Information Systems 140 (2026) 102718

Contents lists available at ScienceDirect

Information Systems

journal homepage: www.elsevier.com/locate/is

Formal semantics for knowledge representation and automated reasoning in BPMN process models☆

Angelo Casciani [a,*], Simone Agostinelli [b], Yves Lespérance [c], Andrea Marrella [a], Sebastian Sardiña [d]

[a] Department of Computer, Control and Management Engineering, Sapienza University of Rome, via Ariosto 25, Rome, 00185, Italy
[b] Department of Engineering and Sciences, Mercatorum University of Rome, Piazza Mattei 10, Rome, 00186, Italy
[c] Department of Electrical Engineering and Computer Science, York University, 4700 Keele Street, Toronto, M3J 1P3, Ontario, Canada
[d] School of Computing Technologies, Royal Melbourne Institute of Technology, 414-418 Swanston Street, Melbourne, VIC 3000, Victoria, Australia

ARTICLE INFO

Dataset link: https://github.com/angelo-casciani/sitcalc4bpmn

Keywords:
AI-augmented business process management systems
ConGolog
Formal semantics of BPMN
Knowledge representation and automated reasoning
Process modeling
Situation Calculus

ABSTRACT

The Business Process Modeling Notation (BPMN) is the de facto standard for business process modeling. While widely adopted for its intuitive graphical notation, its execution semantics described in natural language lacks a commonly agreed formal foundation, leading to variability in execution across different BPM systems (BPMSs) and increasing the risk of creating models with semantic errors costly to correct at runtime. Although many formalisms have been used to model portions of BPMN, their reasoning capabilities are mostly restricted to control-flow, making them unsuitable for semantic analysis where data and global exception handling play a central role in execution. To address this, we propose a formalization from BPMN to ConGolog, a logical concurrent processes language based on the Situation Calculus, for representing and reasoning about dynamic domains. A major innovation is using ConGolog to rigorously capture the semantics of BPMN global exceptions. Our framework supports advanced reasoning, allowing for semantic analysis of BPMN models before execution to predict runtime errors within a safe simulation setting, while laying the foundation for reasoning layers in next-generation AI-augmented BPMSs. We validate the approach through a prototype and comprehensive evaluation, demonstrating the computational feasibility of the translation and the semantic correctness of reasoning tasks.

1. Introduction

Business Process Management (BPM) is an active research field dedicated to overseeing a series of coordinated activities, referred to as *business processes*, which contribute to achieving organizational goals. *Process models* define the structure and boundaries within which processes operate, specifying the workflow, tasks, and rules for their automated execution [1].

The de facto standard for process modeling, called *Business Process Modeling Notation* (BPMN), is widely adopted in industry and academia for its intuitive graphical notation and is supported by the majority of commercial *BPM Systems* (BPMSs) [2]. Nonetheless, it provides an execution semantics that is described in natural language and lacks a commonly agreed formal foundation [3]. As a result, BPMN process execution can vary significantly across different BPMSs, leaving aspects such as data flow behavior, error handling, and multi-party collaboration open to interpretation [4]. This variability increases the risk of creating process models with semantic errors, which are costly and difficult to correct at runtime.

Furthermore, as outlined in a recent research manifesto [5], the emerging shift from traditional BPMSs to AI-augmented BPMSs (ABPMSs) is driving a transformation of the conventional BPM lifecycle phases (e.g., modeling, execution, monitoring, etc.) into a series of advanced AI-enhanced steps. Central to this transformation is a *reasoning layer* that interprets the semantics of a BP model, assesses its current execution state, and determines the next runtime action, guided by operational assumptions, goals, and environmental constraints. Unlike the well-established BPM philosophy, where a BPMN model is manually configured by a system engineer for execution in a BPMS through explicit definitions of data models and custom scripts for business rules, an ABPMS demands a *higher degree of autonomy*. This autonomy can be achieved with a *formal specification framework* that supports automated reasoning over data values and dynamic scenarios, thus allowing for: (i) comprehensive *semantic analysis* of process models

☆ This article is part of a Special issue entitled: 'Autonomous Process Execution Systems' published in Information Systems.

* Corresponding author.
E-mail address: casciani@diag.uniroma1.it (A. Casciani).

https://doi.org/10.1016/j.is.2026.102718
Received 1 December 2025; Received in revised form 30 January 2026; Accepted 9 March 2026
Available online 17 March 2026

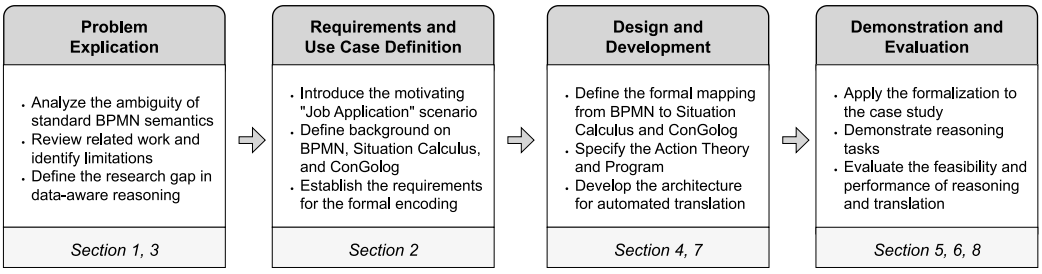


Fig. 1. Research approach based on design science principles by [9].

prior to execution, and (ii) *prediction of runtime errors* in a controlled simulation environment.

In light of these objectives, we envision two research questions (RQs):

RQ1: *"How can Knowledge Representation and Reasoning languages formalize the semantics of BPMN?"*

RQ2: *"How to leverage this formal encoding to enable data-aware reasoning tasks that go beyond the structural analysis typically available in traditional process modeling tools?"*

To address these questions, this paper presents *formal process semantics* for a significant subset of BPMN using ConGolog [6], a formal language rooted in the *Knowledge Representation and Reasoning* (KR&R) domain, designed for concurrent processes based on the Situation Calculus [7]. While various well-established formalisms (e.g., based on Petri nets and process algebras) have been employed to rigorously represent fragments of BPMN, their automated reasoning capabilities mainly rely on synthesis techniques that focus exclusively on the control-flow perspective. As a result, their abstraction level in handling data and dynamic scenarios is relatively limited compared to the advanced reasoning capabilities offered by KR&R methods. An important innovation of our approach lies in the treatment of BPMN global exceptions through the formalization of event-based subprocesses. This is made possible by the capabilities of ConGolog, which supports the modeling of concurrent activities and interrupts, making it well-suited to capturing the multi-threaded nature of business processes. In particular, we introduce a specific extension to the language, i.e., the *guarded execution* construct, to capture the non-trivial semantics of BPMN global exceptions. This allows for modeling asynchronous interrupts in a modular way, avoiding the "state explosion" often encountered when flattening such constructs into standard Petri nets.

Based on these considerations, we argue that our framework serves as a foundational step toward the direct application of reasoning techniques to BPMN processes, aiming to establish a *formal basis for developing a reasoning layer* in ABPMSs, thereby facilitating the creation, acquisition, adaptation, evolution, and sharing of data-driven knowledge. We demonstrate that our framework supports *basic*, like projection and legality, and *advanced reasoning tasks*, such as conformance checking [8] and property verification, extending beyond the capabilities of standard BPMN tools, which are generally limited to syntactic validation and structural soundness checks.

We validate these claims by introducing a *software architecture* that automates the translation of BPMN process models into our formal encoding, thereby making the proposed reasoning capabilities more accessible. We further present an extensive experimental evaluation of the implementation, assessing the correctness and performance of both the automated translation and the reasoning tasks, and demonstrating their practical feasibility.

The rest of the paper is organized according to the activities outlined in [9] for delivering a design science artifact, as reported in Fig. 1:

1. **Problem Explication.** This phase, guided by the RQs reported in the current section, outlines the problem being tackled and motivates its relevance. It is mainly addressed in Section 3, which reviews related work to underscore the limitations of current approaches and the research gap our solution aims to fill.
2. **Requirements and Use Case Definition.** The second phase focuses on defining the scenario used to ground the research. Section 2 addresses this by introducing the motivating case study, alongside the preliminary notions on BPMN, Situation Calculus, and ConGolog necessary to understand the subsequent formalization. We also define here the requirements for the formal encoding, in particular extending ConGolog to support the handling of *global exceptions* in the process model.
3. **Design and Development.** This phase concerns the creation of the design science artifact. Section 4 details the theoretical contribution, presenting the formalization framework that maps BPMN to Situation Calculus and ConGolog. Complementing this, Section 7 introduces the practical part of the work, i.e., a software architecture developed to automate the encoding and reasoning over BPMN process models.
4. **Demonstration and Evaluation.** The final phase focuses on proving the feasibility and effectiveness of the solution. We demonstrate the artifact in Section 5 by applying the formalization to the defined case study, and in Section 6 by showcasing the specific reasoning tasks it enables. Subsequently, Section 8 presents an experimental evaluation assessing the performance and correctness of both the automated translation and the reasoning tasks.

Finally, Section 9 discusses how the overall outcome of the study addresses the previously introduced RQs and the limitations of the framework. Section 10 concludes the paper by summarizing the contributions and introducing future directions.

2. Background

2.1. Business Process Modeling Notation (BPMN)

BPMN is a standard graphical language developed by the Object Management Group (OMG) to design process models[1] with an emphasis on the control flow. BPMN defines a flowchart including a range of diverse constructs that can be divided into: (*i*) *flow objects*, (*ii*) *data*, (*iii*) *connecting objects*, and (*iv*) *pools*.

Our framework accounts for a large subset of BPMN elements encompassing data and control flow components, including those in the model presented in Fig. 2 about a job application process.

[1] We refer to BPMNv2.0: http://www.omg.org/spec/BPMN/2.0/.

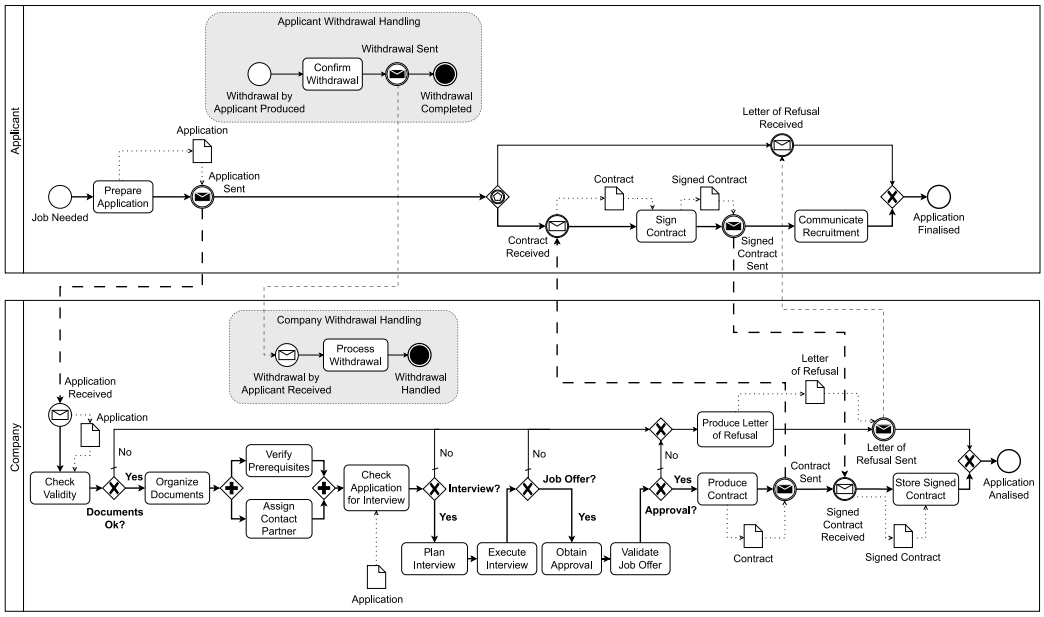


Fig. 2. A BPMN model representing a business process to handle a job application. The main control flow (i.e., the "happy path") and positive decision outcomes are highlighted with thicker arrows and bold text, while the global exception handling subprocesses feature a gray background.

Running example. We use this job application process as a running example. The process models the interaction between two participants: an *applicant* and a *company*. The flow begins when the applicant prepares and submits a job application. Upon receipt, the company *checks the validity* of the documents. This serves as the first decision point: if the documents are invalid, the company produces and sends a refusal letter, terminating the process. If the documents are valid, the company proceeds to execute *parallel tasks*: verifying the candidate's prerequisites and assigning a contact partner. Subsequently, the application is evaluated to determine if the candidate qualifies for an interview. If eligible, the interview is planned and executed. A positive evaluation leads to the *job offer* phase, where the company validates the offer details and seeks internal approval. If approved, a contract is produced and sent to the applicant. The process concludes successfully when the applicant returns the signed contract, which is then stored by the company. Notably, this BPMN model includes *global exception handling*: the applicant can choose to *withdraw* their application at any point during the process. This event acts as a global interrupt that halts the company's current activities, triggers a specific withdrawal handling procedure, and terminates the process instance. Such an example serves as our use case throughout the paper. In Sections 5 and 6, we will revisit it to demonstrate how our formal framework enables rigorous encoding and reasoning over such a process model.

In the following, we detail the BPMN constructs used to model such a process.

Flow objects. *Flow objects* define the behavior of a process and can be classified into *events*, *activities*, and *gateways*. *Events* model the occurrence of states in the real world that are relevant for processes and, more generally, anything that can happen instantaneously (e.g., an application has been received). *Events* in BPMN can be partitioned into three types, based on their position in the process. *Start events*, which are depicted as circles with a thin border (e.g., the initial event in the applicant process indicating the need for a job) and have no incoming sequence flow edges, are used to trigger processes and, from the simulation perspective, create tokens.[2] *Intermediate events*, represented as circles with a double border and having both an incoming and an outgoing sequence flow edge, can delay processes or be triggered during process executions (e.g., the throw message event for sending and the catch message event for receiving of an application). *End events*, which are modeled as circles with a thick border and have no outgoing sequence flow edges, indicate the completion of processes and the destruction of the tokens (e.g., the finalization of the application by the applicant). A specific variant, the *terminate end event*, depicted as an end event with a black inner circle, triggers the immediate termination of a process (e.g., completing the application withdrawal) and causes the destruction of all tokens within the process instance.

