Protecting Sensitive Data Against Deductions in Business Processes: Formal Semantics and Model-Checking

Sabine Frittella, Vincent Hugot*, and Amine Rifi**

INSA CVL & LIFO, UR 4022 – U. Orléans
firstname.name@insa-cvl.fr

Abstract. Achieving confidentiality objectives in business processes handling sensitive or private data may require taking into account the agents' deductive capabilities. Cross-referencing data to which an agent has legitimate access with basic rules of business logic may reveal otherwise denied information: this is a (deductive) *leak*. We create tooling to mechanically verify workflow data safety against deductive leaks.

As a proof-of-concept, this paper introduces DPWN: a fully formal, model-checkable, workflow language that is essentially a subset of BPMN equipped with "epistemic annotations". Within the deduction framework of propositional logic, it enables us to statically check workflows against (among other things) data security properties of the general form "no agent may deduce the value of data to which it is not given explicit access". Its semantics is expressed in TLA$^+$, and our verification software uses the TLC model-checker with Z3 as a SAT-solver deductive backend.

Keywords: BPMN · Formal Semantics · Confidentiality · Verification · Model-Checking · TLA$^+$ · SAT Solvers

1 Introduction

Protecting sensitive data in business processes is crucial to both internal security objectives and regulatory compliance. For instance, the General Data Protection Regulation (GDPR, [6]) enforces standards of *confidentiality* (no unauthorised processing), *transparency* (accurate information about data processing), and *data minimisation* (minimal data collection wrt. its purpose). Regarding confidentiality, we work under the viewpoint that data must be protected both against *explicit* access and against cross-referencing and deductions, which is what we call *leaks* in that context.

Business processes are often modelled using graphical workflow languages, which can be executed through various process engines, studied, verified, etc. Our objective is to build tooling capable of enforcing the notion of confidentiality against deductions through static, automatic verification of a workflow. This

* Corresponding author.
** PEPR iPoP (interdisciplinary Project on Privacy) - ANR-22-PECY-0002

requires both **(1)** formal semantics for control-flow and data, and **(2)** a formal model for agent knowledge and deductive capabilities.

Among graphical workflow-based business process modelling languages, BPMN [16] is an ISO/IEC standard and widely used, in particular in the domain of regulatory compliance [14]. Thus, we choose it as a starting point for our proof-of-concept. Specifically, we start with the core subset of constructs shown in [12] — studying a database of 825 real-world industrial processes — to be the most relevant in practice: start and end events, flows, tasks, and parallel and exclusive gateways. Those are all fairly intuitive concepts which we shall redefine, thus while familiarity with BPMN is helpful, it is not a prerequisite for this paper, and the methodology we present could be applied, with little change, using other workflow languages as starting points, as most workflow languages share the same patterns [20].

Related works: Knowledge modelling is out of BPMN's scope, though there are proposed extensions for sensitive business processes, e.g. [10], for control-flow and data. While the official BPMN specification is only *semi*-formal, there is a wide body of works formalising and developing verification tooling for various subsets of BPMN, using numerous formalisms and tools (Labelled Transition Systems, diverse extensions to Petri Nets, Temporal Logics like CTL & LTL, rewriting logic & Maude, ConGolog, etc). See [4, Table 1] and [18, Sec. 7] for rather comprehensive comparisons of those works.

To navigate all this, let us remember our objective: we need a formalism that is model-checkable under agent reasoning. That implies it must be data-aware, as deductive reasoning requires access to data values (or at least an abstraction thereof). The majority of works abstract away from data entirely and handle control-flow nondeterministically, e.g. [18]. Of the works that do account for data, most focus on enactment, execution, and simulation, as opposed to static verification e.g. [15]. The underlying reason for that is the state-space explosion problem, making data-aware model-checking inherently costly; there are few works within that scope. PE-BPMN and Pleak [19] analyse data flows for privacy leaks using a fixed set of privacy stereotypes and associated rules, rather than general reasoning. [11] translates a subset of BPMN into high-level Petri Net (RECATNets), and from there into rewriting logic, and performs static verification of LTL properties in Maude. It has a notion of stateful data objects. There is no attempt to handle deductions in that work, though we can imagine an extension where agents and deductions are modelled in rewriting logic. The approach would be complex, both in the sense of the various layers of formalisms involved, and in terms of controlling the overall algorithmic complexity and even decidability. A more recent work, [4], formalises a data-aware BPMN subset into ConGolog, a logical concurrent processes language based on the Situation Calculus. This is an application of Knowledge Representation and Reasoning languages to the formalization of BPMN semantics, similar in purpose to data-aware model-checking, not an attempt to account for agents' deductive capabilities.

In this paper, we chose to follow an approach similar to [18], who used the temporal logic of actions (TLA$^+$, [13]) to formalise the control-flow semantics. We will encode data, knowledge, and reasoning in TLA$^+$ as well. The use of (restricted) TLA$^+$ makes the formal semantics executable and model-checkable; we will use TLC [21], and leverage TLC features enabling us to delegate the difficult computations tied to deductions to specialised solvers.

