Knowledge Representation and Reasoning Aspects of Structured BPMN

Alexander Borgida, Varvara Kalokyri, Amélie Marian

Department of Computer Science,
Rutgers University, New Brunswick, USA
{borgida,v.kalokyri,amelie}@cs.rutgers.edu

Abstract. The literature contains arguments for the benefits of representing and reasoning with BPMN processes in (OWL) ontologies, but these proposals are not able to reason about the dynamics of processes. We introduce a new Description Logic, REprocDL, to represent the behavioral semantics of *(block) structured BPMN*. It supports reasoning about process concepts based on their execution traces.
Starting from the notion of "subconcept", provided by the traditional notion of subsumption in Description Logics (including REprocDL), we further investigate the notions of specialization and inheritance, as a way to help build and abbreviate large libraries of processes in an ontology, which are needed in several applications.
We also provide *formal* evidence for the intuition that features of structured BPMN diagrams such as AND-gates and sub-processes can provide substantial benefits for their succinctness. The same can be true when moving from a structured to an equivalent unstructured version.

1 Introduction

We begin with some motivation for our work. Process descriptions can be used for *prescriptive* purposes, such as workflows enacted by computer programs, as well as for *descriptive* purposes, for understanding what actually happens during the execution of a given scenario. Our work [16] uses BPMN for the second purpose: to group and organize "personal digital traces" (byproducts of apps on smartphones, such as emails, web searches, SMS, check-ins, financial transactions, and GPS), helping to reconstruct the user's "memory of autobiographical events".

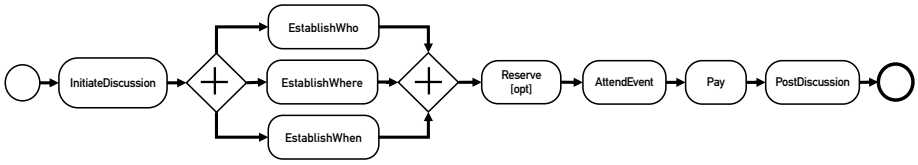


Fig. 1. *Process model for EatingOutAtRestaurant*

An example of such a process model ("script") would be going out to eat at a restaurant, as shown in Fig. 1. Components, such as EstablishWhere, are in turn

sub-process descriptions given elsewhere, involving a communication loop.This process model can help us organize various digital data into instantiations of restaurant outings. For instance, a thread of messages concerning dinner, an email confirmation of an OpenTable reservation, a credit-card payment at a restaurant, and a Facebook discussion of a meal, all occurring within an appropriate time/place frame, would provide evidence for the InitiateDiscussion, Reserve, Pay and PostDiscussion subprocesses respectively in Fig. 1. Such process instances can then help the user get answers to questions like "When did I go to restaurant X?", or "What restaurant did I got to with Mary in Paris?".

Although this may not look like a business process, arranging a meeting in a company is almost identical: instead of initiating a discussion to go out to eat, one must initiate a discussion for having a meeting; deciding who/when/where the activity will take place happens in both; instead of making a restaurant reservation one might need to reserve a room for the meeting; etc.

In order to make a fully functional system, we need process descriptions for thousands of variants of every-day activities. This will be facilitated by placing the process descriptions in an ontology, and then using specialization and inheritance, so that more specific processes, such as GoingOutToEat, GoingOutToTheater, GoingOutToSports, etc. are obtained by indicating only differences from a common superclass, GoingOutForEntertainment. In turn, GoingOutToEat can be specialized to EatingOutAtRestaurant, EatingFastFood, etc. The main distinction of our work from most prior attempts at using ontologies for processes [26, 25], is that we are interested in *reasoning with their behavioural semantics*, rather than just their annotated syntactic structure. For example, we want to be able to infer that "(Eating or Sleeping) followed by Driving" is related to "(Eating followed by Driving) or (Sleeping followed by Driving)".

The paper addresses these goals by (i) providing the Description Logic REprocDL to describe and reason with a subclass of BPMs, and (ii) introduces an improved notion of specialization and inheritance, which allows new actions to be added to subconcepts [6, 1] while still using the subsumption algorithm of REprocDL on a modified subsumee. The algorithms rely on results in the formal language literature.