Activities (or *tasks*), depicted as labeled rounded boxes, represent units of work performed during processes that, unlike the events, have a certain duration (e.g., prepare an application). Activities capturing units of work that are not further refined are named atomic activities or *tasks*, while composite activities that can be broken down into smaller units of work achieving a goal are called *sub-processes*.

An *event sub-process*, delimited by a dotted rectangle with rounded corners, is started by the event attached to an activity's boundary and encloses the procedure that would be triggered by the boundary event. Those sub-processes serve to manage both the reception of external messages as well as handling of exceptions. In the latter scenario, the process is not aborted, and the error is caught by the event sub-process that provides the recovery activity to obtain a consistent state of execution. In this study, we focus on global exceptions that can be raised at any point in the overarching process, such as handling the application's withdrawal coming from the applicant.

Gateways are used to represent the split and join behavior of the control flow when there is a need to model specific conditions like mutual exclusion or concurrency. A gateway determines the forking, merging, or joining of paths. *Split gateways* have one incoming edge and at least two outgoing edges, representing the flow that is branched. By contrast, *join gateways* have at least two incoming edges and one outgoing edge, modeling the merging of the control flow. The main gateway types are *exclusive*, *parallel*, and *event-based*. *Exclusive gateways*, also known as *XOR gateways*, model the relation between two or more alternative activities that are mutually exclusive, i.e., only one of them can be executed according to a data-driven condition (e.g., the gateway that checks if all documents for the application are in order). *Parallel gateways*, or *AND gateways*, model the relation between two

[2] Tokens are a conceptual tool to describe the execution flow of a process instance as it progresses through the flow objects of the model.

or more activities that can be executed concurrently: they have no order dependencies but must all be executed, as seen in the gateway preceding the tasks for verifying prerequisites and assigning a contact partner. Specifically, the AND-split models the parallel execution of two or more branches, and the AND-join awaits all the tokens that have been created and merges them into one. *Event-driven gateways* model the *race* of several events against each other, i.e., the first event to occur determines the continuation of the process. The event-driven gateway acts like a split gateway, with as many outgoing edges as the number of intermediate catching events participating in the race. When a token arrives at this gateway, the instance execution stops until one of the events on the outgoing branches occurs. This construct is used in Fig. 2 to represent the applicant's wait for either the contract or the letter of refusal issued by the company in response to their application.

Data. BPMN allows the explicit modeling of data (e.g., applications, contracts, etc.) using *data objects* and *data stores*. They primarily serve to represent the informational aspects of a process model, but they influence execution semantics, particularly when used in conditions, data associations, or input/output specifications. *Data objects* provide information regarding activity requirements and specify data inputs and outputs of activities. *Data stores* are places containing data objects that need to persist beyond the duration of a process instance, e.g., a database.

Connecting objects. *Connecting objects* connect flow objects or data objects/stores. The *Sequence flow* is used to specify the ordering of flow objects, while *message flow* describes the flow of messages between business partners represented by pools. *Association* is a specific type of connecting object that is used to link data objects/stores to activities.

Pools. From a resource perspective, *pools* are used to model resource classes, i.e., independent organizational entities that do not share any common system, but communicate with each other through messages (e.g., the applicant and the company).

2.2. Situation calculus

The *Situation Calculus* is a logical language designed for representing and reasoning about dynamic domains [7]. The dynamic world is seen as progressing through a series of situations as a result of various *actions* being performed.

Definition 2.1 (*Situations and Actions*)**.** A *situation* s is a first-order term denoting the sequence of actions performed so far. The special constant S_0 stands for the initial situation, where no action has yet occurred. The special binary function symbol $do(a, s)$ denotes the situation resulting from the performance of action a in situation s.

Definition 2.2 (*Sub-situation*)**.** A situation s is a *sub-situation* of s', denoted $s \sqsubseteq s'$, if and only if s is a prefix of s'. Formally, the relation is axiomatized as follows:

$$s \sqsubseteq s' \equiv [s = s' \vee (\exists a, s'').s' = do(a, s'') \wedge s \sqsubseteq s'']$$

Definition 2.3 (*Fluents*)**.** Relations and functions whose value may change from one situation to the next are modeled through *fluents*.

- *Relational fluents* are relations whose truth values vary from situation to situation.
- *Functional fluents* are functions whose values vary from situation to situation.

Both are denoted by function symbols taking a situation term as their last argument. Without loss of generality, we used for the proposed formalization only *relational fluents*.

Two special predicates govern the executability and nature of actions. The predicate $Poss(a, s)$ is used to state that action a is executable in situation s, whereas the situation-independent predicate $Exog(a)$ is used to denote that a is an exogenous event (i.e., an event originating from the external environment). We denote by $\phi(\vec{x})$ a formula whose free variables are among variables $\vec{x}$, assumed to be universally quantified. A *fluent formula* is one whose only situation term mentioned is the situation variable s.

Within this language, one can formulate action theories describing how the world changes as a result of the available actions.

Definition 2.4 (*Basic Action Theory*)**.** A *Basic Action Theory* (BAT) $\mathcal{D}$ is defined, according to [7], as:

$$\mathcal{D} = \Sigma \cup \mathcal{D}_{S_0} \cup \mathcal{D}_{poss} \cup \mathcal{D}_{ss} \cup \mathcal{D}_{una}$$

where:

- Σ are the *domain-independent foundational axioms* describing the structure of situations and auxiliary relations like $\sqsubseteq$ and $Executable(s)$.
- $\mathcal{D}_{S_0}$ are the *initial state axioms*, a collection of first-order sentences whose only situation term mentioned is the situation constant S_0. These define what is true initially in S_0, as well as auxiliary situation-independent information (such as axioms for the predicate $Exog(a)$).
- $\mathcal{D}_{poss}$ are the *action precondition axioms*, one per action, specifying when the action is executable. They are of the form:

  $$Poss(a(\vec{x}), s) \equiv \Pi_a(\vec{x}, s)$$

  where $\Pi_a(\vec{x}, s)$ is a fluent formula defining the conditions under which action a can be legally executed in situation s.
- $\mathcal{D}_{ss}$ are the *successor state axioms*, one per fluent, capturing the effects and non-effects (i.e., frame) of actions. The successor state axiom for a relational fluent $F(\vec{x}, s)$ is of the form:

  $$F(\vec{x}, do(a, s)) \equiv \Psi_F(\vec{x}, a, s)$$

  where $\Psi_F(\vec{x}, a, s)$ is a fluent formula characterizing the dynamics of fluent $F(\vec{x}, s)$. Importantly, $\Psi_F(\vec{x}, a, s)$ can accommodate Reiter's [7] solution to the frame problem.[3] Free variables are assumed to be universally quantified.
- $\mathcal{D}_{una}$ are the *unique name axioms* assumed for actions.

2.3. ConGolog

Building upon situation calculus action theories, logic-based programming languages have been proposed to allow for the specification of complex actions. Golog [10] was the first such language, providing a high-level programming formalism for agents acting in dynamic domains described by the situation calculus. ConGolog [6] extends Golog by introducing explicit constructs for *concurrent processes*, *prioritized execution*, and *interrupts*, thus enabling the specification of programs where multiple activities can progress in an interleaved or parallel fashion.

Formally, ConGolog relies on a transition semantics defined in terms of single-step transitions and final configurations, extending the original Golog semantics to handle multiple control threads. This makes ConGolog suitable for describing *reactive* and *multi-threaded* behaviors within a logical framework.

We focus here on the core fragment of the language.

[3] In AI, the *frame problem* is the challenge of capturing the effects of actions succinctly, without explicitly representing a large number of intuitively obvious non-effects [7].

Definition 2.5 (*ConGolog Syntax*)**.** The set of legal ConGolog programs δ is defined inductively by the following constructs:

α	*atomic action*
$\phi?$	*test for a condition*
$(\delta_1; \delta_2)$	*sequence*
$(\delta_1 \mid \delta_2)$	*nondeterministic choice between actions*
$\pi x.\delta(x)$	*nondeterministic choice of argument*
δ^*	*nondeterministic iteration*
if ϕ **then** δ_1 **else** δ_2 **endIf**	*conditional*
while ϕ **do** δ **endWhile**	*while loop*
proc $P(\vec{x})$ **do** $\delta(\vec{x})$ **endProc**	*procedure definition*
$\delta_1 \parallel \delta_2$	*concurrency*
$\delta_1 \rangle\rangle \delta_2$	*prioritized concurrency*
$\delta^{\parallel}$	*concurrent iteration*
$\langle \phi \rightarrow \delta \rangle$	*interrupt*

Regarding the behavior of specific constructs: the test $\phi?$ proceeds only if condition ϕ holds; $\pi x.\delta(x)$ executes $\delta(x)$ for *some* nondeterministic choice of argument x; and δ^* executes δ zero or more times. Concurrent execution is captured by $\delta_1 \parallel \delta_2$ (interleaved execution) and $\delta_1 \rangle\rangle \delta_2$ (prioritized execution), where δ_1 has higher priority and δ_2 executes only when δ_1 is blocked or completed. The construct $\delta^{\parallel}$ denotes the concurrent iteration of δ. Finally, $\langle \phi \rightarrow \delta \rangle$ represents an interrupt mechanism: if ϕ becomes true, program δ is triggered and must execute to completion.

Definition 2.6 (*Transition Semantics*)**.** The formal semantics of ConGolog is defined through two predicates [6]:

- $Trans(\delta, s, \delta', s')$: Holds if one execution step of program δ in situation s leads to situation s', with δ' remaining to be executed.
- $Final(\delta, s)$: Holds if program δ may legally terminate in situation s.

Relationship with IndiGolog. A further extension of ConGolog, IndiGolog [11], augments the language with support for *interleaved actions*, *sensing*, and *lookahead planning*. While ConGolog specifies programs whose execution semantics are fully determined by logical transition rules, IndiGolog programs are designed for *online* execution: at every step, a legal next action is selected, performed in the world, and its sensing outcome gathered. To account for planning, a special lookahead construct $\Sigma(\delta)$ (the *search operator*) is provided to encode the requirement of finding a complete execution of δ offline before committing to actions.

For our purposes, we rely on the IndiGolog implementation [11] built in Prolog,[4] which provides an executable environment for the constructs defined above. As part of this work (see Section 4.1), we extend this set of constructs with a mechanism to abort the current program.

3. Related work

The formalization of BPMN semantics has been extensively studied to mitigate the ambiguities inherent in the standard. Despite its widespread adoption in academia and industry, the natural language specification of BPMN often leads to inconsistent implementations and underspecified data behaviors, frequently reducing data specifications to unstructured textual comments. Over the years, several approaches attempted to formalize specific perspectives of BPMN semantics. To clarify the positioning of our contribution, we categorize these works into three primary research lines based on their focus: *control-flow*-based approaches, *collaboration-oriented* formalisms, and *data-centric* encodings. Furthermore, we review the broader application of KR&R techniques for process modeling.

[4] IndiGolog repository: https://github.com/ai-krml-uoft/indigolog.

Table 1 provides a detailed comparison of our framework against the main existing BPMN formalizations, highlighting trade-offs in expressive power and reasoning capabilities. While approaches like [12, 13] support, for instance, the inclusive OR-gateway (which we currently do not due to inherent limitations of the employed language), our framework combines native data-awareness and global exceptions handling.

3.1. BPMN semantics for control-flow

Early approaches prioritized mapping BPMN to formalisms such as *Petri nets* to leverage well-established analysis techniques for control-flow verification. Dijkman et al. [3] provided a foundational encoding of BPMN control-flow (i.e., the subset of the notation dealing with the ordering of activities and events) to Petri nets, enabling the verification of properties like soundness and presence of deadlocks. However, these approaches typically abstract away data and exception handling mechanisms to avoid state-space explosion, thereby limiting their analysis and reasoning abilities. Furthermore, they do not address BPMN's non-functional aspects (e.g., artifacts) or organizational features (i.e., pools).

To address some of these limitations, Kheldoun et al. [17] proposed a verification approach based on high-level Petri nets (namely, Recursive ECATNets). This method handles both data (via abstract data types) and cancellations, offering a hybrid middle ground. Nevertheless, it relies on heavy transformation rules to an intermediate formalism, whereas our approach maps BPMN directly to situation calculus and ConGolog.

Other works have employed *Abstract State Machines* (ASMs) to provide a ground model for BPMN 2.0 [16], covering data, events, and message flows. While ASMs are effective for specification and simulation, they lack the native logical inference mechanisms found in KR&R, which allow for tasks like determining if a specific data state is reachable (i.e., projection) without simulation.

3.2. BPMN semantics for collaboration

A significant body of work has focused on the formal verification of BPMN collaborations. Houhou et al. [12] introduced a first-order logic semantics implemented in TLA$^+$ to verify correctness properties of process collaborations. The approach is expressive in control-flow, supporting the complex inclusive OR-gateway and various communication models, but it abstracts data away to keep verification tractable. Similarly, Corradini et al. [13] proposed a tool that addresses the state-explosion problem inherent in collaborative models by integrating statistical model checking. It supports the OR-gateway and checks for system-independent properties such as safeness, but explicitly excludes quantitative data aspects to focus on control-flow correctness. Complementing these, [4] provides a data-agnostic operational semantics of BPMN collaborations based on a labeled transition system (LTS) and defines three correctness properties: well-structuredness, safeness, and soundness.

Shifting focus from single collaborations to *business process architectures*, Eid-Sabbagh et al. [14] analyze collections of process models to capture message and trigger flow relations. By transforming the architecture into open nets (a variant of Petri nets) and using CTL formulas for verification, they ensure structural consistency across interconnected processes. While this encoding is highly expressive at the architectural macro level, it operates at a lower level regarding internal process semantics compared to our framework, as it does not model data values or exception constructs (such as terminate end events), focusing instead on message passing rather than data-based routing.

Table 1
Comparison of our approach with relevant related works focusing on BPMN semantics formalization. A checkmark (✓) denotes supported features, a cross (✗) the unsupported ones, and a circle (●) partial support. In the expressive power column, (+) highlights strengths while (−) indicates limitations.