While the normal modal logic S5 [7] and description logics [2] were plausible candidates for the deduction framework, we settled on classical propositional logic (PL), which is sufficiently expressive for complex business logic. Like in classical epistemic logic, our agents are "logically omniscient", i.e. they have unlimited cognitive or computational resources to perform deductions. This corresponds to the notion of (first-order) *knowledge* in S5: intuitively, $\mathbf{K}_A\varphi$ ("agent A knows that φ") iff φ holds in all worlds compatible with A's information. This is expensive to compute: deductive leak detection is coNP-complete. Fortunately, the choice of PL lets us directly leverage the power of SAT solvers, which routinely handle extremely large instances. Higher-level languages can then be built on top of PL for the end user.

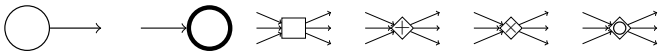
As a proof-of-concept, this paper introduces DPWN (Data [Deduction] Protection Workflow Notation), essentially a formally-specified subset of BPMN, with minor semantic differences, and additional annotations – which we refer to as the *epistemic layer* – dealing with the various *agents* (entities processing data) involved in the workflow, their explicit access rights, and the background knowledge under which they can reason. We provide a prototype model-checking tool for it, based on TLC [21] and Z3 [5].

We apply it to a practical example built on the case study of [1], showing how our tool can enforce confidentiality under deductions, while also achieving transparency and data minimisation objectives. All code used for this paper is open-source and archived on [(a)].

Structure: Sec. 2[p3] introduces DPWN informally and through a synthetic example. Sec. 3[p7] defines the formal semantics of DPWN. Sec. 4[p13] shows that leak detection is coNP-complete and discusses the SAT-Solver–based implementation, Sec. 5[p14] presents an example of application to data minimisation.

2 DPWN Informal Overview and Synthetic Example

In a nutshell, DPWN borrows the following control flow structures from BPMN:



In order: start and end events, tasks, parallel gateway (which, in DPWN, we display as $\|$ instead of $+$), exclusive gateway, and inclusive gateway. The arrows are called *(sequence) flows*. A process starts by executing start event nodes, then execution follows the flows, and is split and merged at gateways; e.g. $\times$ choses

[(a)] https://doi.org/10.6084/m9.figshare.31361536.

exactly one flow, while ‖ executes all flows concurrently. DPWN gateways may be mixed (i.e. not restricted to *split* and *merge*) and unbalanced. The semantics will be formalised through a notion of *token*, similar to Petri nets.

In the example of Fig. 1[p4], and ignoring all annotations except node numbers, start event 1 executes and generates a token on its outgoing flow, then the × gate 2 chooses either the flow to 3 or 4; a task executes, then 5 chooses a flow to either 6, ending the process, or 2, entering a loop. Obviously those choices, and the potential existence of deadlocks, infinite loops, etc, depend on the data.

The annotations (except "Task 1" and "Description 1", which are free text) follow a formal language, and form part of the *epistemic layer* which DPWN introduces on top of (and interacting with) control flow. These annotations deal with **data**[(b)], **agents**, their **access** to or **knowledge** of the data values, and the **actions** that alter data values and access rights. Those should not be mapped directly onto BPMN concepts. A DPWN agent does not generally correspond to a BPMN Lane or Participant. Agents are entities whose knowledge is affected by process execution. A given DPWN task may affect any number of agents *depending on data values*; this dynamic notion cannot be rendered into a static notation. Likewise, DPWN data is a finer-grained notion than Data Objects, and the relationships between data are governed by logical statements which cannot generally be reduced to Data Associations.

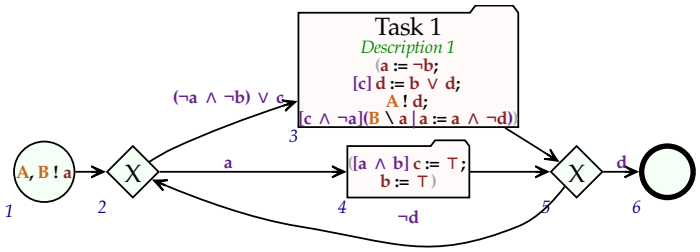


Fig. 1: Synthetic DPWN workflow, graphically as shown by the tool. Only node-level annotations are shown. Workflow-level annotations are discussed in the text.

Fig. 1[p4] illustrates most of the annotations and primitives of the epistemic layer. Tasks and Events are decorated with *actions* (which we will denote by α in the semantics). "`A,B!a`" is an action **granting** to *agents* `A` and `B`, explicit access to *resource* or *data* `a`. Such access can be later **retracted**, e.g. "`B\a`", which appears at the end of node 3, retracts access to `a` for `B`. If "`a`" stands for a resource like "the office stapler", then this is a way to formalise a classical dynamic access

(b) . . . abstracted as Boolean variables; more complex value types and structures can be encoded using several Booleans, e.g. "*age* ⩾ *18*" and "*age* ⩾ *60*" may suffice to encode a numerical value *age*, if those are the tests appearing in the workflow. Such methods are classical in *predicate abstraction*, a technique of abstract interpretation in which the abstract domain is characterised by a set of predicates over program variables, cf. e.g. [9,3]. Automating this in our tool is ongoing work.

control policy. Our focus is on the case when "a" represents data, with a *value*, which may change in the course of the process execution and, crucially, which may be intertwined with others through logical relationships.