2 Structured BPM and REprocDL

We consider a subset of BPMN, called *structured BPM/Workflow* [17, 2, 18], which has properly nested single-entry/single-exit blocks for (i) sequencing, (ii) branching (deferred choice **xor**), (iii) looping (either BPMN 2 loop marker on the body, or **xor**-join and **xor**-split enclosing the body, with return flows to the beginning), and (iv) concurrent execution (**and**). Although less expressive than its unrestricted version (see [17]), Structured BPMN has the advantages of:

- being more understandable and maintainable by humans (see suggestions for "good BPMN style" in the survey by Corradini et al [11]);
- being guaranteed not to have anomalies such as deadlock, livelock;
- having clearer semantics (avoiding the complications of global **xor**-join);

– conforming to the flow control structures offered by other process notations such as the OWL-S service model specification language [21];
– facilitating the mining of workflows [18];

while covering a large fraction of the practical applications we have encountered.

It has been noted [2] that a structured workflow block, corresponding to the sub-graph **xor**-split$\{P_1, P_2, ...\}$**xor**-join say, can be represented by a *process tree* with operator **xor** at its root and children $P_1, P_2, ...$, which we'll write as $\mathbf{xor}(P_1, P_2, ...)$, or the infix-notation $P_1 \sqcup P_2 \sqcup ....$ Similarly, structured workflows have operators **seq**, **and**, and **loop** for sequencing, concurrency and looping.

By treating atomic actions as alphabet symbols, one can see a clear similarity of process trees without **and** to regular expressions (REs) built with $\{ \cdot, +, ^* \}$ [1]. REs have been extended, among others, with an "interleaving/shuffle" operator $\#$ in order to model the notion of concurrency; this will be taken to correspond (in this paper) to **and** in BPMN.

Description Logics (DLs) are knowledge representation languages, which manage information about individuals grouped into classes, possibly related by (binary) relations. DLs are used to define concepts in ontologies, check them for consistency, and especially organize them in hierarchies, ordered by the subsumption relation. The family of DLs is distinguished by the fact the classes in question are intensional entities that have a compositional structure and are involved in inferences. Almost universally, *concepts* (unary predicates) and *roles* (binary predicates) are built using concept/role *constructors* from atomic identifiers. For example, the concept of "Building one of whose owner is crazy and rich" is represented in *variable-free term notation* as $\mathbf{conjoin}(Building, \mathbf{some}(owner, \mathbf{conjoin}(Crazy, Rich)))$, or, in the more familiar infix notation, as $Building \sqcap \exists owner.(Crazy \sqcap Rich)$. This syntax is in contrast with languages using variables and quantifiers (e.g., [27]), or general graphical notations (unless one can find an appropriate syntax for them).

We are now ready to define (see Figure 2) the DL-like ontology language REprocDL (standing for Regular Expression-based Process Description Logic) for representing structured BPMs. Its syntax starts from a finite set of atomic action names $\mathcal{A}$, and a disjoint set $\mathcal{P}$ of (sub)process names. There are some additional useful constant process concepts listed in the top part of the figure; the middle part gives the syntax rules for compound processes P. Please note that REprocDL has the same variable-free term notation as most other members of the family of Description Logics.

We can then express in REprocDL the EatingOutAtRestaurant process as

$$Initiate \cdot (EstablishWho \# EstablishWhen \# EstablishWhere) \cdot \text{Reserve}^{opt} \cdot AttendEvent \cdot Pay \cdot PostDiscussion$$

where we use an abbreviation for optional tasks, P^{opt}, equivalent to $P \sqcup \mathbf{Null}$.

[1] This can be used as another argument in favor of Structured BPM: several papers [12, 22] have shown how to represent the trace semantics of *declarative BPMs*, such as DECLARE, by mapping to regular expressions.

