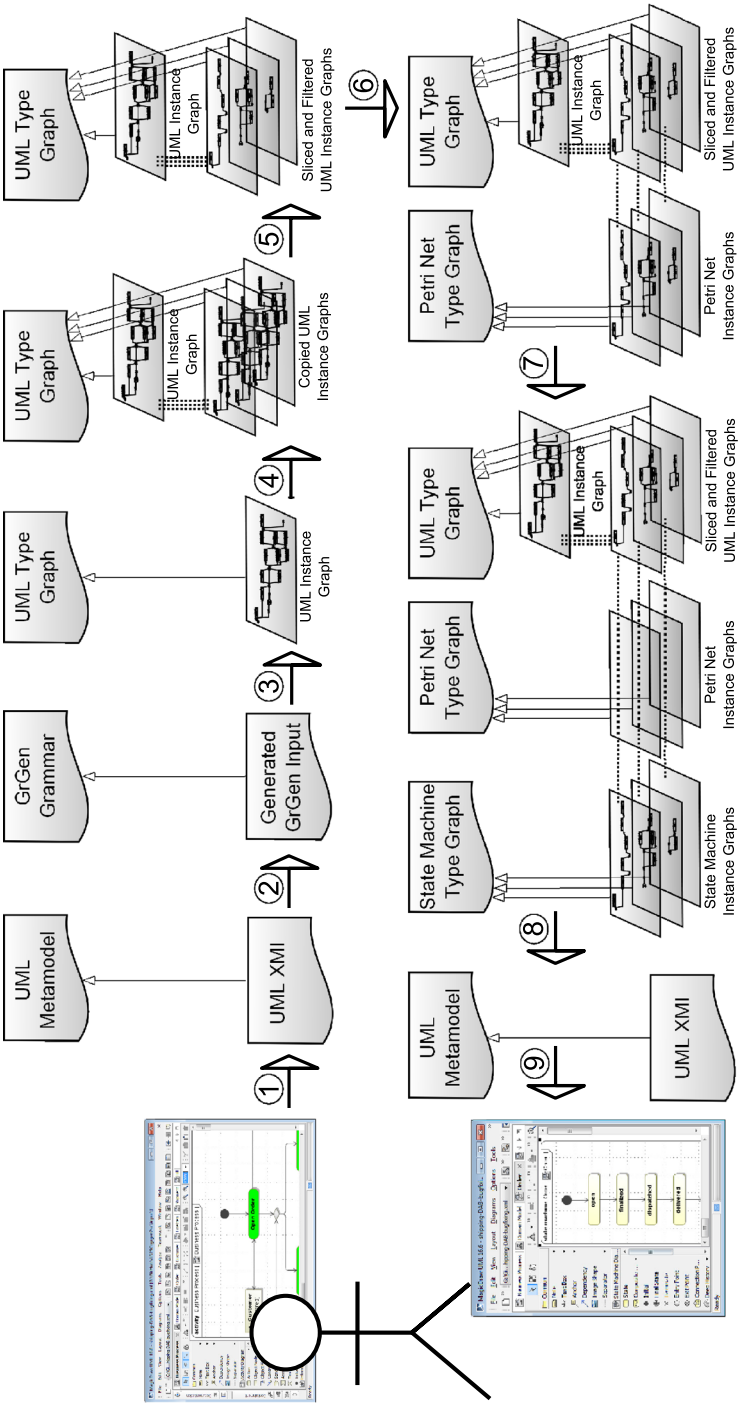

Fig. 15 Architecture of the Implementation

Step ④ is the start of the conceptual translation work. In this step, the graph representing the activity diagram is cloned once for each object type. Also, traceability links are generated between source and target elements. In Fig. 15, traceability links are sketched as dotted edges. For the figure, we assume three object types, resulting in three copied UML instance graphs. The GrGen.NET platform provides checks for ensuring that metamodel constraints are preserved throughout the cloning process. Finally, note that right before the cloning, it should be decided which object types act as the Coordinators of the various activities. Our implementation can mark valid Coordinators automatically or it can let an expert select among valid candidates. Figure 15 does not show that since our prototype has no end-user oriented UI widgets for this optional interaction step. For industrial settings, one may build a user-friendly UML tool plug-in for the Coordinator selection.