At any point during the execution, a variable holds a specific value ($\top$ or $\bot$), which can be **altered**: e.g. in node 3, "a:=¬b" means that, during the execution of the task, the value of a changes to ¬b. The right-hand side may be any propositional formula over the workflow variables (denoted γ in the semantics). The same node illustrates that actions can be **chained** with ";", with the obvious meaning of sequential execution. Node 3 also illustrates the last two action syntaxes in the example: In "[c]d:=b ∨ d", the c is a **guard**, meaning the associated action only occurs if c is true. A guard could be any formula γ. The last action, $[c \wedge \neg a](B \setminus a \mid a := a \wedge \neg d)$, is of the more general form "if..then..else", with | serving as separator. If $c \wedge \neg a$ is true, then B loses access to a, otherwise a is altered.

Guards may also appear on (some) sequence flows, typically outgoing gateway flows. Only the flows whose guards are true may be taken (nondeterministically; there is no notion of "order of evaluation" or "default" in DPWN).

That covers most annotations at the level of flow objects. Other important epistemic annotations apply to the whole workflow. Let's come back to the meaning of "granting access", and logical relationships between variables.

"A,B!a" means that the agents *know the value of a*, and will know the current value of a until that access is retracted. Furthermore, they can reason (conduct deductions) based on that value, logical necessity, and known business logic.

If a stands for "*age* $\geqslant$ *60*", and b for "*age* $\geqslant$ *18*", then they are related by $a \Rightarrow b$. Any agent with access to a at any point when a is true will also know the value of b (true). If that agent does not have explicit access to b, we call that a *leak*. This is an example of tautological relationship; others are an expression of business logic: e.g. *senior_rebate* $\Leftrightarrow$ (*age* $\geqslant 60 \wedge$ *resident*). Real-world business logic routinely involves a large number of variables and complex clauses, cf. [1, Sec. 5]. Our agents thus reason, i.e. conduct logical deductions, over an arbitrary formula of propositional logic φ, which we call the **structural invariant**, considered common knowledge. Our synthetic example has $\varphi = (d \Rightarrow c \wedge b) \wedge (b \Rightarrow s)$. A variable x *leaks* to an agent if all models of φ consistent with the agent's explicit knowledge assign the same value to x. In other words, when there is only one value x could take while being consistent with φ and the known values. This corresponds to the classical notion of knowledge in epistemic logic.

In practice, φ is generally structured as the conjunction of some tautological consistency rules, and business rules (e.g. [1, Tab. 1]). In future works we will build a higher-level user-facing abstraction that generates the consistency rules automatically, so the user can focus on business logic.

There remains to set the initial values of the variables. Is "*age* $\geqslant$ *60*"? Is "*age* $\geqslant$ *18*"? Presumably, we want our workflow to work for *all* customers, and we will leave all possibilities open, so long as they're consistent with φ. Other variables

we may wish to initialise to specific values. This is done with an arbitrary formula $\boldsymbol{\varphi_{\mathbf{init}}}$, called the **initial formula**. Our initial valuations are exactly the models of $\varphi_{\text{init}} \wedge \varphi$. Our synthetic example has $\varphi_{\text{init}} = \neg \mathsf{b} \wedge \neg \mathsf{d} \wedge \mathsf{s}$, which creates four initial states: a and c may take any values.

With this, the model is complete; the example has 33 distinct reachable states, defined by the current values of all variables and the knowledge (access rights) of all agents. From this, we can compute the data leaks (deductions) for each state: cf. Fig. 2[p6].

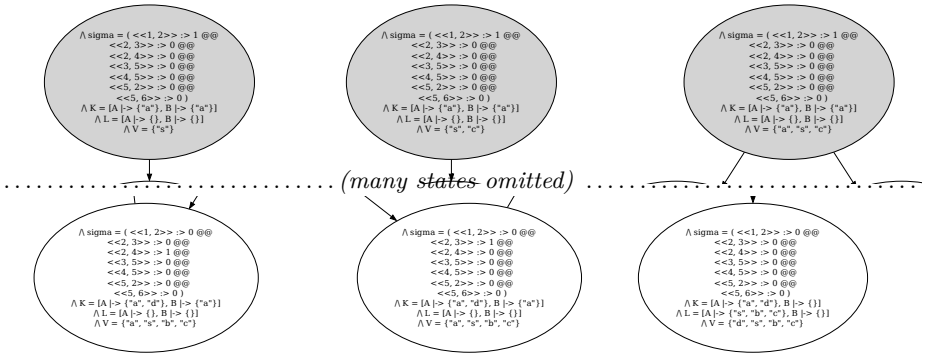


Fig. 2: Extract from the behaviour graph for Fig. 1[p4], generated by the TLC model-checker. The state **VARIABLES** are defined at the beginning of Sec. 3.2[p9]. The full graph has 33 states (4 initial).

The last annotations are the invariants and properties to be verified. They are directly expressed in TLA$^+$ against our formal semantics, though we provide a few definitions to make writing them easier. For instance, our example does not satisfy the invariant `NO_LEAKS`, so *some* variables leak to *some* agents at some point. It does satisfy `SAFE_AGENT("B")`, meaning that nothing ever leaks to `B`, and `SAFE_SET({"a","d"})`, meaning that no agent ever deduces these variables. Every other variable leaks to `A`, though. It also satisfies invariants `TOKENS` $\leqslant$ `1`, meaning there is at most one token active at any given time, and `DEADLOCK_FREE`, which is self-explanatory. It does not satisfy the property `TERMINATES`, because infinite loops are possible.