Name of constructor	Syntax	Term notation	Semantics
$N \in \mathcal{A} \cup \mathcal{P}$	N	N	$N^{\mathcal{I}}$
no-action process	$Null$	**Null**	$\{ \lambda \}$
bottom concept	$\bot_P$	$\mathbf{Bottom}_{Process}$	$\emptyset$
top-process concept	$\top_P$	$\mathbf{Top}_{Process}$	$Sequence(\Delta_{\mathcal{I}})$
any action	$Action$	**Action**	$\bigcup_{A \in \mathcal{A}} A^{\mathcal{I}}$
sequence	$P_1 \cdot P_2$	$\mathbf{seq}(P_1, P_2)$	$\{ uw \mid u \in P_1^{\mathcal{I}}, w \in P_2^{\mathcal{I}} \}$
alternation	$P_1 \sqcup P_2$	$\mathbf{xor}(\mathrm{P}_1,\mathrm{P}_2)$	$P_1^{\mathcal{I}} \cup P_2^{\mathcal{I}}$
repetition (base)	P^0	$\mathbf{repeat}(0,\mathrm{P})$	$Null^{\mathcal{I}}$
repetition (ind'n)	P^{k+1}	$\mathbf{repeat}(\mathrm{k+1},\mathrm{P})$	$P^{\mathcal{I}} \cdot (P^k)^{\mathcal{I}}$
loop	P^*	$\mathbf{loop}(\mathrm{P})$	$\bigcup_{i \geq 0} (P^i)^{\mathcal{I}}$
interleaving	$P_1 \# P_2$	$\mathbf{and}(P_1, P_2)$	$\{ u_1 w_1 ... u_n w_n \mid u_1 ... u_n \in P_1^{\mathcal{I}}, w_1 ... w_n \in P_2^{\mathcal{I}}$, for $n \in \mathbb{N}\}$
subprocess N definition	$N \doteq P$		$N^{\mathcal{I}} = P^{\mathcal{I}}$
P_2 subsumes P_1	$P_1 \sqsubseteq P_2$		$P_1^{\mathcal{I}} \subseteq P_2^{\mathcal{I}}$
inconsistency of P	$P \sqsubseteq \bot$		$P^{\mathcal{I}} = \emptyset$
membership of action sequence α in P	$\alpha \in P$		$\alpha \in P^{\mathcal{I}}$

Fig. 2. Syntax and Semantics of REprocDL

The semantics of REprocDL[2] is provided by an *interpretation* $\mathcal{I} = (\Delta_{\mathcal{I}}, \cdot^{\mathcal{I}})$, where $\Delta_{\mathcal{I}}$ is intended to be the set of atomic action executions/instances; $A^{\mathcal{I}}$ is a *finite subset of sequences of length* one over $\Delta_{\mathcal{I}}$ for $A \in \mathcal{A}$; in turn, $P^{\mathcal{I}}$ is a finite subset of the set $Sequence(\Delta_{\mathcal{I}})$ of *sequences of individual action executions*. $\mathcal{I}$ is then extended in the natural way to the constants and the process concept constructors in the manner shown in Fig. 2.

It is crucial to note that since the denotation of REprocDL concepts are *sequences of action instances*, this corresponds to the "trace semantics" for BPM/Petri Nets (as used in [28, 29, 12, 24]), rather than bisimulation-based semantics [6]. This means that in our model atomic actions can't be truly parallel; even if we replace A with $StartA \cdot EndA$, the beginnings of two actions must be ordered.

The bottom part of Fig. 2 describes the standard predicates used for inferences and axioms in the field of DLs. It is usual to consider the complexity of these inferences and their implementation. In this case, we rely on results about regular-like expressions in the formal language literature. In particular, let us treat atomic action concept names A$\in \mathcal{A}$ as "characters" in the alphabet Σ usually taken as the base case for regular expressions. Then the above predicates on regular languages will give the same results as the semantics in Fig. 2, since the interpretation $\mathcal{I}$ replaces all occurrences of A by the same set, $A^{\mathcal{I}}$. Therefore, deciding subsumption, inconsistency, and membership in REprocDL correspond exactly to deciding containment, emptiness, and string recognition for this class of extended regular expressions. A summary of results concerning the complexity

[2] This is a variant of the semantics of one of the plan DLs, $RegExp(\{ \cdot, \sqcup, ^*, \# \})$, described in [7, 9].

of these reasoning operations, some based on [19], appear in [7, 9]. We mention that the preceding references also show how REprocDL can be integrated with standard DLs, like OWL say, using the notion of "concrete domains" [4], thus obtaining more powerful hybrid but still DL-like formalism.

3 Specialization and Inheritance for Structured BPM

In software development and AI KR, inheritance is a mechanism for abbreviating descriptions based on the notion of IsA and specialization in class-based formalisms. For example, if *Employee* is a subclass of *Person*, then properties of *Person* such as *name*, *address*, *age*, etc. and constraints on them, are *inherited* by the subclass *Employee*. The purpose of inheritance is reuse (avoiding repetition of work), and effective maintenance: a change on *Person* (e.g., adding property *socialSecurityNr*), results in it being propagated to all subclasses.