In Step ⑤, the cloned activity diagrams are sliced per object type: first of all, only object nodes of one specific type are preserved per slice. Additionally, filtering rules remove gateways that are irrelevant for the sliced object life cycle. The filtering rules are reused from [9] (since they were also needed for process flows involving just one object type) while the cloning and slicing rules are new for this paper.

The synthesis of filtered activity diagrams to state machines, defined in earlier work [9], is implemented by first mapping activity diagrams to Petri nets (Step ⑥) and next applying in Step ⑦ Petri-net reduction rules that construct a state machine with state hierarchy [8,43] to the generated Petri nets. This results in a valid state machine for a Petri net if the net has been reduced to exactly one place. As elaborated in previous work [42], the tool chain generates end-to-end traceability links (from activity diagram elements to state machine elements), which is non-trivial given the destructive effect of the Petri-net reduction rules that construct state hierarchy [8]. The reduction rules can deal also with semi-structured processes involving cross-synchronizations between concurrent regions.

After the reduction rules [8,43] have completed, Step ⑦ performs additional substeps specific to this paper. First, it annotates the state machines with *trigger*, *guard* and *effect* (action) labels. This is realized by rules that implement the new coordination-logic rules from Sect. 3. Finally, rules are executed for realizing the execution-logic rules from Sect. 4, e.g. inserting nested flows.

In Step ⑧, the transformation chain dumps the generated state machines to a text file. More specifically, the state machine instance graphs are mapped to text fragments based on the UML and XMI standards. In Step ⑨, the human expert opens that XMI file in a professional UML editor. Tools such as MagicDraw implement automatic layout algorithms that perform well for life cycle models, since these tend to be simple in structure.

A strength of our architecture is its modularity: we incrementally build upon prior work and we have avoided changes to the reused parts. As indicated above, the implementation described in citation [43] has been reused in Step ⑥ and in the first phase of Step ⑦. The second phase of that latter step involves new rules (regarding Sects. 3 and 4) but these additions are incremental. As a second example of modularity, the filtering rules defined in citation [9] are applied without changes in the second phase of Step ⑤. The new cloning rules (cf. Step ④) and filtering rules (cf. phase two of Step ⑤) have been added incrementally to the transformation chain of prior work [9,43].

In future work, we plan to implement the translation defined in Sect. 5. The target language will be CMMN [5]. Since the GSM constructs are directly supported in CMMN, we do not expect any implementation issues.

7 Related work

As stated in the introduction, in the last years a lot of research has been performed in the area of data-centric process modeling approaches such as business artifacts [18,21,29,46], case management [3,40], data-driven process models that are executable [22,28,33] and process models with data flow [25,27,38,44]. Sanz [37] surveys previous work on integrating the data and process perspective in the field of entity-relation modeling in connection to data-centric process modeling. This paper uses UML activity diagrams with object flows as data-centric process modeling notation but we also show how the resulting object-centric designs can be mapped to declarative, artifact-centric schemas [19].

We first discuss in detail existing transformation approaches between activity-centric and data-centric process models. Next, we discuss modeling approaches related to the ones used in this paper.

7.1 Transformation approaches

Related to this paper are approaches that distinguish between process and data models and bridge the gap by deriving a process model that is coherent with a predefined data model [17,35] or object behavior model [13,23,32]. This paper takes the opposite route: it considers a process model with data (object) flow and derives object behavior models that realize the process model.