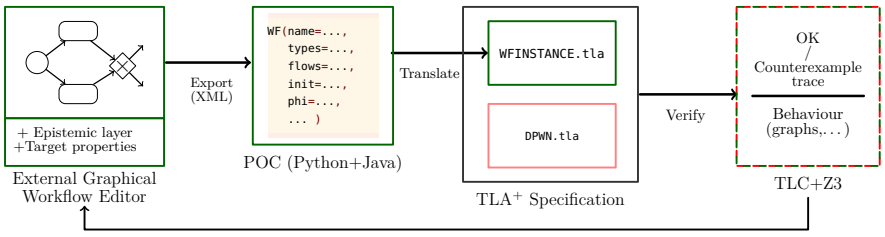


Fig. 3: Verification process (simplified)

Our prototype tool is mostly written in Python, and translates the above information into a TLA$^+$ description of the workflow instance (`WFINSTANCE.tla`), which is then checked by TLC against the language semantics (mostly in `DPWN.tla`). The computation of leaks is done through a TLC override (in Java) interfacing with Z3, which serves as a SAT solver backend. See Fig. 3[p6]. Our examples are directly coded in Python, but very rudimentary `BPMN/20100524` XML import is supported. Tighter integration with existing workflow editors is a longer term perspective.

3 Formal Semantics for DPWN

We now define the main points of the formal semantics of DPWN, written in TLA$^+$. For readers unfamiliar with it, the **Temporal Logic of Actions** (TLA$^+$, [13]) is a formal specification language. It is, in a nutshell, a variant of Linear Temporal Logic (LTL, [17]) with states and transitions specified in untyped Zermelo–Fraenkel set theory, with unambiguous syntax (both in ASCII and typeset versions) and an ecosystem of verification tools (for subsets of TLA$^+$). We use the explicit model-checker TLC [21].

The behaviour of a dynamic system is described mainly by a predicate on *transitions*, i.e. couples of successive states $\langle s, s' \rangle$. A state is a valuation of the system's variables, and the predicate is an arbitrary formula of first order logic relating the old values to the new ones (e.g. $x' = x + 1$ for "x is incremented"). TLA$^+$ is therefore very easy to read without prior knowledge, apart from a few points of syntax and foundations (e.g. in TLA$^+$, functions are a primitive notion, not defined as a set of couples, and tuples are functions of domain `1..n`).

All the notations we use are the standard way to typeset TLA$^+$, except for the following: `UNION S` is typeset $\bigcup$`S`, `SUBSET S` is typeset $\wp$ `S`, `EXCEPT` is typeset $\|$, `:>` is typeset $\mapsto$, `@@` is typeset $\blacktriangleright$. As a reminder, $\wp$ `S` is the powerset of `S`; `[f ‖ ![x]=@+1]`, for instance, is a function $\hat{f}$ identical to `f` except that $\hat{f}[x] = f[x] + 1$; `x` $\mapsto$ `y` is a singleton function; and `f` $\blacktriangleright$ `g` is the union of functions `f` and `g`, but `f` overrides `g` if their domains are not disjoint. Beyond those reminders, we refer the reader to [13].

3.1 Control Flow

The workflow instance (in the generated `WFINSTANCE.tla` file) is a graph defined by the edge relation `fl` (for *flows*) and the function `type` (giving the type of flow object corresponding to each node, e.g. a start event, task, etc). Other useful objects are computed from those two. We have the following type invariant:

```
well_typed ≜ ...
    ∧ fl ∈ ℘ (nodes × nodes) \* flows: instance input
    ∧ type ∈ [ nodes → NodeTypes ] \* instance input
    ∧ in ∈ [ nodes → ℘ fl ] \* incoming: derived from fl
    ∧ out ∈ [ nodes → ℘ fl ] \* outgoing: derived from fl
```

Ellipses (i.e. "...") in the specification are not part of the TLA$^+$ syntax, but indicate that we leave stuff out, in this case hiding clauses relevant to the epistemic layer while we focus on control flow. Our synthetic example is represented as:

```
fl ≜ { ⟨1,2⟩, ⟨2,3⟩,              type ≜  1 ↦ EventStart
       ⟨2,4⟩, ⟨3,5⟩,                    ▶ 2 ↦ GateXOR
       ⟨4,5⟩, ⟨5,2⟩, ⟨5,6⟩ }            ▶ 3 ↦ Task ...
```

The control flow semantics follow the usual "token game" of [16,18], simplified for our needs. A state of the process is a marking $\boldsymbol{\sigma}$ of its edges (i.e. flows) with the number of tokens currently on that edge.

```
VARIABLES σ,...
TypeInvariant ≜ σ ∈ [ fl → ℕ ] ...
```

We can now give the semantics of our various flow objects. A flow `f` has a source `src` and a destination `dst`.

Event Start  **:** At the initial state, some Event Start node generates a token on each of its outgoing edges.

```
Init ≜  ∃ n ∈ oftype(EventStart):
    σ = [ f ∈ fl ↦ IF src(f)=n THEN 1 ELSE 0 ] ...
```

In BPMN, all start events would hold a token initially. However, in DPWN, start events may have actions, whose combined effects would depend upon an arbitrary "order of execution". There is no obstacle to formalising such a behaviour, but we find it unintuitive. Thus, we made the choice to deviate from BPMN's semantics on that point. In DPWN, only one, nondeterministically chosen, start event is triggered initially.