Reference	Formalism	Data	Glob. Ex.	Collab.	Analysis focus	Expressive power & limitations
Dijkman et al. [3]	Petri nets	✗	●	✗	Structural (Control-flow soundness)	(+) Deadlock and reachability check. (−) No data; limited exception handling.
Eid-Sabbagh et al. [14]	Open Nets/CTL	✗	✗	✓	Architecture analysis (interaction soundness)	(+) Interdependencies via triggers/messages. (−) No data values; limited control logic.
Meyer et al. [15]	Relational/SQL	✓	✗	✗	Enactment & data dependencies	(+) Complex data dependencies. (−) No collaboration.
Kossak et al. [16]	ASM	✓	✓	✓	Simulation	(+) Extensive BPMN coverage. (−) Verification requires transformation; higher complexity.
Kheldoun et al. [17]	Recursive ECATNets	✓	●	✓	Verification (LTL/Maude)	(+) Handles data & cancellation. (−) Complex mapping rules.
Houhou et al. [12]	FOL/TLA+	✗	✗	✓	Verification (safety, soundness)	(+) OR-gateways and communication. (−) Data abstracted.
Corradini et al. [4]	Op. Semantics (LTS)/S3	✗	✓	✓	Structural classification (safety, soundness)	(+) Arbitrary topologies, terminate events, and sub-processes. (−) Data abstracted. State explosion.
Corradini et al. [13]	Op. Semantics/Maude	✗	✗	✓	Verification (LTL & Statistical)	(+) OR-gateways & large state spaces. (−) No data support.
Corradini et al. [18]	Op. Semantics (LTS)	✓	✗	✓	Animation & Debugging	(+) Multi-instance data & correlations. (−) Simulation only.
Our approach	SitCalc/ConGolog	✓	✓	✓	Semantic reasoning	(+) Native data fluents and global exceptions. (−) No OR-gateway; structured models assumed.

3.3. BPMN semantics for data

Regarding the data perspective, several approaches have aimed to integrate data into the formalization. Meyer et al. [15] investigated the enactment of data dependencies, generating SQL queries from BPMN data objects. This method can handle $m : n$ data relationships but focuses on enactment rather than formal reasoning. The framework in [18] focuses on the interplay between control features, multiple instances, and data handling (including collections and stores). However, it is mainly an animator for debugging, whereas our encoding uses the IndiGolog interpreter to perform reasoning tasks. In the broader literature, data-centric process modeling techniques like [19] offer interesting alternatives (e.g., artifact-centric models). However, these often require adopting entirely new modeling paradigms and languages and are not widely adopted by practitioners compared to BPMN. In contrast, our goal is to inject data-aware reasoning directly into standard BPMN models.

3.4. Knowledge representation and reasoning for process modeling

We now review KR&R techniques that formalize semantics for process languages beyond BPMN. Early work by [20] provided semantics for WS-BPEL (Business Process Execution Language), a process calculus model of computation for concurrent systems, using an extension of the π-calculus [21]. Originally intended for web service orchestration, WS-BPEL has seen reduced overlap with BPMN over time. Consequently, BPMN has established its own interchange format for process modeling, while WS-BPEL is now considered outdated and no longer mainstream.

More recently, research has explored the intersection of declarative modeling and logic. To tackle the limitations of imperative notations in handling high-variability processes, De Giacomo et al. [22] proposed BPMN-D, a conservative extension of BPMN that include the translation of "open-world" declarative constraints, defined in *Declare* using Linear Temporal Logic on finite traces (LTL_f). Moreover, addressing the translation from imperative to declarative models, Barbaro et al. [23] introduced an algorithm to synthesize semantically equivalent *Declare* specifications from safe and sound workflow nets.

Regarding semantic analysis, the work in [24] demonstrated how Answer Set Programming (ASP) can integrate domain ontologies for reasoning about business processes. By formalizing domain knowledge in a low-complexity description logic and using causal rules, their framework supports the verification of temporal logic properties against process models enriched with semantic constraints. Similarly, [25] investigated relational semantics for Petri nets, defining a spectrum of parameterized relation sets to abstract behavioral profiles for workflow verification.

Focusing on the generation of synthetic data, Skydanienko et al. [26] used the Alloy model checker to handle constraints on both control flow and data attributes for generating event logs from Multi-Perspective Declare models.

Within the specific domain of Situation Calculus and ConGolog, several works have demonstrated feasibility in practical tasks such as process execution, monitoring, and recovery. SmartPM [27] provides a model and prototype BPMS supporting the automated adaptation of knowledge-intensive processes at runtime. However, the formal specification in SmartPM captures only basic BPMN constructs (activities,

AND/XOR gateways), neglecting events, pools, and data objects. Furthermore, Golog was also used for composing web services [28]. Although focused on formalized services, this technique could potentially be applied to automate the construction of BPMN models. Finally, works like [29–31] have investigated the verification of Golog programs. While these are currently research artifacts, they demonstrate the feasibility of verifying non-trivial properties over complex theories, a capability that our framework extends to the BPMN standard.

4. Formalizing BPMN in situation calculus and ConGolog

Table 2 presents our formalization framework for BPMN constructs and patterns using *situation calculus* and ConGolog. It is important to note that this formalization includes the *minimal* set of situation calculus and ConGolog statements to render the considered BPMN patterns, thereby avoiding redundant logical axioms and ensuring the generated encoding is compact and efficient for reasoning.

Durative task. A single BPMN (durative) task, associated with vectors of input and output data objects $\vec{x}$ and $\vec{y}$ and a process instance's identifier id, is represented in the action theory by a primitive action $\alpha_s(id, \vec{x})$ that starts the task and an exogenous action $\alpha_e(id, \vec{y})$ that concludes it. The relational fluents $Running(\alpha_s(id, \vec{x}), s)$ and $Done(\alpha_e(id, \vec{y}), s)$ represent the task's initiation and completion. Specifically, $Running(\alpha_s(id, \vec{x}), s)$ holds if the starting action is ongoing and the completing action is not yet performed. Additionally, we use the fluents $\beta(id, x, s)$ and $\gamma(id, y, s)$ to identify the types of the input and output data objects linked to the execution of the process instance with the provided id. Notably, the starting action $\alpha_s(id, \vec{x})$ can occur when $Running(\alpha_s(id, \vec{x}), s)$ is false and the input data objects satisfy $\beta(id, \vec{x}, s)$ for the given process id. The effect axiom for $\alpha_s(id, \vec{x})$ sets the corresponding $Running(\alpha_s(id, \vec{x}), s)$ fluent to be true. Instead, the completion action $\alpha_e(id, \vec{y})$ can be executed if its corresponding $Running(\alpha_s(id, \vec{x}), s)$ holds and the required output data objects satisfy $\gamma(id, \vec{y}, s)$ for the given execution id. The completion action $\alpha_e(id, \vec{y})$ sets the $Done(\alpha_e(id, \vec{y}), s)$ fluent to be true. Furthermore, $\alpha_e(id, \vec{y})$ generates the output data objects of type γ, as indicated by the associated relational fluent. Both α_s and α_e usually affect some other fluents in a task-specific manner, and these are specified by providing additional effect axioms. Initially, both the $Running(\alpha_s(id, \vec{x}), s)$ and $Done(\alpha_e(id, \vec{y}), s)$ fluents for the task in this instance are assumed to be false, and the output data objects do not yet exist. In the high-level program, the primitive action $\alpha_s(id, \vec{x})$ is executed to start the task for the given case, along with its input data objects. This is followed by a test on $Done(\alpha_e(id, \vec{y}), s)$, which waits for the exogenous action $\alpha_e(id, \vec{y})$ to finish the task. For instantaneous actions, we can omit the completion event $\alpha_e(id, \vec{y})$ and set $Done(\alpha_e(id, \vec{y}), s)$ to true as soon as $\alpha_s(id, \vec{x})$ is executed.

Sub-processes sequence. The second modeled pattern represents a sequence of two sub-processes, δ_i and δ_j, which is naturally translated into the program using the ConGolog sequence operator.

Exclusive gateway. The exclusive gateway pattern is captured with the ConGolog *if-then-else* construct. Specifically, ϕ acts as a composite condition governing the control flow, determining which branch to follow and executing the corresponding sub-processes δ_i or δ_j. The default branch of the BPMN XOR gateway aligns with the *else* clause in the ConGolog conditional construct.

Parallel gateway. The parallel gateway, denoting the concurrent execution of sub-processes δ_i and δ_j, is expressed in the high-level program using the concurrency operator $\delta_i \parallel \delta_j$.

Structured loop. The BPMN pattern for the structured loop, implemented using an exclusive gateway to check if a specific condition ϕ is satisfied after an iteration, is modeled with the sequence composed by the δ sub-process (executed at least once) followed by the corresponding ConGolog *while loop*.

Intermediate message event. The throw intermediate message event is represented in the action theory by the primitive action $\epsilon_t(id, \vec{x})$ and the fluent $Done(\epsilon_t(id, \vec{x}), s)$, which indicates the completion of both sending and receiving the corresponding message and data objects $\vec{x}$ of type β. Instead, the action $\epsilon_t(id, \vec{x})$ is executable if $\beta(id, \vec{x}, s)$ holds for its required input data objects in this process instance id. The effect of $\epsilon_t(id, \vec{x})$ is to set the associated relational fluent to true in the subsequent situation, whereas initially, this fluent is assumed to be false. In the high-level program, this event is implemented by executing the primitive action $\epsilon_t(id, \vec{x})$.

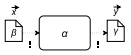
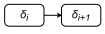
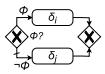
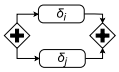
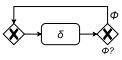
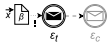
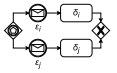
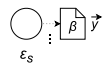
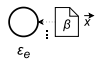
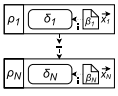
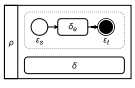
Event-based gateway. The event-based gateway, which triggers the execution of the first branch whose catch intermediate event is activated, is translated in the action theory through a fluent $Done(\epsilon_i(id), s)$ for each catch intermediate event. In the high-level program, this predicate is tested to determine the subsequent execution of the corresponding sub-process δ_i. Each event-based branch, consisting of such a test and its associated sub-process in sequence, is placed within a nondeterministic choice of actions along with the other sequences.

Start event. The start event of a BPMN process model is represented in the action theory by an exogenous action $\epsilon_s(id, \vec{y})$, which initiates the execution of a process instance identified by id, producing the data objects $\vec{y}$ as output. This action is executable whenever the instance id is not active in a given situation s, as specified by the $\neg Active(id, s)$ fluent, and $\beta(id, \vec{y}, s)$ is satisfied for its data objects. The execution of $\epsilon_s(id, \vec{y})$ has the effect of setting $Done(\epsilon_s(id, \vec{y}), s)$ (false in the initial situation S_0) to true, indicating the action's completion. Additionally, $\epsilon_s(id, \vec{y})$ makes true $Active(id, s)$, signaling that a process instance is now running, and waiting for the pool ρ for processing, by making $Waiting(id, \rho, s)$ hold.

End event. Conversely, the end event is modeled in the action theory as a primitive action $\epsilon_e(id, \vec{x})$, which takes the data objects $\vec{x}$ as input. This action can be executed whenever the process instance being completed is active, as indicated by the $Active(id, s)$ fluent. The effects of ϵ_e are to mark its completion by setting the fluent $Done(\epsilon_e(id, \vec{x}), do(a, s))$ to true, which is initially false, and to conclude the process instance by setting $Active(id, s)$ to false.

Pool. BPMN pools ρ_i ($i \in \{1, \ldots, N\}$) are interpreted as servers implemented through ConGolog procedures that are in a concurrent iteration construct. This runs a new instance of the process concurrently whenever the start condition is satisfied. Specifically, the procedure δ_i selects non-deterministically a process instance id that is waiting, acquires it, and handles it with a procedure $handle_i(id)$. All pool procedures are executed concurrently in the overall controller, which employs prioritized concurrency. To ensure that the pools do not terminate prematurely, we include at the lowest priority a busy wait mechanism that can be terminated by shutting down the servers. In the action theory, the primitive action $acquire(id, \rho)$ is used to acquire an instance id for processing by the pool ρ. This action is executable whenever the $Waiting(id, \rho, s)$ fluent is true, indicating that the instance id is waiting for the pool ρ in the current situation s. $Waiting(id, \rho, s)$ becomes true when the proper start event triggers. Executing $acquire(id, \rho)$ changes the fluent to false, signaling that the instance is no longer waiting. The $handle_i(id)$ for a process instance depends on the specific BPMN model and may represent an arbitrarily complex procedure, managing the control flow within the sub-process δ_i. Additionally, the theory includes an exogenous action $shut\_down$, which is always executable. This action sets the relational fluent $Stopped(s)$ that governs the interrupt to true, halting the pool servers and preventing them from processing further process instances. Nonetheless, there are scenarios where a pool serves only one request at a time: in such cases, the pool can be modeled in the program as a *while loop* conditioned on waiting for new instances to arrive.

Table 2
Formalization of major BPMN patterns in situation calculus.