More related are a few works [21,24,26,44] that define a translation between activity-centric and object-centric process models, which consist of communicating object life cycles. As the process models in this paper, activity-centric process models manipulate stateful business objects, where each step in a process model can lead to a change in one or more business objects. The main differences are therefore in the considered data-centric process models. Both Kumaran et al. [21,24] and Meyer and Weske [26] use synchronized object life cycles, which are flat, sequential state machines that synchronize on shared activities. Event communication between objects and task invocations are not specified by these state machines. In contrast, the object life cycles (UML state machines) generated by our approach are hierarchical, allow parallelism, deal with task invocations, and use asynchronous event communication, in line with UML [41].

Because the modeling notation for object life cycles is more complex, also the mapping to object life cycles defined in this paper is more involved than the other mappings [21,24,26], which are specified by relatively simple algorithms. We use synthesis rules in Sects. 3 and 4 to avoid defining a single, complex algorithm. Kumaran et al. [21,24] focus on the problem how to identify the objects in the process model for which an object life cycle should be constructed, while we construct an object life cycle for every stateful object. Like us, Kumaran et al. derive object life cycles

from the syntax of process models, whereas Meyer and Weske [26] construct an object life cycle by processing each trace of a process model. Disadvantage of a trace-based mapping is that the same nodes (activity or object nodes) can occur in multiple traces, resulting in object life cycles with duplicate states.

Next, Meyer and Weske [26] also consider an artifact-centric process model consisting of business rules. Each rule consists of a pre and postcondition on objects plus a set of tasks that manipulate objects to achieve the postcondition. Such a rule roughly corresponds to a GSM stage whose guard is the precondition and a GSM milestone that is achieved when the postcondition is fulfilled.

Wahler and Küster [44], based on earlier work [36], use as target implementation model a static set of predefined business objects that need to be "wired" together, where the activity-centric process model is used to derive the wiring relation. They study how to design the wiring in such a way, by changing the process model, that the resulting wired object design has a low coupling. In contrast, this paper studies the problem of deriving an object-centric design from a process model with object flows. The problem is then defining the set of business objects and their behavior, which are both given in the approach of Wahler and Küster.

To the best of our knowledge, there is only one other paper that discusses a translation into GSM schemas. Popova and Dumas [31] define a mapping from classical Petri nets to GSM schemas. They focus on tasks, modeled by Petri net transitions, and do not consider object flows. Therefore their mapping is not applicable to the UML activity diagrams we consider in this paper. Their translation is complicated by the presence of invisible transitions in Petri nets, which do not occur in activity diagrams and state machines. Each visible Petri net transition maps to a stage with an accompanying milestone that is achieved when the stage completes. Places are not used in the translation. To mimic the token-flow of Petri nets faithfully in GSM schemas, they need to use complex rules. For example, if a Petri net transition consumes a token that can also be consumed by another transition, then in the guard of the created stage time stamps of milestone changes are compared. Our guard rules for stages are much more simple due to the more structured syntax of UML state machines: each transition connects two states that cannot be simultaneously active. Consequently, achieving the milestone created for the target state of a transition invalidates the milestone created for the source state, as illustrated in Fig. 14 in Sect. 5. Drawback of UML state machines is that unlike Petri nets, they cannot directly express cross-synchronization between parallel branches of an object life cycle. In previous work [11], we have defined a specific translation rule based on event synchronization for such cases.

Bhattacharya et al. [4] propose a data-centric design methodology in which GSM-like schemas are constructed step by step. The focus is on defining different life cycles for different artifacts from scratch, which may complicate for stakeholders the understanding of the overall default behavior. The approach we propose complements theirs, since it allows to develop a global process model that specifies the default behavior of the artifacts (objects). The mapping defined in Sect. 5 can be used to generate starting definitions for the artifacts, which can be further refined by applying the methodology of Bhattacharya et al. [4].

7.2 Modeling techniques

The first part of the approach is defined in the context of UML activity diagrams [41], which model business processes in an imperative, activity-centric way. The synthesis rules of Sects. 3 and 4 are directly applicable to any imperative process modeling language that uses a similar semantics for object flows as UML activity diagrams; for instance, an activity can only start if its input objects are present, so read-access is mandatory [34].