All other flow objects are dealt with in the action predicate `Next`, which specifies valid state transitions of the process. The control flow state changes when some node `n` — nondeterministically chosen — *executes*, which is specified in the `exec(n)` operator. The `dexec(n)` operator adds the data/epistemic layer to that; we shall deal with it later.

```
Next ≜ ∃ n ∈ nodes: type[n] ≠ EventStart ∧ dexec(n)

dexec(n) ≜ exec(n) ∧ ... (* exec + data/epistemic layer *)

exec(n) ≜ (* dispatch on node type *)
  CASE type[n] = Task        → exec_Task(n)
    □ type[n] = EventEnd     → exec_EventEnd(n)  □ ...
```

Thus the semantics of each flow object is an action predicate given as an operator with the naming convention "`exec_SomeType(n)`".

Event End  **:** An End Event node executes when it has at least one *marked* incoming flow (that is to say, an incoming flow with at least one token), and consumes the token.

```
in•[n ∈ nodes]    ≜ { i ∈ in[n]  : σ[i] ⩾ 1 }
exec_EventEnd(n) ≜ ∃ i ∈ in•[n] : σ' = [σ ‖ ![i] = @ - 1]
```

Tasks  **:** A Task node executes when it has at least one marked incoming flow; it consumes one of its token, and produces a token on each outgoing flow.

```
exec_Task(n) ≜ ∃ i ∈ in•[n] :
    σ' =  i ↦ σ[i]-1  ▶  [o ∈ out[n] ↦ σ[o] + 1]  ▶  σ
```

Gateway AND  **:** An AND / Parallel (||) gateway executes when all its incoming flows are marked. It consumes one token from each incoming flow, and produces one token on each outgoing flow.

```
exec_Gate_AND(n) ≜ in[n] = in•[n] ∧
    σ' =  [i ∈ in[n]  ↦ σ[i]-1]  ▶  [o ∈ out[n] ↦ σ[o]+1]  ▶  σ
```

Gateway XOR  **:** A XOR gateway supports guards on its outgoing flows. It executes when it has at least one marked incoming flow, and at least one *available* outgoing flow (i.e. , one outgoing flow whose guard evaluates to true). It consumes an incoming token and produces one token on that outgoing flow.

```
exec_Gate_XOR(n) ≜ (* out^γ = ''outgoing flows with true Guard'' *)
    ∃ i ∈ in•[n], o ∈ out^γ[n] : σ' = [σ ‖ ![i]= @ - 1, ![o]= @ + 1]
```

Note: All our flows have guards; if not given explicitly, they default to $\top$. The $\mathtt{out}^{\gamma}$ function is defined in the next section, dealing with data.

Gateway OR  **:** The BPMN inclusive OR gateway has complicated, nonlocal semantics. Discussing that is beyond the scope of this paper; see [18] for more details. Our tool currently supports OR in the loopfree case. There is no fundamental obstacle to fully supporting OR Gateways, or Complex Gateways, so long as their semantics can be defined purely in terms of σ and $\mathtt{out}^{\gamma}$. In practice, OR is rarely used: 5% of workflows in the study [12].

3.2 Data and Epistemic Layer

The set $\mathbb{V}$ of atomic propositional variables is extracted from the epistemic annotations of the workflow instance. Likewise for the set $\mathbb{A}$ of agents that appear in the workflow. In our synthetic example, we have:

```
𝕍 ≜ {"a","b","c","d","s"}      𝔸 ≜ {"A","B"}
```

Thus, the system's state is fully described by the following variables:

```
VARIABLES σ, V, K, L
TypeInvariant ≜
    ∧ σ ∈ [fl → ℕ]  (* control-flow state, as defined previously *)
    ∧ V ⊆ 𝕍         (* state of the world: valuation for 𝕍 *)
    ∧ K ∈ [𝔸 → ℘ 𝕍] (* legitimate knowledge/access *)
    ∧ L ∈ [𝔸 → ℘ 𝕍] (* leaked knowledge: DERIVED from V,K *)
```

`V` is the set of variables of $\mathbb{V}$ that are true; variables of $\mathbb{V} \setminus$ `V` are therefore false.

`K[A]` is the (legitimate) *knowledge* of agent `A`, i.e. the set of variables to which `A` has been granted explicit access. Thus `A` knows the truth value of each of those variables, even if they later change, so long as `A` retains that access.

`L[A]` is the illegitimate/*leaked* knowledge of agent `A`, i.e. the set of variables, *not* in `K[A]`, whose truth values `A` can deduce. It is computed from `V`, `K`, and the structural invariant φ — we will show how shortly.

The structural invariant φ and initial formula φ_{init} are encoded as operators taking a valuation as argument and returning a Boolean. In our example:

```
φ_init(V) ≜ LET b ≜ "b" ∈ V  d ≜ "d" ∈ V  s ≜ "s" ∈ V IN
    ¬b ∧ ¬d ∧ s

φ(V) ≜ LET b ≜ "b" ∈ V  c ≜ "c" ∈ V  d ≜ "d" ∈ V  s ≜ "s" ∈ V IN
    (d ⇒ (c ∧ b)) ∧ (b ⇒ s)
```

`φ(V)` is a invariant, and our tool checks it. With this, we can now specify leaks. A variable *leaks* if it is true (resp. false) in all models of φ that extend the partial valuation known by an agent. In the following operators, K_A stands for the legitimate knowledge `K[A]` of some agent `A` (set of known variables) and `V` for a valuation, as above (set of true variables).

```
pvals(V,K_A) ≜ \* possible valuations, given knowledge K_A
    { (K_A ∩ V) ∪ E : E ∈ ℘ (𝕍 \ K_A) }
vvals(V,K_A) ≜ \* valid valuations of φ, given knowledge K_A
    { v ∈ pvals(V,K_A) : φ(v) }
leak(V,K_A) ≜ \* set of leaked variables, given knowledge K_A
    LET W ≜ vvals(V,K_A) IN { x ∈ (𝕍 \ K_A) : x ∈ ⋂W ∨ x ∉ ⋃W }
```

In practice, `leak(V,K_A)` is implemented via a SAT solver; cf. Sec. 4[p13].