PATTERN	HIGH-LEVEL PROGRAM	BASIC ACTION THEORY
	$(\alpha_s(id,\vec{x});$ $Done(\alpha_s(id,\vec{x}))?)$	$\alpha_s, \alpha_e \in \mathcal{A}.\ Exog(\alpha_e).\ \forall x \in \vec{x}.\beta(id,x,s), \forall y \in \vec{y}.\gamma(id,y,s),$ $Running(\alpha_s(id,\vec{x}),s), Done(\alpha_s(id,\vec{x}),s) \in \mathcal{F}.$ $Poss(\alpha_s(id,\vec{x}),s) \equiv \neg Running(\alpha_s(id,\vec{x}),s) \wedge \beta(id,\vec{x},s).$ $Poss(\alpha_e(id,\vec{y}),s) \equiv Running(\alpha_s(id,\vec{x}),s) \wedge \gamma(id,\vec{y},s).$ $a = \alpha_s(id,\vec{x}) \supset Running(\alpha_s(id,\vec{x}), do(a,s)).$ $a = \alpha_e(id,\vec{y}) \supset Done(\alpha_s(id,\vec{x}), do(a,s)).$ $Running(\alpha_s(id,\vec{x}),S_0) = false.$ $Done(\alpha_s(id,\vec{x}),S_0) = false.$
	$(\delta_i; \delta_{i+1})$	-
	**if** ϕ **then** δ_i **else** δ_j **endIf**	-
	$\delta_j \,||\, \delta_k$	-
	$(\delta;$ **while** ϕ **do** δ **endWhile**$)$	-
	$\epsilon_t(id,\vec{x})$	$\epsilon_t \in \mathcal{A}.\ \ Done(\epsilon_t(id,\vec{x}),s), \forall x \in \vec{x}.\beta(id,x,s) \in \mathcal{F}.$ $Poss(\alpha_s(id,\vec{x}),s) \equiv \beta(id,\vec{x},s).$ $a = \epsilon_t(id,\vec{x}) \supset Done(\epsilon_t(id,\vec{x}), do(a,s)).$ $Done(\epsilon_t(id,\vec{x}),S_0) = false.$
	$(Done(\epsilon_i(id))?; \delta_i) \mid$ $(Done(\epsilon_j(id))?; \delta_j)$	$Done(\epsilon_i(id),s), Done(\epsilon_j(id),s) \in \mathcal{F}.$
	-	$\epsilon_s \in \mathcal{A}.\ Exog(\epsilon_s).\ Active(id,s), \forall y \in \vec{y}.\beta(id,\vec{y},s),$ $Done(\epsilon_s(id,\vec{x}),s), Pool(\rho,s), Waiting(id,\rho,s) \in \mathcal{F}.$ $Poss(\epsilon_s(id,\vec{x}),s) \equiv \neg Active(id,s) \wedge \beta(id,\vec{y},s).$ $a = \epsilon_s(id,\vec{x}) \supset Active(id, do(a,s)).$ $a = \epsilon_s(id,\vec{x}) \supset Done(\epsilon_s(id,\vec{x}), do(a,s)).$ $a = \epsilon_s(id,\vec{x}) \supset Waiting(id,\rho, do(a,s)).$ $Active(id,S_0) = false.\ \ Done(\epsilon_s(id,\vec{x}),S_0) = false.$ $Waiting(id,\rho,S_0) = false.$
	$\epsilon_e(id,\vec{x})$	$\epsilon_e \in \mathcal{A}.\ Active(id,s), Done(\epsilon_e(id,\vec{x}),s), \forall x \in \vec{x}.\beta(id,\vec{x},s) \in \mathcal{F}.$ $Poss(\epsilon_e(id,\vec{x}),s) \equiv Active(id,s) \wedge \beta(id,\vec{x},s).$ $a = \epsilon_e(id,\vec{x}) \supset \neg Active(id, do(a,s)).$ $a = \epsilon_e(id,\vec{x}) \supset Done(\epsilon_e(id,\vec{x}), do(a,s)).$ $Done(\epsilon_e(id,\vec{x}),S_0) = false.$
	**proc** ρ_i $\delta_i^{||}$ **endProc** **proc** δ_i $\pi id.(acquire(id,\rho_i); handle_i(id))$ **endProc** **proc** ρ $\rho_1 \,||\, \ldots \,||\, \rho_N$ $\rangle\rangle$ $\langle \neg Servers\_stopped \rightarrow true?\rangle$ **endProc**	$acquire, shut\_down \in \mathcal{A}.\ \ Exog(shut\_down).$ $Servers\_stopped(s), Pool(\rho,s), Waiting(id,\rho,s) \in \mathcal{F}.$ $Poss(acquire(id,\rho),s) \equiv Pool(\rho,s) \wedge Waiting(id,\rho,s).$ $a = acquire(id,\rho) \supset \neg Waiting(id,\rho,do(a,s)).$ $Poss(shut\_down,s) \equiv \neg Servers\_stopped(s) \wedge$ $\neg(\exists id.Active(id,s)).$ $a = shut\_down \supset Servers\_stopped(s).$ $Servers\_stopped(S_0) = false.$
	**gexec**(ϕ, δ); **if** $\neg\phi$ **then** $(\delta_e; \epsilon_t)$ **else** $()$ **endIf**	$\epsilon_s, \epsilon_t \in \mathcal{A}.\ Exog(\epsilon_s).\ \ Active(id,s), Done(\epsilon,s) \in \mathcal{F}.$ $Poss(\epsilon_s(id),s) \equiv true.\ \ Poss(\epsilon_t(id),s) \equiv Done(\delta_e,s).$ $a = \epsilon_s(id) \supset \neg\phi(do(a,s)).$ $a = \epsilon_s(id) \supset Done(\epsilon_s(id), do(a,s)).$ $a = \epsilon_t(id) \supset Done(\epsilon_t(id), do(a,s)).$ $a = \epsilon_t(id) \supset \neg Active(id, do(a,s)).$ $Done(\epsilon_s(id),S_0) = false.\ \ Done(\epsilon_t(id),S_0) = false.$

4.1. Handling BPMN global exceptions

A standard feature of BPMN is that of *global exceptions* (last row of Table 2). Global exceptions can trigger at any point during the process execution and, in many real scenarios (like the one described later in our case study), they are used to abort the current active pool execution.

Surprisingly, the existing ConGolog-like languages do not support the graceful "*abortion*" of a running program. So, we extend the high-level language with a new construct for *guarded execution*, **gexec**(ϕ, δ), that executes program δ only as long as condition ϕ holds; if ϕ becomes false, the execution of the construct can immediately terminate. We have also incorporated it in the Prolog implementation of IndiGolog [11] that we shall use for our case study later on.

The following axioms characterize the predicates $Trans$ and $Final$ for the new guarded execution construct:

$$Trans(\textbf{gexec}(\varphi,\delta), s, \delta', s') \equiv \varphi[s] \wedge \exists \delta''.Trans(\delta, s, \delta'', s') \wedge \delta' = gexec(\varphi, \delta'')$$
$$Final(\textbf{gexec}(\varphi,\delta), s) \equiv \neg\varphi[s] \vee Final(\delta, s)$$

The last row in Table 2 shows how we leverage the new construct to account for global exceptions in BPMN. This block represents the exception handling procedure, structured as a standard BPMN flow with a start event ϵ_s, a sub-process δ_e, and a terminate end event ϵ_t, which concludes the execution of the overall process instance. Notably, the start event ϵ_s is modeled as an exogenous action that can be triggered at any point during execution. When triggered, it makes the condition $\phi(do(a,s))$ false and sets the relational fluent $Done(\epsilon_s, do(a,s))$ to true, signaling its completion. Subsequently, first δ_e is executed, and then the terminate end event ϵ_t is executed, represented by a primitive

action that updates its corresponding fluent $Done(\epsilon_t, do(a, s))$ to true and sets $Active(id, do(a, s))$ to false, thereby indicating the termination of the overall process instance.

It is important to notice that this extension addresses a critical limitation in existing process modeling formalisms. Indeed, in standard approaches like Petri nets, modeling a "global" interrupt often requires adding explicit transition arcs from every possible state of the process to the exception handler, leading to cluttered models and state-space explosion. By introducing the guarded execution construct for ConGolog at a logical level, we can capture this semantics natively, allowing the reasoner to handle the asynchronous termination of complex, multi-threaded activities (like the applicant and company pools running in parallel in our running example) without altering the structure of the main control flow.

5. Case study

We use the BPMN model in Fig. 2 to demonstrate our framework. The model details the execution flow of a *job application* process, involving two pools: the applicant and the company. We stress that this model is a realistic one that resembles real-world domains in its complexity and structure [1]. The complete formalization in situation calculus and ConGolog is presented in the appendix. We also provide the corresponding implementation[5] that can be executed through the IndiGolog interpreter to simulate the process behavior before its concrete enactment.

The process begins when the applicant initiates the job application process, marked by a start event. This is modeled in the action theory using the exogenous action JOB_NEEDED(id), where id identifies the specific process instance. This action sets the $Active(id, s)$ fluent to true, triggering the primitive action PREPARE_APPLICATION_START(id). The process remains in this state until the concluding exogenous action PREPARE_APPLICATION_END(id, app) produces an application data object app for the given process instance id. The applicant then submits the application, initiating the company's process with the action APPLICATION_SENT(id, app). From the high-level program perspective, the company waits for the $Done$(APPLICATION_SENT(id, app), s) fluent to evaluate to true before performing a validity check on the application.[6] This step involves the pair of actions CHECK_VALIDITY_START(id, app) and CHECK_VALIDITY_END($id, result$), where $result$ is a boolean indicating if the documents in the application are considered valid. This $result$ defines the value of the relational fluent $Documents\_ok(id, s)$, which governs a conditional statement in the program. In this exclusive gateway, if the documents are valid, the company organizes the application for further processing through the appropriate actions in the program. Conversely, if the documents are invalid, the company rejects the application, creating a letter of refusal data object with the pair of primitive and exogenous actions PRODUCE_LETTER_OF_REFUSAL_START(id) and PRODUCE_LETTER_OF_REFUSAL_END($id, lett$), and sending it to the applicant through LETTER_OF_REFUSAL_SENT($id, lett$). The process then concludes for the company with APPLICATION_ANALYSED(id) and for the applicant with APPLICATION_FINALISED(id), which sets $Active(id, s)$ to false, signaling the end of this process instance. Instead, if the documents are valid, the company organizes them and performs parallel tasks, such as assigning a contact partner to the applicant and verifying the candidate's prerequisites. This parallelism is modeled using the ConGolog concurrency operator. Subsequently, the application is evaluated to determine whether the applicant qualifies for an interview. This assessment is carried out using a conditional statement and the relational fluent $Interview(id, s)$, whose value is determined by the exogenous action CHECK_APPLICATION_FOR_INTERVIEW_END($id, result$). The boolean parameter $result$ specifies the truth value of $Interview(id, s)$ governing the exclusive gateway. If the candidate qualifies for an interview, the process advances to planning and conducting the interview, using the corresponding actions in the program. If not, the application is rejected as before. During the interview, the company evaluates the applicant's suitability for the final job offer. A positive outcome transitions the process to the approval phase, where the $Job\_offer(id, s)$ fluent governing the gateway is evaluated as true. On the other hand, a negative outcome redirects the flow to the application's rejection. Following the necessary approvals, the company validates the job offer by confirming its details before initiating contract production. These tasks follow the standard translation, involving a primitive action to start them and an exogenous action to complete them. Once the job offer is confirmed after the exclusive gateway ruled by the $Approval(id, s)$ fluent, the company produces the contract and sends it to the applicant (CONTRACT_SENT(id, con)). The applicant then signs and returns the signed contract, which is archived by the company through STORE_SIGNED_CONTRACT_START($id, sign\_con$) and STORE_SIGNED_CONTRACT_END(id). In the meantime, the applicant communicates the recruitment to his employment office through the primitive and exogenous actions COMMUNICATE_RECRUITMENT_START(id) and COMMUNICATE_RECRUITMENT_END(id). At this point, the process concludes, with $Active(id, s)$ set to false to indicate the instance completion.

Additionally, the applicant may choose to withdraw their application at any point. This is modeled using the guarded execution construct, which exits the main program upon the completion of the exogenous action WITHDRAWAL_BY_APPLICANT(id) for the relevant process instance. The rest of the global exception handling is modeled in the standard manner, utilizing a combination of primitive and exogenous actions to confirm and send the withdrawal request. Subsequently, the company processes the withdrawal and terminates the overall process.

In the program, the pools are represented as servers executed concurrently by the overarching controller procedure, $control(bpmn\_process)$. The execution is prioritized over the busy wait state. Each server procedure consists of a concurrent iteration, where it acquires the id of the request to process, along with the pool currently awaiting processing (indicated by $Waiting(id, pool\_name, s)$). This acquisition is performed through the primitive action ACQUIRE($id, pool\_name$). Once a request is acquired, the server executes the corresponding procedure (e.g., $applicant\_procedure(id)$) within a guarded execution, which is governed by a condition to monitor exceptions such as verifying $\neg Done$(WITHDRAWAL_SENT(id), s) for the applicant.

6. Exploiting our semantics for reasoning

Our BPMN formal semantics, by mapping process models to a ConGolog program operating on a *situation calculus* basic action theory, can support a wide range of analyses. These range from foundational logical queries inherent to the *situation calculus* itself to advanced semantical process analysis tasks. Our goal is to enable the use of existing and future reasoning techniques off-the-shelf on BPMN models, leveraging our framework's coverage of a substantial subset of the BPMN standard.

While standard BPMN analysis tools typically support syntactic checks (e.g., graph-based soundness), they lack the expressiveness to perform this kind of rich, data-aware semantic reasoning.

The *situation calculus* provides two foundational reasoning tasks: the *projection* task determines the truth value of fluents after the execution of a given sequence of actions, and the *legality* task finds if a given sequence of actions is executable [7]. Beyond these fundamental queries, our framework supports more complex, process-centric analyses that often build upon them, namely *conformance checking* [8] and *property verification*. These tasks can be implemented and executed using the existing IndiGolog interpreter. We now illustrate the use of our semantics for reasoning through four practical examples, using the situation calculus and ConGolog encoding presented in Section 5 (and fully reported in the Appendix) as the underlying knowledge base KB.

[5] Case Study Implementation: https://github.com/angelo-casciani/sitcalc4bpmn/tree/main/pl_models/case_study.

[6] Note that events in BPMN are referred in past participle tense, and we keep that convention in our corresponding actions.

```
?- eval(application(1),[prepare_application(end,1),prepare_application(start,1),acquire(1,applicant),job_needed(1)],true).
true .
```

```
?- indigolog([job_needed(1),acquire(1,applicant),prepare_application(start,1),prepare_application(end,1)]).
  ...
PROGRAM: Program has executed to completion!! History done:
        [prepare_application(end,1),prepare_application(start,1),acquire(1,applicant),job_needed(1)]
```

Fig. 3. Screenshots showing projection (top) and legality (bottom) tasks.

6.1. Projection

The *projection* task involves determining the truth value of fluents after a given sequence of actions is executed.

Definition 6.1 (*Projection Task*)**.** Suppose that $R(s)$ is a formula with a free situation variable s. To verify if $R(s)$ would be true after performing $(a_1, \ldots, a_n)$ in the initial situation, we determine whether or not

$$\mathcal{KB} \models R(\mathrm{do}(a_n, \mathrm{do}(a_{n-1}, \ldots, \mathrm{do}(a_1, S_0) \ldots)))$$

In the BPMN context, this directly translates to a *"what-if" analysis* with which we can formally ask if predicates representing specific data objects or conditions ruling exclusive gateways have their expected values after a particular path in the process is executed.

```
?- eval(application(1),[prepare_application(end,1),
   prepare_application(start,1),acquire(1,applicant),
   job_needed(1)],true).
```

This query asks whether the relational fluent *application(1)* holds after the given sequence of actions (in reversed order according to the $do(a, s)$ definition), verifying that an application data object for process instance $id = 1$ was created. The result, reported in the top image of Fig. 3, confirms that the application data object for the instance $id = 1$ was correctly initialized.

6.2. Legality

The *legality* task checks if a given sequence of actions is executable, i.e., whether all action preconditions ($Poss$ action precondition axioms) are satisfied at each step.