BPMN [30] supports a similar object-flow notation as UML activity diagrams, but the BPMN semantics is different for some models. UML uses a token-flow semantics for objects. For instance, if an object node is read via an outgoing object flow, the object (token) moves to the target of the object flow and is no longer available for other outgoing object flows of the object node, as explained in Sect. 2. Whereas BPMN uses a copy semantics for objects: if available, the input object of a BPMN object flow, called data association, is copied to the output [30]. This implies that if a BPMN object node, called data object, is read via an outgoing data association, the read object remains available for the other outgoing data associations. So while a UML object node with multiple outgoing object flows specifies exclusive behavior, a BPMN data object with multiple outgoing data associations does not. However, if in a BPMN model every data object has at most one incoming and one outgoing data association, its behavior is similar to an activity diagram. The rules defined in this paper can be applied to such BPMN models with little modification, only replacing the UML activity diagram concepts by their BPMN counterparts.

As alternative for GSM schemas, case management [3,40] or object-aware process management [22] could be used as target framework. A discussion of the similarities and differences between these approaches is outside the scope of this paper, but Reichert and Weber present an overview [34]. Note that all these data-centric frameworks allow for much more fine-grained behavior than can be expressed in UML activity diagrams [41] or related imperative process modeling notations like BPMN [30]. For instance, enabling of an activity can depend on specific attribute values of multiple, related object instances. Such fine-grained behavior can be added by refining the initial data-centric process design obtained by applying the synthesis approach.

Most data-centric process management approaches [3,19,22] also allow to design process models using an activity-centric style. For instance, it is possible in GSM schemas to use a flow-style modeling that resembles imperative, activity-centric modeling [19]. The default behavior of a business process can therefore also be modeled directly using a data-centric process modeling approach instead of UML activity diagrams. However, these data-centric approaches were never intended to develop imperative, activity-centric models: their strength is the flexible specification and execution of data-centric processes. While the strength of imperative process modeling techniques like UML activity diagrams is that they clearly and concisely specify default behavior of a business process. For this reason, we propose to model the default behavior of processes in an imperative, activity-centric way and have defined an approach to transform them in data-centric process models. The resulting data-centric process designs can be further refined and extended to deal with anticipated exceptional circumstances, which is the strength of data-centric process management approaches [3,19,22].

Finally, Meyer et al. [27] take a different approach to solve the tension between activity-centric and data-centric process models. Based on a subset of requirements identified by Künzle and Reichert [22], they address shortcomings of BPMN in the area of data-centric process management. In particular, they introduce BPMN annotations to specify data-centric behavior and define SQL queries that realize these annotations. While their approach is supported by state-of-the-art process- and data-management technology, it does not support the full range of options supported by data-centric approaches [3,19,22].

8 Conclusion

We have presented a semi-automated approach that synthesizes an object-centric system design from a business process model that references multiple objects. The approach distinguishes between coordination-logic synthesis rules that realize object-flow constraints and execution-logic rules for control-flow constraints. The rules heavily use the state hierarchy for the object life cycles to establish a clear link with the process model constructs. We have implemented and tested the rules in a prototype based on graph-transformation technology [14].

The resulting object-centric design can be used to perform the process in a flexible way [32]. We showed this by mapping object life cycles to Guard-Stage-Milestone schemas, which have inspired the emerging OMG standard on case management (CMMN) [5]. We expect the proposed translation carries over directly to CMMN. Future work includes connecting the prototype to CMMN tools.

The approach is defined in the context of UML [41], but we plan to define a similar approach for BPMN [30]. As explained in Sect. 7, BPMN uses a similar object-flow notation as UML activity diagrams but with a different semantics in certain cases.

The approach can also be extended to deal with more advanced UML activity diagram constructs, for instance activities that can be instantiated multiple times in parallel or iteratively to process a collection of objects. Object-aware frameworks like PHILharmonicFlows [22] support such functionality, but also allow much more fine-grained synchronization among multiple objects and activities. In line with design method proposed in Sect. 1, such fine-grained synchronization can then be added in a refinement step.