When specializing a class, which has associated properties/features common to its instances, several principles have been observed in AI and conceptual modeling [8]:

A. Extensional IsA constraint: the instances of the subclass are instances of the superclass.

B. Unmodified inheritance: properties/features and their constraints are inherited unchanged to the subclass.

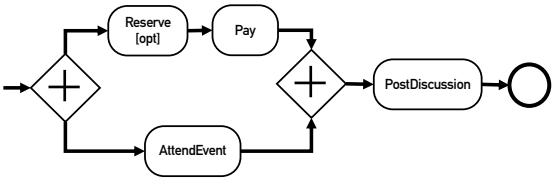
C. Adding restrictions to existing properties: an inherited property may have more restrictions placed on it. For example, while the *age* of *Person*s has lower bound 0, the *age* of *Employee*s must be at least 16.

D. Adding new properties: A subclass may have new properties defined over it. For example, *salary* is added for *Employee*.

Recently, another principle has come to dominate in object-oriented programming circles: the "substitution principle" requires that an instance of a subclass must be usable in any situation where an instance of a superclass could appear. This however is not crucial in our situation, since we are not writing software systems (which would use polymorphic typing), but process descriptions, where we try to *describe* variant sub-processes in some principled manner.

Specialization and inheritance are helpful whenever there are many similar classes, as in our case. In fact, already decades ago the Taxis project suggested that the benefits of specialization and inheritance were useful not just for describing entities but also transactions, exceptions, and scripts/workflows [20, 8, 5]. More recently, Frank [15] has provided an in-depth survey of the field of process specialization.

The question that arises is what we can do during the specialization of process descriptions. A crucial part of this is what counts as "properties/features" of processes. Obvious answers used in the past are the participants in the process, and the mereologic components of the process; these can all be inherited, specialized or added. In our case, it will be the sub-expressions of the REprocDL concept terms that will be viewed as such inheritable features/components.



seq(
seq(
and(
seq(
opt(Reserve)
Pay)
AttendEvent))
PostDiscussion)

(a) BPMN Model *(b) Process Tree*

Fig. 3. GoingOutForEntertainment

The semantics of REprocDL, as given in Section 2, provides a subsumption relationship between $(SitDown \cdot Drink)$ and $SitDown \cdot (Eat \,\#\, Drink)$, because every instance scenario (sequence of primitive action instances) of the former is an instance of the latter. This parallels the notion of "extensional IsA constraint" (A) above, and has the immediate side-effect of providing inheritance as in (B). It also corresponds to the notion of process specialization suggested by Wyner and Lee [28]. We therefore start from the assumption that the specialization of class X leads to a class Y, whose extension, as a set of traces, should be contained in that of class X.

Next, we address the issues of how items (C) and (D) above should be handled. Concerning specialization of properties (C), since we want to maintain the "extensional IsA constraint (A)", some natural ways to do this include:

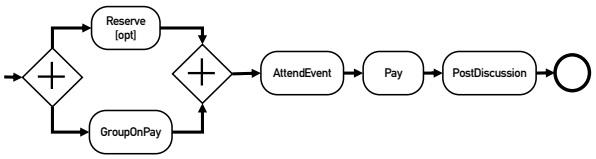
- replace named primitive actions/sub-processes (e.g., Pay) by specializations (e.g., $PayByCreditCard$ or $PayByApplePay$). This requires the introduction of a so-called Terminologic Box for named concepts, where IsA hierarchies of them can be kept. (See [9] for a simple way to continue reasoning with REprocDL in the presence of such taxonomies.)
- replace "control constructs" by ones that allow fewer sequences. For example:
  - eliminate some of the choices in a **xor** or an **and** (and possibly use **xor**(p)≡p, **and**(p)=p to simplify);
  - replace **loop**(p) by the **xor** of zero or more iterations of p;
  - replace **and** by some ordering of its arguments.

The above are easy cases to check syntactically, but unfortunately they do not form a complete set of specialization operations.

So let us return to the idea of allowing process subexpression P to be replaced by Q whenever the extension of Q is contained in that of P (a problem that is algorithmically decidable for REprocDL as $Q \sqsubseteq P$), but observe that we can do this replacement *at any place*, including the root of the expression.