Definition 6.2 (*Legality Task*)**.** To find out if the sequence $(a_1, \ldots, a_n)$ can be legally performed in the initial situation, we determine whether or not

$$\mathcal{KB} \models \mathrm{Poss}(a_i, \mathrm{do}(a_{i-1}, \ldots, \mathrm{do}(a_1, S_0) \ldots))$$

for every i such that $1 \leq i \leq n$.

For BPMN, this allows for detecting unreachable tasks by checking if a seemingly valid control flow is actually *illegal* due to actions' preconditions not being met.

```
?- indigolog([job_needed(1),acquire(1,applicant),
   prepare_application(start,1),prepare_application(end,1)]).
```

The above command thus verifies whether the above sequence of activities, also considered for the projection task, is executable from the *applicant* perspective. The result, reported in the bottom image of Fig. 3, confirms that the action sequence is indeed executable as expected.

6.3. Conformance checking

One fundamental reasoning capability is *conformance checking*: determining whether a given execution trace (i.e., a *history* in ConGolog) is compliant with the BPMN process model. This can be done by instructing the IndiGolog interpreter to attempt to execute the history using the process model. If the trace cannot be reproduced, it indicates a deviation from the modeled process. Note that the trace need not be complete, and partial traces can be checked for compliance. Moreover, by performing incremental checks, one can identify the longest prefix of the trace that is compatible with the model and, therefore, find the action where conformance fails.

Fig. 4 shows a positive and negative conformance check of partial traces in our job application case study that reach the interview step in the company pool. The query:

```
?- H1 = [...], proc(sim(bpmn_process), E),
   trans_star(E, [], _, H1))).
```

asks whether program `bpmn_process` representing the process in Fig. 2, together with the simulation of exogenous actions, may eventually evolve to produce the fixed history `H1`.

The only difference between the first trace `H1` and the second trace `H2` is in the validity check for the application (driving the flow in the exclusive gateway "`Documents OK?`" in Fig. 2). While the check passes in history `H1` (action `check_validity(end, 1, true)`), it fails in history `H2` (action `check_validity(end, 1, false)`). From the original process in Fig. 2, it is clear that the second trace `H2` is *not compliant*: an application should not be processed further if documents are not correct. Note that both checks take at most *one second*.

6.4. Properties verification

Another reasoning task that one may be interested in when reasoning about BPMN processes is whether a certain condition may arise in *some* legal execution of a process. In our case study, one may want to check whether a contract could end up signed, with the application then being withdrawn, preventing it from being finalized. Thus, highlighting a potential flaw in the process design. This amounts to checking whether there is an execution of the following ConGolog program (where δ is the job application business process) that we shall call `property_verification`:

$$(\delta \parallel (\pi a.Exog(a); a)^*);$$
$$?(\exists id.Signed\_contract(id, s) \wedge \neg(Done(\textsc{application\_finalised}(id), s))).$$

The execution of this program, shown in the third screenshot of Fig. 4, returns a positive result: such a situation may arise. In Fig. 4 (bottom screenshot), the query:

```
?- do(property_verification, [], H).
```

aims to find a complete terminating execution history of the program (from the empty history `[]`) which is returned in (unified with) variable `H`.

This verification task highlights a scenario that is difficult to verify with traditional modeling approaches, as it calls for the interplay between data availability, concurrency, and asynchronous interrupts. Indeed, in our case study, the application withdrawal is a global exception that can occur concurrently with the company pool's execution.

When found, the witness history execution `H` identifies a "*race condition*" between the signing of the contract and the withdrawal

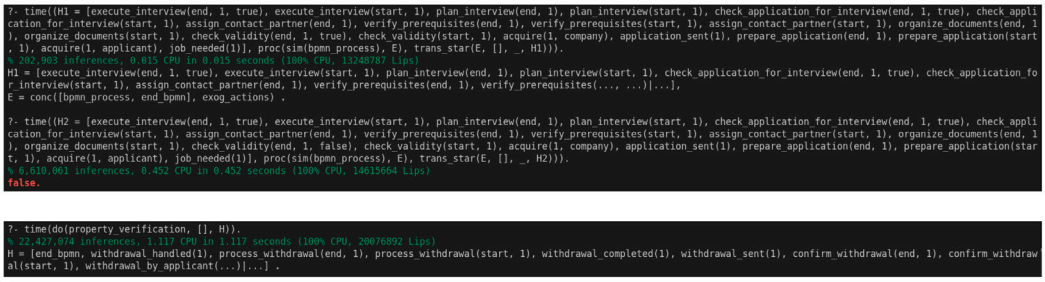


Fig. 4. Screenshots showing two reasoning tasks: conformance checking (top) and property verification (bottom). Note that all queries take at most 1 s.

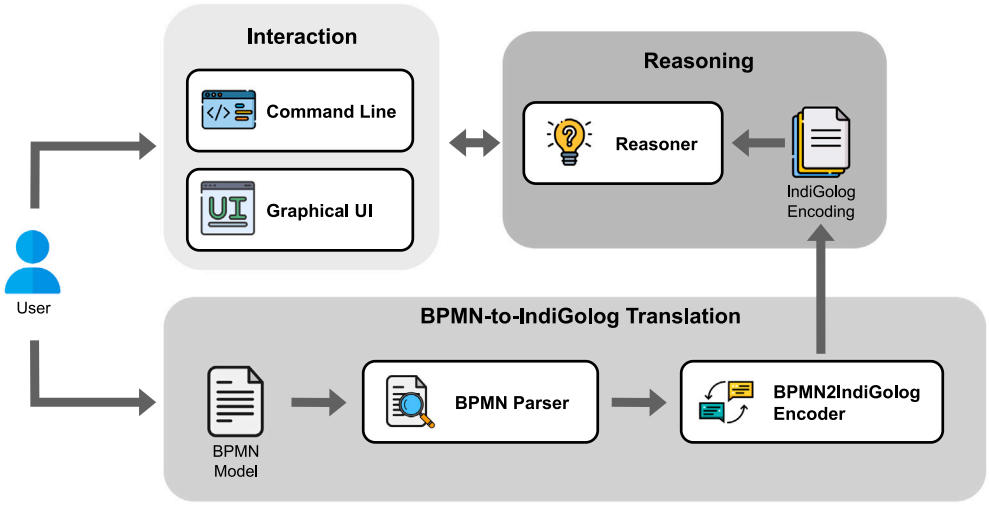


Fig. 5. The overall architecture of the translation and reasoning tool.

process. Standard verification often misses this or requires an extensive state-space expansion that becomes infeasible. The IndiGolog reasoner instead detects it by finding a legal execution trace where *Signed_contract(id,s)* holds, yet the process was legally interrupted by the withdrawal. This proves that the process model design is *unsafe* as it allows the company to commit to a contract even after the applicant has withdrawn the application, suggesting the need for a restriction of the withdrawal procedure to occur before the contract is signed to prevent unintended or inconsistent process states.

7. Tool support for translation and reasoning

To bridge the gap between our theoretical formalization and practical process analysis, we have developed a software tool[7] that automates the entire workflow, from BPMN translation in IndiGolog to reasoning over the generated formalization. The tool makes our encoding accessible also to users who may not be experts in KR&R, lowering the barrier to entry for applying formal semantic analysis and reasoning to BPMN processes.

As shown in the architecture detailed in Fig. 5, the tool is composed of three major stages: *BPMN-to-*IndiGolog *Translation*, *Reasoning*, and *Interaction*.

*BPMN-to-*IndiGolog *translation.* In this first stage, the user provides the BPMN process model as input to a BPMN parser component, which is in charge of extracting all the BPMN constructs relevant for the subsequent encoding, i.e., all the components represented in Table 2.

In particular, the *BPMN Parser* first reads the XML file and translates its structure into internal data structures. These intermediate data structures are then passed to the *BPMN-to-*IndiGolog *Encoder* to automatically generate the corresponding IndiGolog encoding according to the formalization introduced in Section 4. This encoding contains both the *situation calculus* basic action theory (defining actions, fluents, successor state axioms $\mathcal{D}_{ss}$, action precondition axioms $\mathcal{D}_{poss}$, and initial situation $\mathcal{D}_{S_0}$) and the high-level ConGolog program.

Reasoning. The encoding output from the previous stage is then leveraged by a *Reasoner* that is built upon the IndiGolog interpreter and allows for invoking reasoning tasks on this encoding programmatically. A user can submit reasoning queries based on the tasks discussed in Section 6. The Reasoner passes these queries to the IndiGolog interpreter, which executes them against the generated encoding of the process model.

Hence, users can run queries for:

- *projection*, to evaluate the truth value of a fluent after a sequence of actions.
- *legality*, to validate the executability of a specific execution path.
- *conformance checking*, to check a given trace against the model.
- *property verification*, to look for executions that satisfy or violate a given property.

Moreover, this reasoner also enables the *online execution* of the encoded process, leveraging the IndiGolog Tkinter interface to issue exogenous events and carry out the execution by allowing the evaluation of each decision point.

[7] GitHub repository of the tool: https://github.com/angelo-casciani/sitcalc4bpmn.

Interaction. To respond to different user workflows and expertise levels, the tool provides two interfaces for interaction: a *Command-Line Interface* (CLI) and a *Graphical User Interface* (GUI).

The CLI is designed to make the translation and reasoning functionalities available programmatically. This is particularly useful for the integration of this tool and encoding in larger ABPMS architectures to support their *reasoning* stage.

On the other hand, the Web-based GUI provides a user-friendly, interactive environment where the user can translate and reason over process models in an accessible manner. Moreover, it allows for performing reasoning tasks using a simplified and more intuitive syntax, handles the interaction with the underlying Prolog engine, and presents the results back to the user in an accessible format.

Fig. 6 shows two screenshots taken from the GUI.

The GUI provides four different tabs:

- In the "*Translation*" tab, the user can directly upload their BPMN process model and translate it to IndiGolog. Moreover, they can also visualize the uploaded BPMN and inspect the generated IndiGolog encoding in Prolog, as shown in the top image of Fig. 6.
- The "*Load Existing Model*" tab allows the user to access and reload previously translated models.
- The "*Reasoning Tasks*" tab executes reasoning over the generated encoding of the BPMN process model. It allows for the selection of the reasoning task to perform and the input parameters based on the selected task in a simplified syntax, and also the visualization of the results. Fig. 6 (bottom) shows the results of an online execution for the job application process model considered in our case study.
- There is also a "*Quick Guide*" tab to provide an easy overview for the user on how to use the UI and instruct them on how to leverage the various reasoning tasks for process analysis.

We implemented the tool in Python using the libraries *Gradio*[8] for the GUI and *PM4Py*[9] for the visualization and parsing of the BPMN process model. For the reasoner, we used the IndiGolog interpreter[10] implemented in Prolog.

8. Experimental evaluation

This section discusses the experiments conducted to evaluate the correctness and performance of the translation and of the reasoning tasks performed using the encoding introduced in Section 4 and the software for the automated translation and for the execution of the reasoning tasks detailed in Section 7. To this end, in the following subsections, we describe the process models used for the tests, the experimental setup, and the results.

8.1. Datasets

Our evaluation is performed on two distinct datasets to test different aspects of our formalization and reasoning tasks.

The first dataset is the one introduced in [32,33]. It provides a collection of 24 textual process descriptions in different granularities, each associated with BPMN models that represent possible modeling solutions for the associated description (from 8 to 11 models per description). Each BPMN model is assigned a score from 0 to 5, reflecting a human expert's assessment of the solution's quality. We refer to this as the "*Textual BPMN*" dataset.

We pre-processed this archive to select only the models reporting the maximum score (i.e., a score of 5), resulting in a final dataset of 89 BPMN models.[11]

The BPMN models in this collection are characterized by complex control-flow structures, making them ideal for evaluating the *legality* and *conformance* tasks. Thus, to create a benchmark for these reasoning tasks, we performed the *playout* of each model, generating a set of ordered sequences of executed activities and events that constitute valid execution traces. We generated 8 traces per model: 4 for legality and 4 for conformance. Specifically, for the legality task, 2 traces are legal and 2 are not; the same distribution applies to the conformance checking samples. The illegal or non-conformant traces were obtained by randomly swapping one (or more) pairs of activities within a valid trace. This process resulted in an automatically generated dataset of 712 samples.[12] The composition of a sample follows the structure $\langle sample\_id, model\_name, task\_type, actions, expected\_result \rangle$, where:

- $sample\_id$ reports the sample identifier within the dataset;
- $model\_name$ identifies the BPMN model selected for this evaluation;
- $task\_type$ specifies the task (legality or conformance checking);
- $actions$ reports the sequence of actions to consider for the reasoning task;
- $expected\_result$ reports the expected boolean outcome of the task for this sequence.

On the other hand, for the evaluation of projection and property verification, we gathered a collection of 10 BPMN models from modeling exercises in the BPM Master's course held at Sapienza University of Rome.[13] A key feature of this dataset is the detailed inclusion of data objects and data-flow, making it the designated benchmark for evaluating the data-centric *projection* and *property verification* tasks. We refer to this as the "*Exams BPMN*" dataset.

For this collection, we manually performed a semantic analysis of each model to derive a dataset of 40 samples.[14] For projection, we defined fluents related to data objects to check if a sequence makes them true or false. For property verification, we defined relevant properties involving both control flow and data objects to test against the entire process model. The composition of a sample in this case follows the structure $\langle sample\_id, model\_name, actions, property, expected\_result \rangle$, where:

- $sample\_id$ reports the sample identifier within the dataset;
- $model\_name$ identifies the BPMN model selected for this evaluation;
- $actions$ reports the sequence of actions to consider for the reasoning task;
- $property$ is either the fluent related to a data object (for projection) or a composed property (a conjunction of control flow and data object conditions) to check via verification;
- $expected\_result$ reports the expected boolean outcome of the task.

As a preliminary step, we pre-computed a set of structural metrics for all models in both datasets, as detailed in our evaluation framework. These metrics include the count of tasks, exclusive gateways,

[8] Gradio Web page: https://www.gradio.app/.

[9] PM4Py documentation: https://processintelligence.solutions/static/api/2.7.17/index.html.

[10] IndiGolog repository: https://github.com/ai-krml-uoft/indolog.

[11] The resulting collection of BPMN process models obtained from the Textual BPMN dataset is available at: https://github.com/angelo-casciani/sitcalc4bpmn/tree/main/bpmn/dataset/processed.

[12] The generated samples for legality and conformance checking are available at: https://github.com/angelo-casciani/sitcalc4bpmn/blob/main/evaluation/datasets/samples_leg_conf.csv.