In this paper, we envision that the data-centric process design generated using the approach can be further refined by specifying responses to exceptional circumstances. This does imply that these circumstances need to be anticipated [34]. For imperative processes, adaptive process management technology has been proposed to deal with unanticipated events at run-time [34]. Though adaptive process management still needs to be developed for data-centric processes [22,34], we do expect that it can deal in the future with unanticipated events for data-centric processes.

Appendix: Source code and online demonstrator

We provide open access to the source code of the prototype via GitHub.[1] That online repository provides access to sources for steps ② to ⑧ from Fig. 15. In particular, folder *ad2grgen* contains a Java based tool for step ②. Folder *grgen2sc* provides rules and transformation scripts for the GrGen.NET platform [14]. Source file *UML23AD.grg* contains the heart of the implementation. In the following, we walk the reader through the implementation of one specific rule that is defined in that file. That should provide sufficient insight for exploring the other rules online. Since installing the GrGen.NET platform may be a burden to many readers, we also provide an online demonstrator in the form of a remotely accessible virtual machine.[2]

```
1 rule Add_GuardsTriggersAndEffectsForCoordinator {
2   c:UmlClass <-:smType- cbnS1:CentralBufferNode -:ObjectFlow-> a:ActivityNode -:ObjectFlow
        -> cbnS2:CentralBufferNode -:smType-> c;

3   a -:Coordinator-> c;
4   negative { a -:VisitedEdge-> c; }
5   cbnS1 -:AN2SC-> s1:SC_State;
6   cbnS2 -:AN2SC-> s2:SC_State;
7   s1 -trans:SC_Transition-> s2;
8   iterated {
9     alternative {
10      UFR { // Generic for Update/Finalize/Read
11        cbnT1:CentralBufferNode -:ObjectFlow-> a;
12        scbnT1:State <-:inState- cbnT1 -:smType-> c2:UmlClass;
13        cbnT1 -:AN2SC-> t1:SC_State;
14        negative { cbnT1 -:VisitedEdge-> a; }
15        alternative {
16          Update { // Update Rule
17            a -:ObjectFlow-> cbnT2:CentralBufferNode -:smType-> c2;
18            cbnT2 -:inState-> scbnT2:State;
19            cbnT2 -:AN2SC-> t2:SC_State;
20            t1 -trans2:SC_Transition-> t2;
21            modify {
22              eval {
23                trans.effect= cbnT2.name+''.to''+scbnT2.name+'' ''+trans.effect;
24                trans2.trigger= ''to''+scbnT2.name+'' ''+trans.trigger;
25              }
26            }
27          }
28          Finalize { /* Finalize Rule omitted from paper */ }
29          Read { /* Read Rule: no extra modify */ a -:ObjectFlow-> cbnT1; }
30        }
31        modify {
32          cbnT1 -:VisitedEdge-> a;
33          eval {
34            trans.guard=''[''+cbnT1.name+''.in(''+scbnT1.name+'')] ''+trans.guard;
35          }
36        }
37      }
38      Create { // Create Rule
39        a -:ObjectFlow-> cbnInit:CentralBufferNode;
40        cbnInit -:smType-> typeOf_cbnInit:UmlClass;
41        negative {
42          cbnOther:CentralBufferNode -:ObjectFlow-> a;
43          cbnOther -:smType-> typeOf_cbnInit;
44        }
45        modify {
46          eval {
47            trans.effect=cbnInit.name+'':= create(''+cbnInit._type+'')'';
48          }
49        }
50      }
51    }
52  }
53  modify {
54    a -:VisitedEdge-> c;
55  }
56 }
```

The listing defines a graph transformation rule named "Add_GuardsTriggersAndEffectsForCoordinator". The ruleshould be executed as long as possible during the

[1] See https://github.com/pvgorp/AD2SC.