To illustrate the syntax for specifying the substitution, let us consider in Figure 3, part of the process GoingOutForEntertainment[3]. If, during specialization for GoingToRockConcert, we want to force Reserve and Pay to occur

[3] For brevity, we omit henceforth the beginning, including Initiate, EstablishWho,...



seq(
  and(
    opt(Reserve)
    GroupOnPay)
  seq(
    AttendEvent
    Pay
    PostDiscussion))

(a) BPMN Model *(b) Process Tree*

Fig. 4. GroupOn_EatingOutAtRestaurant

before AttendEvent, we need to replace **and**(?X,?Y) by **seq**(?X,?Y)[4]. To identify the particular node on which we want to match (there are three **seq** nodes in the tree in Fig. 3(b)), we use a notation based on XPath, so that, for example, AttendEvent node is identified by `/.[1]/.[1]/.[2]`. Therefore the above proposed specialization is written as

$$/.[1]/.[1]: \quad \mathbf{and}(?X, ?Y) \rightsquigarrow \mathbf{seq}(?X, ?Y)$$

In the case of specializing GoingOutForEntertainment to EatingOutAtRestaurant (see Fig. 1) we could use

$$/.[1]/.[1]: \quad \mathbf{and}(\mathbf{seq}(?X1, ?X2), ?Y) \rightsquigarrow \mathbf{seq}(?X1, ?Y, ?X2)$$

to indicate that payment at the restaurant occurs after attending, while the reservation (may) occur before.

Note that both of the above specializations result in properly subsumed concepts, since all the traces in the subclass appear in the extension of the superclass. And this can be detected by the REPROCDL subsumption reasoner.

Concerning *adding new properties (D)*, we face a conundrum: in many cases we would like to add new kinds of actions to be carried out as part of the specialized process (like we added *salary* to *Employee*). For example, in creating a specialization GroupOn_EatingOutAtRestaurant, we want to add a step GroupOnPay, for paying for a GroupOn meal (which provides a discount). Let's say GroupOnPay is a new atomic action (in our case signaled by possible digital traces for web search on `groupon.com`, and a credit card payment to GroupOn). So suppose we carry out the specialization

$$/.[1]/.[1]/.[1] : \mathbf{opt}(Reserve) \rightsquigarrow \mathbf{and}(\mathbf{opt}(Reserve), GroupOnPay)$$

resulting in the process tree shown in Fig. 4, and corresponding BPMN model. However, this would cause problems because any sequence containing such a new action instance would immediately not be in the extension of the superclass. Van der Aalst and Basten [6, 1] have thoroughly investigated this question, and suggested four alternative approaches, based on the following two ideas:

[4] We use here pattern matching on terms in the style of modern functional languages such as Standard ML, Haskell, etc.

Projection inheritance: ignore new action B, by making its instances be invisible. In formal language terms, appropriate for REPROCDL, this corresponds to replacing the new atomic concept B by the concept **Null**, *for purposes of computing "proper specialization"*. This substitution can be done very efficiently, so we can use the REPROCDL subsumption algorithm to check for projection inheritance.

In the above example, REPROCDL subsumption still works, since replacing GroupOnPay by **Null** coincides with the **Null** in **opt**(Reserve), giving only a pre-existing trace. But if reservations were not optional, then we could now have a trace with no reservation (when $GroupOnPay \rightsquigarrow$ **Null**), thus obtaining an invalid specialization under projection inheritance.

Protocol inheritance: eliminate from consideration all action sequences *that contain* the new action B. In formal language terms, this corresponds to reasoning with a more complex specialized process R': the intersection of the original REPROCDL R with the "strings" *not* containing B. The latter can be described by the extended regular expression

$$B_{not} \;=\; \neg\,((Action \sqcup B)^* \cdot B \cdot (Action \sqcup B)^*)$$

Therefore, to decide if S is a proper superclass of R according to protocol inheritance, it is sufficient to decide $(R \cap B_{not}) \sqsubseteq S$. Although regular expression reasoning with constructor $\neg$ and $\cap$ is highly complex in general, this particularly simple case can be handled without increase in complexity.

Reconsidering the GroupOn_EatingOutAtRestaurant example, since all traces contain GroupOnPay, they are all eliminated, leaving the extension to be the empty set, whether or not reservation was optional.