[13] We make the dataset available at: https://github.com/angelo-casciani/sitcalc4bpmn/tree/main/bpmn/exams-bpmn.

[14] The obtained dataset is available at: https://github.com/angelo-casciani/sitcalc4bpmn/blob/main/evaluation/datasets/samples_proj_verif.csv.

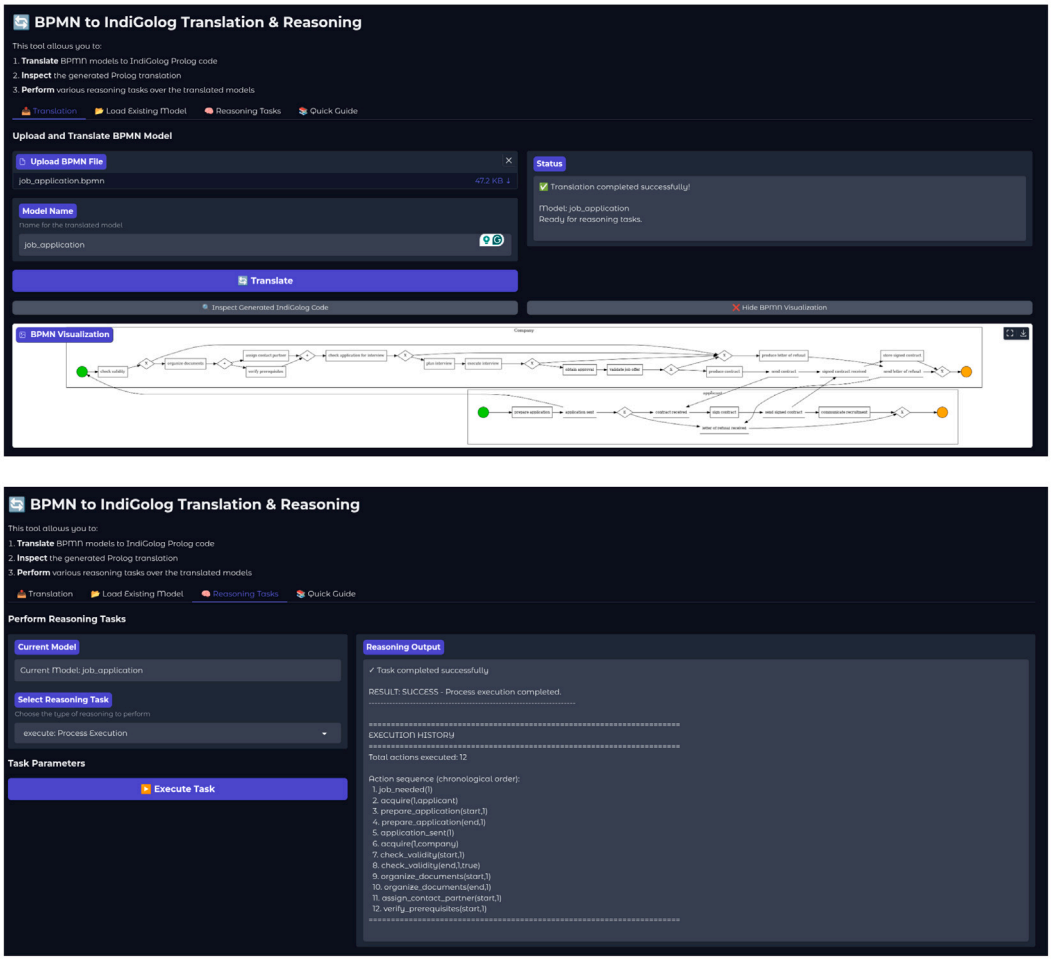


Fig. 6. Screenshots from the GUI. The GUI translation tab (top) allows for uploading, visualizing the BPMN model, and inspecting the generated IndiGolog encoding. The reasoning tasks tab (bottom) enables the execution of the reasoning tasks over the encoding of the BPMN process model.

Table 3
Average metrics of the BPMN process models contained in the datasets considered for evaluation.

Dataset	Pools	Tasks	Sub-processes	XOR	AND	Events	Data objects	Total elements
Textual	1.0	14.1	0.0	7.4	2.7	2.8	0.5	27.5
Exams	2.2	12.0	1.1	4.7	1.7	19.0	8.8	47.3

parallel gateways, events, pools, sub-processes, and data objects. Table 3 presents the average metrics for both datasets.

This statistical characterization highlights the distinct structural nature of the two datasets, justifying their specific usage. As shown in Table 3, the *Textual BPMN* dataset exhibits a significantly higher number of gateways (averaging 7.4 exclusive and 2.7 parallel gateways per model) compared to the *Exams BPMN* dataset, while possessing a negligible number of data objects (0.5 per model on average) and low degree of process collaboration (including a single pool per model). This complexity in decision points without data dependencies makes it the optimal choice for testing control-flow tasks like *legality* and *conformance*. Conversely, the *Exams* dataset is defined by a higher number of data objects (8.8 on average), events (19.0), and pools (2.2), confirming its suitability for the data-aware properties required by *projection* and *verification*.

8.2. Experimental settings

The primary objectives of our experimental evaluation are twofold: first, to assess the computational feasibility and scalability of our automated BPMN-to-IndiGolog translation; and second, to quantify the semantic correctness and performance of the four reasoning tasks (legality, conformance, projection, and property verification).

All experiments were conducted on a Ubuntu Linux workstation equipped with a 14th Gen Intel Core i9-14900HX processor, 32 GB of DDR5-5600 MHz RAM, and an NVIDIA GeForce RTX 4090 GPU with 16 GB of GDDR6 VRAM. Our evaluation framework, implemented in Python 3.10.12, interfaces with the IndiGolog interpreter (November 2025 version) running on SWI-Prolog.[15]

To improve statistical reliability and mitigate performance fluctuations, each translation and reasoning task was executed five times, and we report the averaged results. Finally, to provide a consistent reference for assessing scalability, we defined a *complexity score* for each process model, calculated as the sum of its total elements (i.e., tasks, sub-processes, gateways, events, and data objects). This metric is used as the x-axis in our performance evaluation charts to rank models by their structural size.

[15] SWI-Prolog Website: https://www.swi-Prolog.org/.

*BPMN-to-*IndiGolog *translation.* This evaluation phase assesses the computational feasibility and scalability of our automated translation of BPMN process models into the IndiGolog encoding. Its goal is to demonstrate that the translation is efficient and that the size of the resulting logical theory scales manageably with the complexity of the input model.

We executed the translation for all models across both the *Textual* and *Exams* BPMN datasets. To quantify performance and the complexity of the output, we measured:

- the *translation time* (in *seconds*);
- the number of generated *action* declarations, including both primitive and exogenous actions;
- the number of generated *fluent* declarations;
- the total *program size in lines of code* (LOC).

We also analyzed the relationship between the time required to produce translations of the BPMN models and other metrics by computing the *Pearson correlation coefficient* [34].

Reasoning. The reasoning evaluation is designed to verify the semantic correctness of our formalization and to assess its computational performance. This evaluation is *two-fold*, aligning with the distinct characteristics of our datasets, as described in Section 8.1:

- *Control-flow evaluation:* This was performed on the benchmark derived from the *Textual BPMN* dataset, focusing on the *legality* and *conformance checking* tasks. These tasks primarily stress the interpreter's ability to unfold the process model's complex control-flow and correctly resolve decision points.
- *Data-property evaluation:* This was conducted on the samples defined over the models from the *Exams BPMN* dataset, focusing on the *projection* and *property verification* tasks. These tasks specifically evaluate the framework's capacity to reason about data objects and properties regarding both data and control-flow.

For both evaluations, we collected a comprehensive set of metrics. To assess semantic correctness, we formulated the evaluation as a binary classification problem, identifying *True Positives* (TP), *True Negatives* (TN), *False Positives* (FP), and *False Negatives* (FN) based on the alignment between the IndiGolog interpreter's output and the ground truth.

Specifically, the reasoning engine returns a boolean value for all tasks:

- For *legality* and *conformance checking*, the system returns `true` if the execution trace is legally executable or conformant to the model, respectively.
- For *projection*, the system returns `true` if the queried data object fluent holds after the considered action sequence.
- For *property verification*, the system returns `true` if the specified property holds for the process model.

Accordingly, a result is categorized as a TP when the reasoner correctly returns *true* for a sample expected to be true (e.g., a valid trace), and as a TN when it correctly returns false for a negative sample (e.g., a trace with swapped actions). Conversely, an FP occurs if the system evaluates a negative sample as true, and an FN occurs if it fails to validate a positive sample. Based on these definitions, we measured *accuracy*, *precision*, *recall*, and *F1-score*.

To assess computational performance, we measured the *reasoning time* (in *seconds*) and the total number of *logical inferences* reported by the IndiGolog interpreter.

8.3. Evaluation results

We now present the results of our experiments, beginning with the analysis of the automated translation, followed by the correctness and performance of the reasoning tasks over the previously introduced BPMN datasets.

Table 4
Average translation metrics for the BPMN datasets.

Dataset	Avg. time (ms)	Avg. actions	Avg. fluents	Avg. size (*LOC*)
Textual	4.50	34.3	8.4	261.7
Exams	5.18	40.3	9.6	318.1

8.3.1. *BPMN-to-*IndiGolog *translation*

The initial phase of our evaluation assesses the computational feasibility and scalability of the automated BPMN-to-IndiGolog translation. We analyzed all 99 models from both datasets, measuring the translation time and the complexity of the resulting logical encoding in terms of generated actions, fluents, and program size (in terms of Prolog LOC). Table 4 summarizes the average translation metrics observed across the two datasets.

For the *Textual BPMN* dataset, the translation was highly efficient. As reported in Table 4, the average translation time was merely 4.5 ms, with a median of 3.86 ms and a maximum of only 11.8 ms. This demonstrates that the translation step is not a computational bottleneck, even for the articulated control-flow structures represented in this dataset.

More importantly, the correlation analysis based on the Pearson coefficient for this large dataset confirms that the translation time scales predictably with model complexity. We found statistically significant positive correlations between the translation time and the number of generated actions ($r = +0.507, p \ll 0.001$), fluents ($r = +0.670, p \ll 0.001$), and total program size ($r = +0.625, p \ll 0.001$). This indicates that as models become more complex, the translation time increases in a predictable manner. Fig. 7 (top) plots the translation time for each of the 89 models, confirming the low cost of the translation process.

Similar performance was observed for the encoding of the BPMN models in the *Exams* dataset. As shown in Table 4, such encodings are larger on average (318.1 LOC, 40.3 actions, and 9.6 fluents) compared to the ones generated for the *Textual* dataset. This was expected given the wide variety in terms of BPMN elements used by the process models contained in such a dataset and demonstrated by Table 3. However, despite this increased complexity, the translation remained fast, averaging 5.18 ms with a maximum of 7.15 ms.

The correlation analysis on this smaller dataset (including 10 process models) showed a statistically significant relationship between translation time and the total program size ($r = +0.838, p = 0.0025$). Such a strong connection, reinforced by the data in Fig. 7 (bottom), confirms that the translation scales with the overall size of the generated encoding. On the other hand, the correlations between translation time and counts of actions and fluents were not statistically significant (with $p > 0.05$), as expected given the small sample size.

Regarding the *quality* of the generated encodings, while the automated translation successfully produced output for all models, it relies on the input BPMN adhering to standard modeling conventions [3]. Specific models in the datasets deviated from these standards, for instance, by omitting the requirement to have an activity immediately preceding an exclusive gateway to determine the evaluation of its condition. In these cases, the tool generated syntactically valid encodings, but they could not reconcile the non-standard structural patterns. As detailed in the next section, these translation-level discrepancies were the main cause of failures in reasoning tasks, rather than timeouts.

Thus, these results on both datasets collectively confirm that our automated BPMN-to-IndiGolog translation is computationally feasible, adding only a negligible overhead.

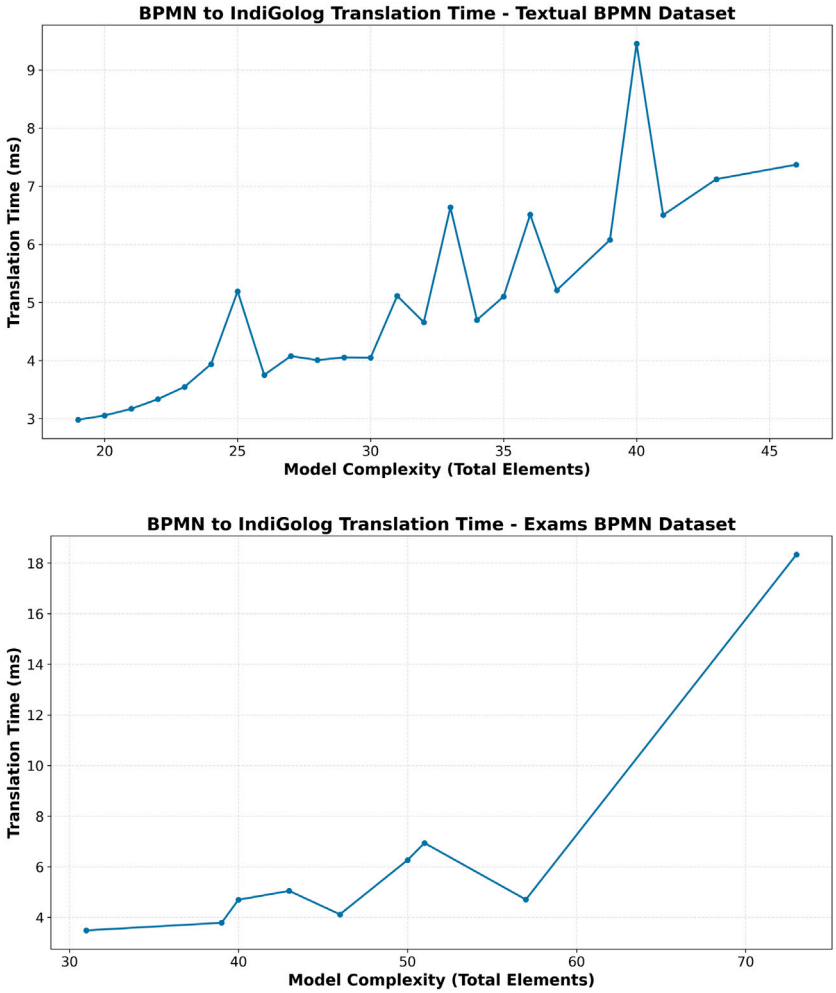


Fig. 7. BPMN to IndiGolog translation time (in ms) for BPMN model complexity (in terms of total number of elements) in the *Textual* (top) and *Exams* (bottom) datasets.