[2] See http://share20.eu/?page=ConfigureNewSession&vdi=Win7_AD2SC_i.vdi.

second phase of Step ⑦ in the transformation chain (cf. Fig. 15). The termination of the rule is handled by requiring in its matching part the absense of an edge (cf. line 4) of type *VisitedEdge* while generating exactly such a link in the side-effects part of the rule (cf. line 54). The example rule realizes the full behavior of the Coordinator synthesis rules from Sect. 3. Below, we further clarify the transformation language syntax by example. A comprehensive manual of the GrGen.NET syntax and execution environment is available online.[3]

Line 2 declares via "c:UmlClass" a node variable *c* of type *UmlClass*. Line 3 specifies that this node should be the Coordinator for activity *a*, by requiring that there is a special edge from *a* to *c* (the expression "$-:Coordinator->$" defines an anonymous variable for a directed edge of type *Coordinator*). The creation of that edge is handled by another rule, which is executed during Step ④ in in the transformation chain (cf. Fig. 15). Line 3 only checks for the existence of the edge. Line 2 also states that *c* is the type of two object nodes: *cbnS1* (which has an outgoing object flow to activity *a*) and *cbnS2* (which has an incoming object flow starting from *a*). Together, lines 2 and 3 specify textually a rule that formally represents the shared parts of Figs. 3, 4, 5 and 6.

Lines 5 to 7 further specify the matching part of the rule. The expressions "$-:AN2SC->$" require the presence of traceability links from the two object nodes in the activity diagram to their corresponding state machine elements (two nodes of type SC_State). At line 7, the name "$trans$" is bound to the edge variable that represents the transition between these states.

A detailed explanation of the remainder of the code is omitted for the sake of brevity. Do note that via constructs such as *iterated*, the rule is applied maximally to the host graph, ensuring that it also works in case multiple update rules apply to the same UML activity. Also note that via the *alternative* construct, the variation between the different Coordination synthesis rules (cf. Figs. 3, 4, 5 and 6) is handled without duplicating code. Finally, note that all expressions within a *modify* clause realize side-effects. Within the example code, most side-effects relate to updating the labels within the state machine graph. For example, the expression on line 47 updates the *effect* attribute of the *trans* variable that was matched via the rule pattern on line 7.

References

1. van der Aalst WMP, van Hee KM (2002) Workflow management: models, methods, and systems. MIT Press, USA
2. van der Aalst WMP, ter Hofstede AHM, Kiepuszewski B, Barros AP (2003) Workflow patterns. Distrib Parallel Databases 14(1):5–51
3. van der Aalst WMP, Weske M, Grünbauer D (2005) Case handling: a new paradigm for business process support. Data Knowl Eng 53(2):129–162
4. Bhattacharya K, Hull R, Su J (2009) A data-centric design methodology for business processes. In: Handbook of Research on Business Process Management, Information Science Publishing, pp 503–531
5. Bizagi et al. (2013) Case Management Model and Notation (CMMN). Object Management Group, OMG Document Number dtc/2013-01-01

[3] http://www.info.uni-karlsruhe.de/software/grgen/GrGenNET-Manual.