In fact, in our case Projection Inheritance implies Protocol Inheritance, since all the sequences where **Null** was inserted would be removed from the extension. (This is due to our use of the trace equivalence instead of branching bisimulation equivalence, as in [6]). Moreover, van der Aalst and Basten suggest two other possible semantics for specialization, involving conjunction and disjunction of the above two. Because of the trace semantics, these collapse in our case.

By adopting one of the semantics for specialization suggested by van der Aalst and Basten, we then achieve a principled work-around the constraints of strict subsumption in REPROCDL, which allows the introduction of new activities in the specialized process.

4 On the Succinctness of some BPMN Models

Based on trace semantics for BPMN, we can use results from the literature on formal languages, to provide *formal proofs* for certain intuitions about constructs in BPMN notation, and how they affect brevity. The proofs of such results almost always involve exhibiting a sequence of languages/process concepts $L_1, L_2, ..., L_n$ such that their description using some feature grows slowly (e.g., linearly) as n grows, but *any* corresponding description without that feature grows much faster (e.g., exponentially). The following are a few such results:

(a) Allowing sub-process definitions can yield exponentially shorter descriptions of BPMs. In our opinion, this is an important result to remember because definitions of BPMN frequently omit this notion, possibly considering it trivial. In this one case, we provide a proof outline:

Proof. Letting $\mathcal{A}$={ A }, $\mathcal{P}$={ $E_1, E_2, ...$ }, consider the subprocess definitions:

$$E_1 \doteq A \cdot A \qquad E_2 \doteq E_1 \cdot E_1 \quad ... \quad E_n \doteq E_{n-1} \cdot E_{n-1}$$

We can see that the denotation of E_n is equal to $A^{2^{n+1}}$, and the above description has size $O(n)$. But there is no finite automaton of size less than $O(2^n)$ for this, because *any* finite automaton needs at least k states to distinguish the string A^k from all other strings. There is then no ordinary RE that is smaller in size than the FA describing its language, since there is a well known standard procedure for constructing FAs from REs, which have the same size. It only remains to show that in this case **and** cannot help, which is true because $\mathbf{and}(A^i, A^j) = A^{i+j}$ for any i, j, so nested **and**'s can be inductively eliminated. □

(b) Eliminating all **and** *split/join pairs can result in double-exponentially larger structured BPM.* The basic idea is that, with trace semantics, $\mathbf{and}(A, B)$ on atomic action pairs can be replaced by $\mathbf{xor}(\mathbf{seq}(A, B), \mathbf{seq}(B, A))$, and by iteratively substituting nested **and**s, one can eventually eliminate them. The hard part is finding a family L_n for which one can prove that the result has size at least 2^{2^n} no matter what RE/BPM one uses.

(c) "Structured" BPM comes at a price: there are unstructured BPM such that every structured version is exponentially larger.

5 Related Work

There are two threads of related work. One concerns DLs for representing processes. The second concerns specialization and inheritance of process models.

5.1 Description Logics for Processes

As discussed in Section 2, our aim was to develop a DL that can describe the behavioral semantics of (business) processes, and be able to reason with them. Within this class, we focused on a language that had the characteristic DL feature of having *variable-free concept constructors*, through which concepts and axioms can be presented and reasoned with.

One relevant thread of work here is tied to Propositional Dynamic Logic, which has program constructors { ; , | , * } corresponding to our concept constructors { $\cdot$, $\sqcup$, * }. Among others, [13] developed expressive DLs which use these as *role constructors*, rather than concept constructors. PDL is more expressive (and more complex to reason with) because in addition to complex programs, these are applied to complex formulas describing states. If the state description is just the formula *true*, then PDL expressions and REprocDL, minus # , are in some sense equivalent. However, the above DLs do not involve

concurrency. Maier & Stockmayer [19] (among others) have studied extending PDL with interleaving,but it is unclear what concurrency means with DL roles.

Another related domain are temporal logics with modal operators, such as linear temporal logic (LTL). The most relevant work here is the recent formalization of OCBC [3], which has DL concepts such $\bigcirc Eat$, denoting the instances of Eat at the next time point. On the one hand, OCBC can express some parallelism, as in $\bigcirc Eat \sqcap \bigcirc Breathe$, which $Eat \# Breathe$ must linearize. On the other hand, OCBC is based on LTL, which is incapable of representing the REPROCDL concept $(\mathbf{Action} \cdot Breathe)^*$ – breathing every second time point. So the formalisms are not fully comparable.