8.3.2. Reasoning

The second step of the evaluation focused on the semantic correctness and computational performance of the reasoning tasks. We discuss the results separately for the two evaluation settings.

Control-flow evaluation. Table 5 summarizes the results for the *Textual* BPMN dataset. Our tool demonstrated robust performance in reasoning tasks over processes characterized by intricate control-flow structures (including a high number of exclusive and parallel gateways).

The *legality* task achieved an aggregate accuracy of 91.1% with a negligible average reasoning time of 26 ms. The *conformance checking* also performed well, with a high precision. However, it exhibited a slightly lower recall and a significantly higher logical inference count compared to the legality task. This increase is attributable to the search required to compare the evaluation trace against the valid execution paths possible according to the process model.

It is important to highlight that the IndiGolog reasoner functioned correctly in all cases, and no timeouts were reached during this evaluation. The observed incorrect results that are preventing perfect scores for all metrics are attributable to specific BPMN models in the dataset that do not fully adhere to the standard modeling conventions assumed by our automated translation [3]. These deviations resulted in generated encodings that could not reconcile certain non-standard structural patterns with our formalization, rather than indicating failures in the reasoning engine itself.

Fig. 8 presents a comprehensive view of the reasoner's behavior across the 89 process models. The *accuracy* chart (top) reveals that, for the majority of models, the reasoner achieves perfect accuracy. The drops usually correlate across both tasks, suggesting that specific structural ambiguities within those process models posed shared challenges for both legality and conformance checking.

The *reasoning time* (middle) and *inferences* (bottom) plots instead provide insight into the computational cost for these tasks. The *legality* task (i.e., the orange line in the chart) remains flat across both metrics, confirming that checking the preconditions of a single execution trace is a constant-time operation with respect to the model size. In contrast, the *conformance checking* (i.e., the blue line) exhibits significant variance, with spikes that align both in the measured reasoning time and number of inferences for specific processes. This correlation confirms that the increased time is not due to translation overhead, but rather to the explosion of the search space: when validating a trace against a model with high concurrency (e.g., multiple parallel gateways), the IndiGolog interpreter must explore a factorial number of valid action interleavings. Despite these peaks, our reasoner handles the complexity well, maintaining an acceptable reasoning time for the majority of cases.

Data-property evaluation. Table 6 presents the results for the *Exams BPMN* dataset, selected for its wide usage of data objects and collaborations. These experiments reveal the impact of data-aware reasoning on both correctness and computational cost.

Table 5
Average performance results for the control-flow evaluation on the Textual BPMN dataset. The reasoning time is reported in *seconds*.

Task	Evaluated samples	Timeout	Accuracy	Precision	Recall	F1-score	Avg. time	Avg. inferences
Legality	384/384	0/384	91.1%	92.0%	90.1%	91.1%	0.026	114,954
Conformance	384/384	0/384	88.3%	97.4%	78.6%	87.0%	0.050	662,876

Table 6
Average performance results for the data-property evaluation on the Exams BPMN dataset. The reasoning time is reported in *seconds*.

Task	Completed samples	Timeout	Accuracy	Precision	Recall	F1-score	Avg. time	Avg. inferences
Projection	20/20	0/20	100.0%	100.0%	100.0%	100.0%	0.001	2488
Verification	17/20	3/20	94.1%	100.0%	85.7%	92.3%	1.002	16,247,482

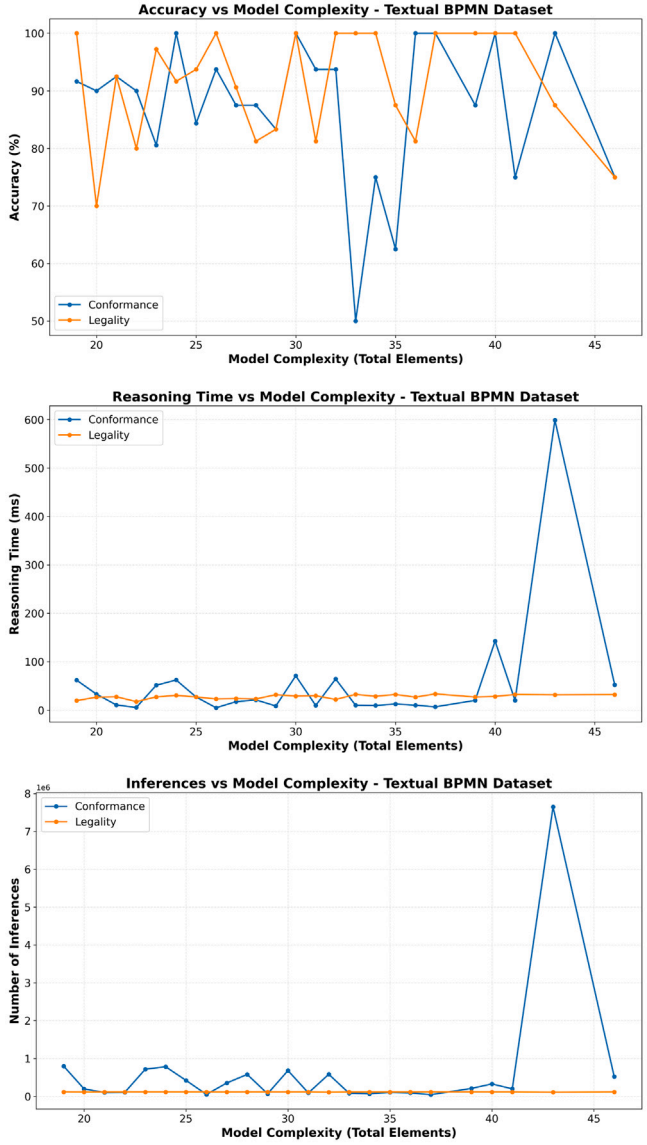


Fig. 8. Performance analysis of legality task and conformance checking on the Textual BPMN dataset: accuracy percentage (top), reasoning time in ms (center), and number of logical inferences (bottom) with respect to the BPMN model complexity. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

The *projection* task proved to be computationally trivial. As shown in Table 6, it required an average of just 1 ms and only 2488 inferences per sample, while achieving perfect scores across all metrics. This confirms that determining the value of a fluent after a fixed historical sequence is a highly efficient and sound operation, scaling linearly regardless of the underlying data model's complexity.

Conversely, the *property verification* exhibited a higher computational cost, with an average reasoning time of ∼1 s and over ∼16 *million* inferences per sample. This increase is directly attributable to the nature of the *Exams* dataset. Unlike projection, verification requires the reasoner to explore the state space of all possible process executions to validate the data-aware properties.

Fig. 9 offers a granular view of the performance across the models. The *accuracy* plot (top) highlights that the projection (i.e., the blue line) is always correct and stable. In contrast, the property verification (i.e., the orange line) reports failures for three out of ten process models.

The *reasoning time* (middle) and *inferences* count (bottom) charts reveal the causes of these failures. Considering that we imposed a *30-second timeout* for all reasoning tasks, as indicated by the red crosses in the charts, the reasoner timed out for process models 3 (once) and 9 (twice) in the property verification. In particular, they represent the most computationally intensive scenarios in the considered dataset. Process model 3, which represents the interaction between a work seeker and a hiring company, features extensive concurrency with multiple parallel gateways (each with three branches), loops, a global exception, and repeated message exchanges between two pools. All these factors contribute to significantly increasing the complexity of reasoning. On the other hand, model 9 depicts an order-to-cash process involving a customer and a bike producer, featuring parallel manufacturing branches with event-driven sub-processes for automated order cancellation following a payment timeout. In these specific cases, the reasoner must explore a factorial number of valid action interleavings to verify the property, resulting in a combinatorial explosion of the state space that prevented it from reporting a result within the imposed time limit. Only one actual logical error occurred: for the BPMN model 6, the reasoner returned an incorrect result (we can indeed observe an accuracy drop in the corresponding plot). This suggests a complex case where the property verification was completed but yielded a false negative due to specific data-flow translation issues, always related to a process model that is not fully conformant to the standard modeling conventions assumed in our work [3]. Conversely, for the remaining models, the system effectively verified properties with manageable computational effort.

9. Discussion and threats to validity

In this section, we discuss the implications of our findings in relation to the RQs that guided the paper, while also acknowledging the limitations of the proposed encoding.

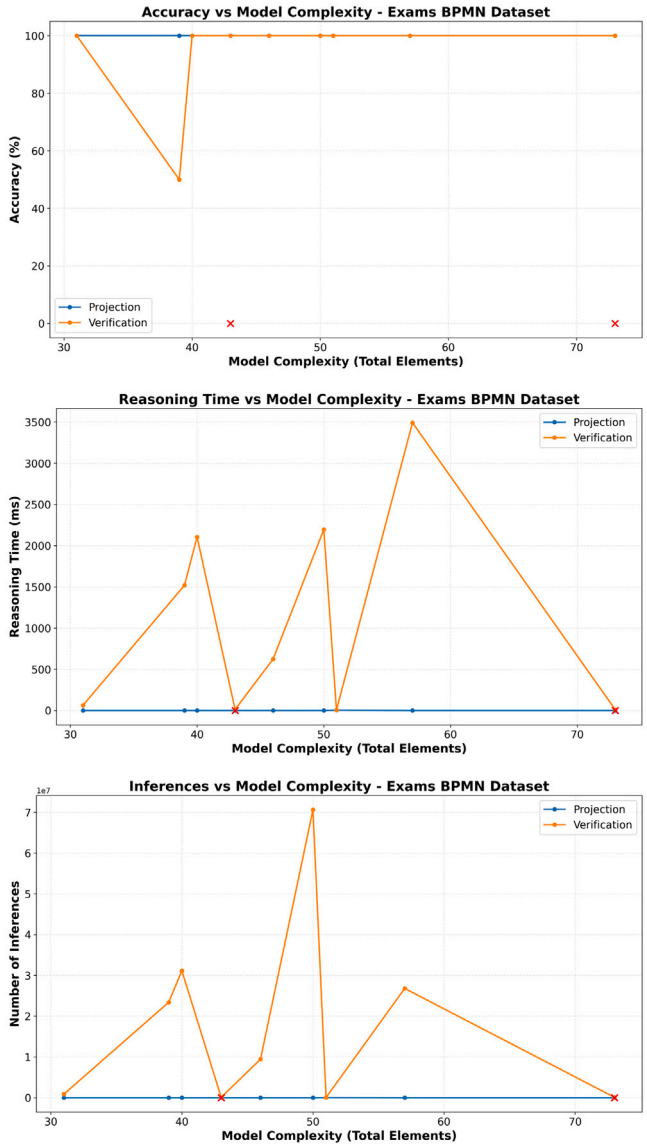


Fig. 9. Performance analysis of projection and property verification tasks on the Exams BPMN dataset: accuracy percentage (top), reasoning time in ms (center), and number of logical inferences (bottom) with respect to the BPMN model complexity. Points marked with a red cross (×) indicate timeouts (in this case 30 s). (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

Our objective is to provide a rigorous formalization of BPMN semantics for enabling formal reasoning on business process models. Regarding RQ1, our work demonstrates that KR&R languages, and in particular the situation calculus and ConGolog, provide a robust foundation for capturing BPMN semantics, directly encoding major BPMN patterns. Notably, the introduction of the *guarded execution* construct in ConGolog enables the modeling of *global exceptions* through the formalization of event-based subprocesses in a modular way, preserving the integrity of the main process flow while supporting native handling of interrupts.

In response to RQ2, the experimental results in Section 8 confirm that our formalization generally enables a multi-perspective semantic analysis in an efficient and tractable way. Our semantics supports data-centric tasks such as *projection* and *property verification* that are inherently out of reach for purely structural BPM analysis tools. Additionally, the reasoning over the case study in Section 6 demonstrates the ability of our encoding to support the detection of semantic errors typically overlooked in standard BPM systems, such as the "race condition" between contract signing and application withdrawal.

Despite these promising outcomes, there remains room for improvement, as our framework is subject to limitations regarding expressiveness, scalability, and generalizability.

While our encoding covers a significant subset of the BPMN 2.0 standard, it is not exhaustive. The current semantics does not explicitly render the *inclusive* (OR) *gateway*, and more fine-grained organizational constructs like *lanes* (representing sub-entities like departments). Other specific BPMN events and behaviors not covered include *timers* (currently simulated as standard tasks), *escalation* and *cancellation* events, *compensation* handling, *signal* events broadcasting across processes, and *conversation* diagrams. It is worth noting that while some constructs, such as *timers*, cannot be natively captured due to the discrete nature of the situation calculus (requiring its temporal extension [35]), others are straightforward to encode. For instance, *signal* events can be considered as a mere syntactic extensions, functioning as one-to-many message exchanges.

A threat to validity lies in the composition of high-level programs presented in Table 2, which relies on standard conditional and loop blocks, implying that our semantics is mainly applicable to well-structured BPMN models. Unstructured models with arbitrary sequence flows (e.g., a gateway followed by another without a decision task to define the branch condition) cannot be directly encoded with this framework without further manipulation. This consideration aligns with the translation issues observed in our evaluation (e.g., for model 6 of the Exam BPMN dataset), where deviations from the assumed modeling conventions [3] led to logical errors in reasoning.

From a computational perspective, while the translation phase is highly efficient and scales linearly, reasoning, and specifically property verification, is sensitive to state-space explosion. We explicitly acknowledge that while our framework scales well for standard processes, models with *massive parallelism* remain a computational bottleneck, as the factorial growth of interleavings to consider in highly concurrent models can lead to timeouts.

Finally, another aspect worth discussing regards whether the formalization in Table 2 could generate behaviors outside the scope of BPMN. Our encoding is defined inductively: the encoder takes a valid BPMN pattern and translates it into a specific situation calculus theory and ConGolog program fragment. Since the translation is dictated by the syntactic mapping in Table 2 and applying it only to well-formed BPMN patterns, it is structurally impossible for the encoding to synthesize something that does not correspond to the input BPMN model. However, the converse is not true: the expressive power of ConGolog (e.g., allowing for arbitrary non-deterministic iteration and more complex concurrent behaviors) exceeds that of standard BPMN. This distinction defines the boundaries of our semantics, as we use the situation calculus and ConGolog to represent the BPMN logic, rather than to generate process models outside the standard.