6. Coalition WM (1997) Workflow management coalition workflow client application (interface 2) application programming interface (WAPI) specification
7. Damaggio E, Hull R, Vaculín R (2013) On the equivalence of incremental and fixpoint semantics for business artifacts with guard-stage-milestone lifecycles. Inf Syst 38(4):561–584
8. Eshuis R (2009) Translating safe petri nets to statecharts in a structure-preserving way. In: Cavalcanti A, Dams D (eds) FM. Lecture Notes in Computer Science, vol 5850. Springer, pp 239–255
9. Eshuis R, Van Gorp P (2012) Synthesizing object life cycles from business process models. In: Atzeni P, Cheung DW, Ram S (eds) Proceedings of ER 2012. Lecture Notes in Computer Science, vol 7532. Springer, pp 307–320
10. Eshuis R, Van Gorp P (2014a) Synthesizing object-centric models from business process models. In: Lohmann N, Song M, Wohed P (eds) Proceedings of Business Process Management Workshops 2013, Revised Papers. Lecture Notes in Business Information Processing, vol 171. Springer, pp 155–166
11. Eshuis R, Van Gorp P (2014b) Synthesizing object life cycles from business process models. Softw Syst Model. doi:10.1007/s10270-014-0406-4 (in press)
12. Eshuis R, Hull R, Sun Y, Vaculín R (2014) Splitting GSM schemas: A framework for outsourcing of declarative artifact systems. Inf Syst 46:157–187. An early version appeared. In: Daniel F, Wang J, Weber B (eds) BPM 2013. Lecture Notes in Computer Science, vol 8094. Springer, pp 259–274
13. Fritz C, Hull R, Su J (2009) Automatic construction of simple artifact-based business processes. In: Fagin R (ed) ICDT, ACM, ACM International Conference Proceeding Series, vol 361, pp 225–238
14. Geiß R, Kroll M (2008) GrGen.NET: A fast, expressive, and general purpose graph rewrite tool. In: Rensink A, Täntzer G (eds) AGTiVE 2007, Kassel, October 10–12, 2007, Revised Selected and Invited Papers, Springer, LNCS, vol 5088, pp 568–569
15. Gray J, Reuter A (1993) Transaction processing: concepts and techniques. Morgan Kaufmann, USA
16. Harel D, Gery E (1997) Executable object modeling with statecharts. IEEE Comput 30(7):31–42
17. van Hee KM, Hidders J, Houben GJ, Paredaens J, Thiran P (2009) On the relationship between workflow models and document types. Inf Syst 34(1):178–208
18. Hull R (2008) Artifact-centric business process models: Brief survey of research results and challenges. In: Meersman R, Tari Z (eds) OTM Conferences (2). Lecture Notes in Computer Science, vol 5332. Springer, pp 1152–1163
19. Hull R, Damaggio E, Fournier F, Gupta M, Heath FT, 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: Bravetti M, Bultan T (eds) WS-FM. Lecture Notes in Computer Science, vol 6551. Springer, pp 1–24
20. Hull R, Damaggio E, Masellis RD, Fournier F, Gupta M, Heath FT, Hobson S, Linehan MH, Maradugu S, Nigam A, Sukaviriya PN, Vaculín R (2011) Business artifacts with guard-stage-milestone lifecycles: managing artifact interactions with conditions and events. In: Eyers DM, Etzion O, Gal A, Zdonik SB, Vincent P (eds) ACM, DEBS, pp 51–62
21. Kumaran S, Liu R, Wu FY (2008) On the duality of information-centric and activity-centric models of business processes. In: Bellahsene Z, Léonard M (eds) CAiSE. Lecture Notes in Computer Science, vol 5074. Springer, pp 32–47
22. Künzle V, Reichert M (2011) Philharmonicflows: towards a framework for object-aware process management. J Softw Maint 23(4):205–244
23. Küster JM, Ryndina K, Gall H (2007) Generation of business process models for object life cycle compliance. In: Alonso G, Dadam P, Rosemann M (eds) BPM. Lecture Notes in Computer Science, vol 4714. Springer, pp 165–181
24. Liu R, Wu FY, Kumaran S (2010) Transforming activity-centric business process models into information-centric models for soa solutions. J Database Manag 21(4):14–34
25. Meyer A, Weske M (2012) Data support in process model abstraction. In: Atzeni P, Cheung DW, Ram S (eds) Proceedings of ER 2012. Lecture Notes in Computer Science, vol 7532. Springer, pp 292–306
26. Meyer A, Weske M (2014) Activity-centric and artifact-centric process model roundtrip. In: Lohmann N, Song M, Wohed P (eds) Proceedings of Business Process Management Workshops 2013, Revised Papers. Lecture Notes in Business Information Processing, vol 171. Springer, pp 167–181
27. 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) Proceedins of BPM 2013. Lecture Notes in Computer Science, vol 8094. Springer, pp 171–186