Guards are encoded as an operator γ associating a formula – encoded in the same style as φ – to each flow. In our example:

```
γ(V,K, f) ≜  CASE ...
    □ f = ⟨2,3⟩ → LET a ≜ "a" ∈ V  b ≜ "b" ∈ V  c ≜ "c" ∈ V
                  IN (¬a ∧ ¬b) ∨ c
    □ f = ⟨2,4⟩ → LET a ≜ "a" ∈ V IN a      □ OTHER → ⊤
```

The parameter `K` is unused for now, but included in case we want to extend the scope of guards to knowledge/access information. We can now define $\texttt{out}^\gamma$:

```
out^γ[n ∈ nodes] ≜ { f ∈ out[n] : γ(V',K', f) }
```

We now define the grammar of actions α. Let N denote an identifier, such as `"Alice"` or `"x"`, and N^+ a comma-separated list of N, seen as a set of identifiers.

A γ is any formula of propositional logic, with the obvious syntax, as seen in previous examples. A feature *not* seen thus far is that atoms may also be *TLA*$^+$ *injections*, with the syntax `<§` $\mathsf{T}_{\mathbb{B}}$ `§>`, where $\mathsf{T}_{\mathbb{B}}$ is an arbitrary TLA$^+$ expression evaluating to a Boolean in the context of the generated specification.

An action α is defined by the following grammar, where all symbols are terminals except the above and the ellipsis "`...`", which has its usual intuitive meaning:

α	::=	$\mathsf{N}^+ \,!\, \mathsf{N}^+$	e.g. `"A,B ! c,d"`: **grant** agents `A,B` access to `c,d`
		$\mathsf{N}^+ \setminus \mathsf{N}^+$	e.g. `"A,B \ c,d"`: **retract** `A` and `B`'s access to `c,d`
		$\mathsf{N} := \gamma$	e.g. `"a := b ∧ c"`: **alter** value of `a`
		$[\gamma]\alpha$	e.g. `"[b ∧ c] a := b"`: **guard** an action
		$[\gamma](\alpha \mid \alpha)$	e.g. `"[b ∧ c](a := b | b := a)"`: **if-then-else**
		$\alpha; \ldots; \alpha$	e.g. `"a := b ; b := c"`: **chain** of actions
		(α)	same as α, to override precedence rules

There is also the syntactic sugar $\langle x_1, \ldots, x_n \rangle$:= `<§` $\mathsf{T}_{\langle \mathbb{B},\ldots \rangle}$ `§>`, to break down an injected TLA$^+$ tuple of Boolean-evaluating expressions. It is translated into $x_1 :=$ `<§` $\mathsf{T}_{\langle \mathbb{B},\ldots \rangle}[1]$ `§>`; $\ldots$; $x_n :=$ `<§` $\mathsf{T}_{\langle \mathbb{B},\ldots \rangle}[n]$ `§>`. There are many possible extensions to the grammar above, but designing a higher-level user-facing language is the topic of ongoing work, and outside the scope of this paper. We only introduce this construct to make the practical example of Sec. 5[p14] more legible.

As for precedence rules: guards bind more tightly than chains. The parser also matches the longest chains possible, which does not matter given its (associative) semantics, but makes the abstract syntax tree (AST) flatter, and the resulting TLA$^+$ a little easier to read by a human being.

Actions deterministically affect the valuations of variables $\mathbb{V}$ and the legitimate knowledge of agents.[(c)] Thus we define the semantics of an action α as a function $[\![\alpha]\!](V, K) = V', K'$.

$$
\begin{aligned}
[\![\mathsf{N}_1^+ \,!\, \mathsf{N}_2^+]\!](V,K) &= V, \left[A \in \mathsf{N}_1^+ \mapsto K[A] \cup \mathsf{N}_2^+\right] \blacktriangleright K \\
[\![\mathsf{N}_1^+ \setminus \mathsf{N}_2^+]\!](V,K) &= V, \left[A \in \mathsf{N}_1^+ \mapsto K[A] \setminus \mathsf{N}_2^+\right] \blacktriangleright K \\
[\![\mathsf{N} := \gamma]\!](V,K) &= (\textbf{IF } \gamma \textbf{ THEN } V \cup \{\mathsf{N}\} \textbf{ ELSE } V \setminus \{\mathsf{N}\}), K \\
[\![[\gamma]\alpha]\!](V,K) &= \textbf{IF } \gamma \textbf{ THEN } [\![\alpha]\!](V,K) \textbf{ ELSE } (V,K) \\
[\![[\gamma](\alpha_1 \mid \alpha_2)]\!](V,K) &= \textbf{IF } \gamma \textbf{ THEN } [\![\alpha_1]\!](V,K) \textbf{ ELSE } [\![\alpha_2]\!](V,K)
\end{aligned}
$$