10. Conclusion

Within the BPM field, it is widely recognized that a robust formal framework for BPMN modeling is necessary to assist in stating and verifying the process properties. To this end, in this paper, we offer a formal semantics for a significant subset of BPMN using *situation calculus* and ConGolog. Unlike previous works, our encoding specifies the semantics of relevant BPMN fragments concerning the management of data flow, the handling of multi-party collaborations, and exception handlers.

Using well-established KR&R languages has allowed us to establish a formal framework for specifying BPMN models that supports advanced reasoning techniques to perform semantics analysis. For example, by encoding the BPMN logic as a high-level program with ConGolog, we can simulate the process execution before its concrete enactment,

ensuring it behaves as expected. Moreover, we can explore alternative process execution paths by simulating the execution of different sequences of actions, helping identify the best paths for achieving the process goals. Then, by relying on the *legality task* of situation calculus, we can detect and resolve unreachable tasks or deadlocks, and exploiting the *projection task* enables us to investigate if certain data objects will be associated with their expected values given a specific sequence of activities. Moreover, we showed that our framework allows for the performance of more advanced reasoning tasks like conformance checking and property verification, going beyond what is typically available in standard BPMN tools, which are mostly limited to syntactic validation and structural soundness checks.

To bridge the gap from theory to practice, this work's contributions extend beyond the formalization itself. In Section 7, we introduced a software architecture that automates the translation from BPMN models into our executable IndiGolog encoding, making our framework's reasoning capabilities accessible. Furthermore, we provided an extensive experimental evaluation using two datasets, containing BPMN models with complementary features, demonstrating the feasibility of the approach. Indeed, the results show that the automated translation is highly efficient (adding only negligible overhead in the order of ms) and the reasoning tasks execution delivers robust results from both the control- and data-flow perspectives. This confirms that our overall approach can serve as a practical and scalable foundation for the reasoning layer in next-generation ABPMSs.

Future work will tackle the limitations discussed in the previous section to enhance the framework's expressiveness and applicability. First, regarding *coverage*, we plan to extend the basic action theory and high-level program to support *inclusive gateways* and organizational *lanes*, as well as a broader range of event types (e.g., signals). Second, to tackle the *scalability* issues observed in highly concurrent models, we aim to investigate optimization techniques for the reasoner to mitigate state-space explosion in the property verification. Finally, another avenue for research will address the reliance on strict modeling conventions by extending the encoding to support *less structured models* that do not rigorously follow them, enhancing the framework's real-world applicability.

CRediT authorship contribution statement

Angelo Casciani: Writing – review & editing, Writing – original draft, Visualization, Validation, Software, Resources, Methodology, Investigation, Formal analysis, Data curation, Conceptualization. **Simone Agostinelli:** Writing – review & editing, Writing – original draft, Supervision, Conceptualization. **Yves Lespérance:** Writing – review & editing, Writing – original draft, Validation, Supervision, Methodology, Investigation, Formal analysis, Conceptualization. **Andrea Marrella:** Writing – review & editing, Writing – original draft, Validation, Supervision, Project administration, Methodology, Investigation, Formal analysis, Conceptualization. **Sebastian Sardiña:** Writing – review & editing, Writing – original draft, Validation, Supervision, Software, Resources, Methodology, Investigation, Formal analysis, Conceptualization.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgments

This work has been partially supported by the Sapienza project FOND-AIBPM, the Italian National Recovery and Resilience Plan (NRRP) MUR project PE0000013-FAIR, the Natural Sciences and Engineering Research Council of Canada (NSERC), and York University. Angelo Casciani conducted this research while enrolled in the Italian National Doctorate in Artificial Intelligence run by Sapienza University of Rome (CUP B53C23003500006, Mission 4 NRRP Generic Research Scholarship, Component 1). Sebastian Sardina gratefully acknowledges the hospitality and financial support of Sapienza University of Rome during a Visiting Professorship in 2023, when part of this research was carried out.

Appendix A. Supplementary data

Supplementary material related to this article can be found online at https://doi.org/10.1016/j.is.2026.102718.

Data availability

The source code and datasets used for evaluation are available at: https://github.com/angelo-casciani/sitcalc4bpmn. The complete formalization for the case study is reported in the appendix provided as supplementary material.

References

[1] M. Dumas, L.M. Rosa, J. Mendling, A.H. Reijers, Fundamentals of Business Process Management, Springer, 2018.
[2] M. Weske, Business Process Management. Concepts, Languages, Architectures, Springer, 2024.
[3] R.M. Dijkman, M. Dumas, C. Ouyang, Semantics and analysis of business process models in BPMN, Inf. Softw. Technol. 50 (12) (2008) 1281–1294, http://dx.doi.org/10.1016/J.INFSOF.2008.02.006.
[4] F. Corradini, A. Morichetta, C. Muzi, B. Re, F. Tiezzi, Well-structuredness, safeness and soundness: A formal classification of BPMN collaborations, J. Log. Algebraic Methods Program. 119 (2021) 100630, http://dx.doi.org/10.1016/J.JLAMP.2020.100630.
[5] M. Dumas, F. Fournier, L. Limonad, A. Marrella, M. Montali, J.-R. Rehse, R. Accorsi, D. Calvanese, G. De Giacomo, D. Fahland, et al., AI-augmented Business Process Management Systems: A Research Manifesto, ACM Trans. Manag. Inf. Syst. 14 (1) (2023) 1–19.
[6] G. De Giacomo, Y. Lespérance, H.J. Levesque, ConGolog, a concurrent programming language based on the situation calculus, Artificial Intelligence 121 (1) (2000) 109–169, http://dx.doi.org/10.1016/S0004-3702(00)00031-X, URL https://www.sciencedirect.com/science/article/pii/S000437020000031X.
[7] R. Reiter, Knowledge in Action: Logical Foundations for Specifying and Implementing Dynamical Systems, MIT Press, 2001.
[8] J. Carmona, B.F. van Dongen, A. Solti, M. Weidlich, Conformance Checking - Relating Processes and Models, Springer, 2018.
[9] P. Johannesson, E. Perjons, An Introduction to Design Science, Springer, 2014, pp. 1–193, http://dx.doi.org/10.1007/978-3-319-10632-8.
[10] H.J. Levesque, R. Reiter, Y. Lespérance, F. Lin, R.B. Scherl, GOLOG: A logic programming language for dynamic domains, J. Log. Program. 31 (1–3) (1997) 59–83.
[11] G. De Giacomo, Y. Lespérance, H.J. Levesque, S. Sardiña, IndiGolog: A high-level programming language for embedded reasoning agents, in: Multi-Agent Programming, Languages, Tools and Applications, Springer, 2009, pp. 31–72, http://dx.doi.org/10.1007/978-0-387-89299-3_2.
[12] S. Houhou, S. Baarir, P. Poizat, P. Quéinnec, A first-order logic semantics for communication-parametric BPMN collaborations, in: Business Process Management - 17th International Conference, BPM 2019, Vienna, Austria, September 1-6, 2019, Proceedings, in: Lecture Notes in Computer Science, vol. 11675, Springer, 2019, pp. 52–68, http://dx.doi.org/10.1007/978-3-030-26619-6_6.
[13] F. Corradini, F. Fornari, A. Polini, B. Re, F. Tiezzi, A. Vandin, A formal approach for the analysis of BPMN collaboration models, J. Syst. Softw. 180 (2021) 111007, http://dx.doi.org/10.1016/J.JSS.2021.111007.
[14] R. Eid-Sabbagh, M. Hewelt, M. Weske, A tool for business process architecture analysis, in: Service-Oriented Computing - 11th International Conference, ICSOC 2013, Berlin, Germany, December 2-5, 2013, Proceedings, in: Lecture Notes in Computer Science, vol. 8274, Springer, 2013, pp. 688–691, http://dx.doi.org/10.1007/978-3-642-45005-1_61.
[15] A. Meyer, L. Pufahl, D. Fahland, M. Weske, Modeling and enacting complex data dependencies in business processes, in: Business Process Management - 11th International Conference, BPM 2013, Beijing, China, August 26-30, 2013. Proceedings, in: Lecture Notes in Computer Science, vol. 8094, Springer, 2013, pp. 171–186, http://dx.doi.org/10.1007/978-3-642-40176-3_14, URL https://doi.org/10.1007/978-3-642-40176-3_14.

[16] F. Kossak, C. Illibauer, V. Geist, J. Kubovy, C. Natschläger, T. Ziebermayr, T. Kopetzky, B. Freudenthaler, K. Schewe, A Rigorous Semantics for BPMN 2.0 Process Diagrams, Springer, 2014, http://dx.doi.org/10.1007/978-3-319-09931-6.

[17] A. Kheldoun, K. Barkaoui, M. Ioualalen, Formal verification of complex business processes based on high-level Petri nets, Inf. Sci. 385 (2017) 39–54, http://dx.doi.org/10.1016/J.INS.2016.12.044.

[18] F. Corradini, C. Muzi, B. Re, L. Rossi, F. Tiezzi, Formalising and animating multiple instances in BPMN collaborations, Inf. Syst. 103 (2022) 101459, http://dx.doi.org/10.1016/J.IS.2019.101459.

[19] S. Steinau, A. Marrella, K. Andrews, F. Leotta, M. Mecella, M. Reichert, DALEC: A framework for the systematic evaluation of data-centric approaches to process management software, Softw. Syst. Model. 18 (2019) 2679–2716.

[20] R. Lucchi, M. Mazzara, A pi-calculus based semantics for WS-BPEL, J. Log. Algebraic Methods Program. 70 (1) (2007) 96–118, http://dx.doi.org/10.1016/J.JLAP.2006.05.007.

[21] D. Sangiorgi, Pi-calculus, in: Encyclopedia of Parallel Computing, Springer US, Boston, MA, 2011, pp. 1554–1562, http://dx.doi.org/10.1007/978-0-387-09766-4_202.

[22] G.D. Giacomo, M. Dumas, F.M. Maggi, M. Montali, Declarative process modeling in BPMN, in: Advanced Information Systems Engineering - 27th International Conference, CAiSE 2015, Stockholm, Sweden, June 8-12, 2015, Proceedings, in: Lecture Notes in Computer Science, vol. 9097, Springer, 2015, pp. 84–100, http://dx.doi.org/10.1007/978-3-319-19069-3_6.

[23] L. Barbaro, G. Varricchione, M. Montali, C.D. Ciccio, From sound workflow nets to LTL_f declarative specifications by casting three spells, in: Business Process Management Forum - BPM 2025 Forum, Seville, Spain, August 31 - September 5, 2025, Proceedings, in: Lecture Notes in Business Information Processing, vol. 564, Springer, 2025, pp. 3–22, http://dx.doi.org/10.1007/978-3-032-02929-4_1.

[24] L. Giordano, D.T. Dupré, ASP and ontologies for reasoning on business processes, in: Discussion and Doctoral Consortium Papers of AI*IA 2019 - 18th International Conference of the Italian Association for Artificial Intelligence, Rende, Italy, November 19-22, 2019, in: CEUR Workshop Proceedings, vol. 2495, CEUR-WS.org, 2019, pp. 45–52, URL https://ceur-ws.org/Vol-2495/paper6.pdf.

[25] M. Weidlich, J.M.E.M. van der Werf, On profiles and footprints - relational semantics for Petri nets, in: Application and Theory of Petri Nets - 33rd International Conference, PETRI NETS 2012, Hamburg, Germany, June 25-29, 2012. Proceedings, in: Lecture Notes in Computer Science, vol. 7347, Springer, 2012, pp. 148–167, http://dx.doi.org/10.1007/978-3-642-31131-4_9.

[26] V. Skydanienko, C.D. Francescomarino, C. Ghidini, F.M. Maggi, A tool for generating event logs from multi-perspective declare models, in: Proceedings of the Dissertation Award, Demonstration, and Industrial Track At BPM 2018 Co-Located with 16th International Conference on Business Process Management (BPM 2018), Sydney, Australia, September 9-14, 2018, in: CEUR Workshop Proceedings, vol. 2196, CEUR-WS.org, 2018, pp. 111–115, URL https://ceur-ws.org/Vol-2196/BPM_2018_paper_23.pdf.

[27] A. Marrella, M. Mecella, S. Sardina, Intelligent process adaptation in the smartpm system, ACM Trans. Intell. Syst. Technol. (TIST) 8 (2) (2016) 1–43.

[28] S. Sohrabi, N. Prokoshyna, S.A. McIlraith, Web service composition via the customization of Golog programs with user preferences, in: Conceptual Modeling: Foundations and Applications, in: LNCS, vol. 5600, Springer, 2009, pp. 319–334.

[29] J. Claßen, M. Liebenberg, G. Lakemeyer, B. Zarrieß, Exploring the boundaries of decidable verification of non-terminating Golog programs, in: Proceedings of the Twenty-Eighth AAAI Conference, AAAI Press, 2014, pp. 1012–1019.

[30] G. De Giacomo, Y. Lespérance, F. Patrizi, S. Sardiña, Verifying ConGolog programs on bounded situation calculus theories, in: Proceedings of the Thirtieth AAAI Conference, AAAI Press, 2016, pp. 950–956.

[31] J. Li, Y. Liu, Automatic verification of liveness properties in the situation calculus, in: AAAI'20, AAAI Press, 2020, pp. 2886–2892, http://dx.doi.org/10.1609/AAAI.V34I03.5679.

[32] N. Klievtsova, J.-V. Benzin, T. Kampik, J. Mangler, S. Rinderle-Ma, Conversational process modelling: State of the art, applications, and implications in practice, in: Business Process Management Forum, Springer Nature Switzerland, Cham, 2023, pp. 319–336.

[33] J. Mangler, N. Klievtsova, Textual Process Descriptions and Corresponding BPMN Models, Zenodo, 2023, Version 1.0. URL https://doi.org/10.5281/zenodo.7783492.

[34] J. Benesty, J. Chen, Y. Huang, I. Cohen, Pearson correlation coefficient, in: Noise Reduction in Speech Processing, Springer, 2009, pp. 1–4.

[35] R. Reiter, Natural actions, concurrency and continuous time in the situation calculus, in: L.C. Aiello, J. Doyle, S.C. Shapiro (Eds.), Proceedings of the Fifth International Conference on Principles of Knowledge Representation and Reasoning (KR'96), Cambridge, Massachusetts, USA, November 5-8, 1996, Morgan Kaufmann, 1996, pp. 2–13.