28. Müller D, Reichert M, Herbst J (2007) Data-driven modeling and coordination of large process structures. In: Meersman R, Tari Z (eds) OTM Conferences (1). Lecture Notes in Computer Science, vol 4803. Springer, pp 131–149
29. Nigam A, Caswell NS (2003) Business artifacts: an approach to operational specification. IBM Syst J 42(3):428–445
30. (OMG) OMG (2011) Business process model and notation (bpmn) version 2.0. Tech. rep
31. Popova V, Dumas M (2013) From Petri nets to guard-stage-milestone models. In: Rosa ML, Soffer P (eds) Proceedings of Business Process Management Workshops 2012, Revised papers, Lecture Notes in Business Information Processing, vol 132. Springer, pp 340–351
32. Redding G, Dumas M, ter Hofstede AHM, Iordachescu A (2008) Generating business process models from object behavior models. IS Manag 25(4):319–331
33. Redding G, Dumas M, ter Hofstede AHM, Iordachescu A (2010) A flexible, object-centric approach for business process modelling. Serv Oriented Comput Appl 4(3):191–201
34. Reichert M, Weber B (2012) Enabling Flexibility in process-aware information systems: challenges, methods, technologies. Springer, Berlin
35. Reijers HA, Limam S, van der Aalst WMP (2003) Product-based workflow design. J Manag Inf Syst 20(1):229–262
36. Ryndina K, Küster JM, Gall H (2006) Consistency of business process models and object life cycles. In: Kühne T (ed) MoDELS Workshops. Lecture Notes in Computer Science, vol 4364. Springer, pp 80–90
37. Sanz JLC (2011) Entity-centric operations modeling for business process management: a multidisciplinary review of the state-of-the-art. In: Lu X, Younas M, Zhu H, Gao JZ (eds) SOSE, IEEE, pp 152–163
38. Sun SX, Zhao JL, Nunamaker JF, Sheng ORL (2006) Formulating the data-flow perspective for business process management. Inf Syst Res 17(4):374–391
39. Sun Y, Hull R, Vaculín R (2012) Parallel processing for business artifacts with declarative lifecycles. In: Meersman R, Panetto H, Dillon TS, Rinderle-Ma S, Dadam P, Zhou X, Pearson S, Ferscha A, Bergamaschi S, Cruz IF (eds) OTM Conferences (1). Lecture Notes in Computer Science, vol 7565. Springer, pp 433–443
40. Swenson KD (2010) Mastering the unpredictable: how adaptive case management will revolutionize the way that knowledge workers get things done. Meghan-Kiffer Press, USA
41. UML Revision Taskforce (2010) UML 2.3 Superstructure Specification. Object Management Group, oMG Document Number formal/2010-05-05
42. Van Gorp P (2011) Applying traceability and cloning techniques to compose input-destructive model transformations into an input-preserving chain. In: 1st Workshop on Composition and Evolution of Model Transformations, King's College, London, UK
43. Van Gorp P, Eshuis R (2010) Transforming process models: executable rewrite rules versus a formalized java program. In: Petriu DC, Rouquette N, Haugen Ø (eds) MoDELS (2). Lecture Notes in Computer Science, vol 6395. Springer, pp 258–272
44. Wahler K, Küster JM (2008) Predicting coupling of object-centric business process implementations. In: Dumas M, Reichert M, Shan MC (eds) BPM. Lecture Notes in Computer Science, vol 5240. Springer, pp 148–163
45. Wang T, Vonk J, Kratz B, Grefen PWPJ (2008) A survey on the history of transaction management: from flat to grid transactions. Distrib Parallel Databases 23(3):235–270
46. Yongchareon S, Liu C, Zhao X (2011) An artifact-centric view-based approach to modeling inter-organizational business processes. In: Bouguettaya A, Hauswirth M, Liu L (eds) WISE. Lecture Notes in Computer Science, vol 6997. Springer, pp 273–281