[(c)] We may allow nondeterministic actions in future versions; there is no strong reason not to, and there are use-cases.

$$[\![\alpha_1;\ldots;\alpha_n]\!](V,K) \quad = \quad [\![\alpha_n]\!]\big(\ldots[\![\alpha_1]\!](V,K)\big)$$

There remains to translate those semantics into pure TLA$^+$. We let ρ denote a *position* in the AST, e.g. the empty word ε for the root, 12 for the second child of the first child, etc. We denote by $(\!|\alpha|\!)_\rho$ the TLA$^+$ translation of the semantics of α at position ρ, and build it such that, intuitively,

$$(\!|\alpha|\!)_\rho \quad = \quad a_\rho(V_\rho,K_\rho) \triangleq [\![\alpha]\!](V_\rho,K_\rho)\ . \tag{1}$$

That is to say, $(\!|\alpha|\!)_\rho$ is the TLA$^+$ *definition* of an operator named a_ρ with the effect of $[\![\alpha]\!]$. The full definition of $(\!|\alpha|\!)_\rho$ thus looks almost identical to that of $[\![\alpha]\!]$, which is already very close to pure TLA$^+$ syntax, with the addition of some trivial "boilerplate": mostly **LET** .. **IN** clauses to bind the inductive calls, and explicit tuple construction/deconstruction. For instance, we have

$$\begin{aligned}(\!|[\gamma](\alpha_1 \mid \alpha_2)|\!)_\rho \quad = \quad & a_\rho(V_\rho,K_\rho) \triangleq \\ & \textbf{LET } (\!|\alpha_1|\!)_{\rho 1}\ (\!|\alpha_2|\!)_{\rho 2} \textbf{ IN} \\ & \textbf{IF } (\!|\gamma|\!)_\rho \textbf{ THEN } a_{\rho 1}(V_\rho,K_\rho) \textbf{ ELSE } a_{\rho 2}(V_\rho,K_\rho)\ ,\end{aligned}$$

where $(\!|\gamma|\!)_\rho$ means the application of the same translation scheme for formulæ as seen for φ, but adapted to the valuation V_ρ instead of V. The other transformations are equally trivial; we omit them for space. Note that it is necessary for our translation to use fresh symbols at every position – which indexing by ρ accomplishes – for TLA$^+$ forbids redefinition of a bound variable within its scope. Thus we cannot reuse, e.g. , V as a parameter in nested definitions.

Actions α_n are associated to nodes n via an operator. The default action is the identity; the rest is dispatched to individual operators $\alpha\_n(V,K)$:

```
α(V,K, n) ≜ CASE n=1 → α_1(V,K) ... □ OTHER → ⟨V,K⟩
```

Each of those operators is of the form

```
α_n(V₀,K₀) ≜ LET (|α_n|)_ε IN a(V₀,K₀)
```

The main purpose is to enclose the translation within a scope, so we can reuse the symbols for other nodes. (The alternative would be a double-subscript translation: $(\!|\alpha|\!)_{n,\rho}$). For instance, we have, in a non-recursive case (node 1 of the synthetic example):

```
α_1(V₀,K₀) ≜ \* {"A","B"} ! {"a"}
  LET a(V, K) ≜ ⟨V, [A ∈ {"A","B"} ↦ K[A] ∪ {"a"}] ▶ K⟩ IN a(V₀,K₀)
```

The translation of node 4, `[a & b] c := ⊤; b := ⊤`, is of the nested form

```
α_4(V₀,K₀) ≜ LET a(V, K) ≜ \* chain
    LET a1(V1, K1) ≜ \* ifthen a & b (guard)
        LET a1_2(V1_2, K1_2) ≜ ...  \* c := ⊤
```

The other actions present no further difficulties. We can now fully define our initial states. Note that Start events do bear actions, and initial states are computed *after* initial actions are taken:

```
vvals_init ≜  { v ∈ ℘ 𝕍 :  φ_init(v) ∧ φ(v) } (*SAT solver override*)
Init ≜ ∃ n ∈ oftype(EventStart):
    ∧ σ = [ f ∈ fl ↦ IF src(f)=n THEN 1 ELSE 0 ]
    ∧ ∃ VI ∈ vvals_init:
        LET KI ≜ [a ∈ 𝔸 ↦ ∅] IN LET VK ≜ α(VI,KI,n) IN
        V = VK[1] ∧ K = VK[2] ∧ L = [ a ∈ 𝔸 ↦ leak(V,K[a]) ]
```

Likewise, for the execution of all other nodes:

```
dexec(n) ≜
    ∧ LET VK_α ≜ α(V,K,n) IN V' = VK_α[1] ∧ K' = VK_α[2]
    ∧ L' = [ A ∈ 𝔸 ↦ leak(V',K'[A]) ]
    ∧ exec(n)
```

With this, the operational semantics is complete. The invariants of interest are straightforward to define:

```
LEAK(x) ≜ ∃ A ∈ 𝔸: x ∈ L[A]        SAFE(x) ≜ ¬LEAK(x)
SAFE_SET(X) ≜ ∀ x ∈ X: SAFE(x)     NO_LEAKS ≜ SAFE_SET(𝕍)
SAFE_AGENT(A) ≜ L[A] = ∅
```

Other predefined operators appear in the examples: `TOKENS`, which stands for the number of tokens active in the process, and the properties `DEADLOCK_FREE` and `TERMINATES`. They don't meaningfully interact with the epistemic layer, and we omit their definitions.