Finally, there is a large collection of works (e.g., [24, 10]) that encode (aspects) of BPMs, and then use Description Logic reasoning to answer particular questions about them. The main difference from these is that they do not have concepts directly denoting process executions, about which one can ask questions like subsumption, and many do not handle concurrency.

5.2 Process Specialization

Our own work in this area in the early 1980's [5, 8] added specialization and inheritance to Zisman's pioneering work on workflows [30], by allowing Petri Net edges to be specialized or added. This work was not however formalized.

Van der Aalst and colleagues have carried out influential and extensive related work we have already referenced, including on process trees, and the notions of protocol and projection inheritance. The differences are that they use branching bisimulation as their equivalence semantics, and that for inheritance, they require that on the original "vocabulary" of actions, the specialized process must behave identically to the original one, while we allow sub-process/atomic action specialization.

Wyner and Lee [28] provide a theory of process specialization for processes described by state machines (FAs). Like ours, their approach is based on an extensional semantics of the traces accepted by the FA. Their paper proposes collections of FA editing operators leading to all possible specializations (e.g., deleting nodes, adding edges). This approach is not however fully based on the set of all traces accepted by the FA, because it does not consider alternative FAs that accept the same set of traces. In contrast, by considering the language of REs (possibly built from the automata), and then using algorithms that are provably complete for detecting language containment, REPROCDL solves the more general problem.

While the above approaches, like this paper, restrict what can be done during specialization, at the other extreme is work such as [14], which uses ideas of "default inheritance" from Artificial Intelligence to permit arbitrary modifications of web-service method specifications.

An intermediate stance is taken by the subfield of process adaptation/ variability, which uses ideas such as those in [29], to "freeze" and otherwise constrain parts of the process description that can/cannot be modified during specialization. (See Prieto et al [23] for related literature.)

6 Conclusions and Future Work

The paper is motivated by our desire to reconstruct memories of autobiographical events from digital traces that can be found on smart-phones, for example. We illustrated the use of Structured BPMN to describe every-day "scripts", which support the reconstruction of the memories. We detailed REprocDL, one of the family of process description logics based on regular expressions introduced in [7, 9], and showed how it can be used to capture the dynamic semantics of Structured BPMN diagrams. This complements earlier work on ontology languages for static, syntactic aspects of BPMN [26]. The main contribution of the paper is the development of a new, more effective notion of specialization and inheritance for BPMs and workflows, which is needed to gather and organize the large number of scripts required in practice. It is based on the formal notion of subsumption in REprocDL, but allows for new action concepts to appear in specialized process. We use ideas from [1] for this extension, while still relying on the REprocDL subsumption algorithm to check the validity of specialization, albeit on a modified subsumee concept. The paper also discussed some descriptional complexity results for Structured BPMN, made possible by the formalization using regular expressions.

An open problem with specialization is that one cannot eliminate actions that appear in all traces of the superclass, when this class was over-generalized. Such situations are not rare, and we are working on an approach that adjusts definitions in such cases of "default inheritance", while maintaining clear, First Order Logic semantics. We are also investigating how to extend REprocDL with additional constructors that would cover all Petri Net languages. Finally, we plan to explore more refined notions of concurrency.

References

1. van der Aalst, W., Basten, T.: Life-cycle inheritance. In: International Conference on Application and Theory of Petri Nets. pp. 62–81. Springer (1997)
2. van der Aalst, W.M.P., Buijs, J.C.A.M., van Dongen, B.F.: Towards improving the representational bias of process mining. In: SIMPDA. pp. 39–54. Springer (2011)
3. Artale, A., Kovtunova, A., Montali, M., van der Aalst, W.M.: Modeling and reasoning over declarative data-aware processes with object-centric behavioral constraints. In: Proc. BPM'19 (2019), (to appear)
4. Baader, F., Hanschke, P.: A scheme for integrating concrete domains into concept languages. In: Proc. IJCAI'91. pp. 452–457 (1991)
5. Barron, J.: Dialogue and process design for interactive information systems using taxis. ACM SIGOA Newsletter (Proc. SIGOA'82) **3**, 12–20 (1982)
6. Basten, T., van der Aalst, W.: Inheritance of behavior. The Journal of Logic and Algebraic Programming **47**(2), 47–145 (2001)
7. Borgida, A.: Initial steps towards a family of regular-like plan description logics. In: Description Logic, Theory Combination, and All That. LNCS, vol. 11560, pp. 90–109. Springer (2019)
8. Borgida, A., Mylopoulos, J., Wong, H.K.T.: Generalization/specialization as a basis for software specification. In: On Conceptual Modelling (Intervale Workshop 1982), pp. 87–117. Springer (1984)

9. Borgida, A., Toman, D., Weddell, G.: On special description logics for processes and plans. In: Proc. Workshop on Description Logics (DL 2019) (2019)
10. Calvanese, D., Montali, M., Patrizi, F., De Giacomo, G.: Description logic based dynamic systems: Modeling, verification, and synthesis. In: IJCAI'15. pp. 4247–4253
11. Corradini, F., Ferrari, A., Fornari, F., Gnesi, S., Polini, A., Re, B., Spagnolo, G.O.: A guidelines framework for understandable BPMN models. Data & Knowl. Engineering **113**, 129–154 (2018)
12. De Giacomo, G., Dumas, M., Maggi, F.M., Montali, M.: Declarative process modeling in BPMN. In: Proc. CAiSE'15. pp. 84–100 (2015)
13. De Giacomo, G., Lenzerini, M.: TBox and ABox reasoning in expressive description logics. In: Proc. AAAI'96. pp. 37–48. AAAI Press (1996)
14. Ferndriger, S., Bernstein, A., Dong, J.S., Feng, Y., Li, Y.F., Hunter, J.: Enhancing semantic web services with inheritance. In: Proc. ISWC'08. pp. 162–177 (2008)
15. Frank, U.: Specialisation in business process modelling: Motivation, approaches and limitations. Tech. rep., Institut für Informatik und Wirtschaftsinformatik (ICB), Universität Duisburg, Essen (2012)
16. Kalokyri, V., Borgida, A., Marian, A.: YourDigitalSelf: A personal digital trace integration tool. In: Proc.CIKM. pp. 1963–1966. ACM (2018)
17. Kiepuszewski, B., Ter Hofstede, A.H.M., Bussler, C.J.: On structured workflow modelling. In: Proc. CAiSE. pp. 431–445. Springer (2000)
18. Leemans, S.J.J., Fahland, D., van der Aalst, W.M.P.: Discovering block-structured process models from incomplete event logs. In: Proc. Petri Nets 2014. pp. 91–110
19. Mayer, A.J., Stockmeyer, L.J.: The complexity of PDL with interleaving. Theoretical Computer Science **161**(1-2), 109–122 (1996)
20. Mylopoulos, J., Bernstein, P.A., Wong, H.K.T.: A language facility for designing database-intensive applications. ACM Trans. Database Syst. **5**(2), 185–207 (1980)
21. OWL-S Coalition: OWL-S 1.1 Release (2004)
22. Prescher, J., Di Ciccio, C., Mendling, J.: From declarative processes to imperative models. Proc. SIMPDA'14 **14**, 162–173
23. Prieto, Á.E., Lozano-Tello, A., Rodríguez-Echeverría, R., Preciado, J.C.: A hierarchical adaptation method for administrative workflows. IEEE Access **7** (2019)
24. Ren, Y., Gröner, G., Lemcke, J., Rahmani, T., Friesen, A., Zhao, Y., Pan, J.Z., Staab, S.: Validating process refinement with ontologies. In: Proc. Workshop on Description Logics (DL 2009)
25. Riboni, D., Bettini, C.: Owl 2 modeling and reasoning with complex human activities. Pervasive and Mobile Computing **7**(3), 379–395 (2011)
26. Rospocher, M., Ghidini, C., Serafini, L.: An ontology for the business process modelling notation. In: Proc. FOIS'14. pp. 133–146 (2014)
27. Schmiedel, A.: Temporal terminological logic. In: Proc. AAAI'90. pp. 640–645
28. Wyner, G.M., Lee, J.: Process specialization: Defining specialization for state diagrams. Computational & Mathematical Organization Theory **8**(2), 133–155 (2002)
29. Wyner, G.M., Lee, J.: Applying specialization to petri nets: implications for workflow design. In: Proc. BPM'05. pp. 432–443. Springer (2005)
30. Zisman, M.: Representation, Specification And Automation Of Office Procedures. Ph.D. thesis, University of Pennsylvania (1977)