4 Leaks are coNP-complete: SAT Solvers to the Rescue

The TLA$^+$ specification of `leak(V,K_A)` (or `leak(φ,V,K_A)`, to make all parameters explicit) given in Sec. 3.2[p9] is TLC-executable, which is sufficient for small examples, but the time required to compute leaks in that way scales exponentially with the number of variables unknown to the agent: $O\left(2^{|\mathbb{V}\setminus K_A|}\right)$. This is not an accident, as the leak detection problem (**in:** φ, V, K_A, x **out:** $x \in \mathtt{leak}(\varphi, V, K_A)$?) is coNP-complete. Writing SAT(ψ) for *"is ψ satisfiable?"*, SAT(ψ) reduces to $x \notin \mathtt{leak}(\psi \vee x, \varnothing, \varnothing)$, showing the lower bound. Fortunately, we can also reduce `leak` to SAT calls. Writing $V|_K$ for the conjunction of literals $\bigwedge_{x\in K_A} l(x)$, with $l(x) \triangleq x$ if $x \in V$, and $\neg x$ otherwise, we have the new, equivalent definition:

$$\mathtt{leak}(V, K_A) \triangleq \left\{ x \in \mathbb{V} \setminus K_A \;\middle|\; \neg\left[\mathrm{SAT}(\varphi \wedge V|_K \wedge x) \wedge \mathrm{SAT}(\varphi \wedge V|_K \wedge \neg x)\right] \right\}.$$

Two satisfying valuations thus provide a polynomial-length certificate that a variable does *not* leak, proving the upper-bound.

Modern SAT solvers are, in practice, capable of handling *extremely* large formulæ, involving tens of millions of variables and clauses [8]. This definition reduces to a

linear number of calls to a solver (we use Z3 as backend). This is implemented via a TLC feature called "operator override", whereby one can easily override specific TLA$^+$ operators with a Java implementation. This is done efficiently by loading φ once, initially, while incremental solver scopes handle each $V|_K$, and each literal. Our implementation is caching and compatible with multiple TLC worker threads (each thread has its own independent Z3 instance and cache).

In practice, we can deal with fairly large state spaces. On our test machine, Arch Linux, AMD Ryzen 9 5900X, 32G RAM, using 24 worker threads, an intractable example (wrt. state space) is checked at a consistent rate of $3 \cdot 10^7$ states/min, whether it has 10, 100 or 200 variables. Without a SAT implementation override, TLC errors out immediately, and any naïve implementation would timeout, even though the leak problem is trivial in that example.

What then would make leak detection *non*-trivial? **(1)** The difficulty of SAT(φ). Decades of large-scale industrial applications of SAT solvers have shown this to be very unlikely to be a problem in practice [8]. **(2)** State-space explosion. Explicit access control can abstract away the value of data, but protecting data from deductive leaks generally requires modelling value-dependent decisions and outcomes. The next example, Fig. 4, in Sec. 5[p14], presents a worst-case scenario for us in that respect: we protect a vector of independent variables $\langle p_1, \ldots, p_n \rangle$, encoding a user's personal information. This creates 2^n initial states, one per possible user profile, creating as many disjoint transition systems.

Overall, we see leak detection as a straight-forward extension of explicit access model-checking, whose overhead cost can be mitigated by the use of specialised theory solvers and some care in the choice of the protected variables.

5 Example: Minimal Exposure Welfare Application

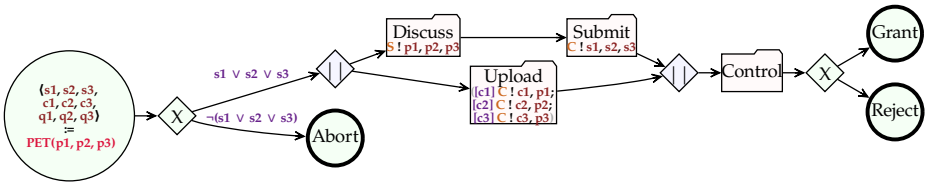


Fig. 4: Simplified workflow for application to services of the French social welfare services with minimal exposure of personal information.

In this section, we walk through a small workflow (Fig. 4) focusing on privacy-relevant aspects of the process of applying to services offered by the French social welfare. This workflow is based on the running example in [1]. In this article, the authors present a new Privacy Enhancing Technology (PET) for minimal and informed data collection in the case of data collection via forms. Their case study focuses on data collection during the application process for social welfare. The proposed PET aims to minimise the data collected by the agencies and to inform individuals about what can be deduced from the data they provide. We refer the

reader to [1] for a more thorough examination of the data privacy aspects of that example.

In our process, the User is a person characterised by a certain set of personal predicates p_i, $i \in 1..3$, who seeks to obtain all the services s_i, $i \in 1..3$ to which they are entitled (e.g. $s_2 \equiv$ "entitled to tax reduction"). Their entitlements are determined by the relevant business rules, captured by the workflow's structural invariant formula

$$s_1 \Leftrightarrow p_1 \vee (p_2 \wedge p_3) \quad \wedge \quad s_2 \Leftrightarrow p_1 \wedge \neg p_2 \quad \wedge \quad s_3 \Leftrightarrow p_1 \wedge \neg p_3 \ , \qquad (\varphi)$$

that is, for instance, an individual is entitled to service s_1 iff it satisfies the requirement $p_1 \vee (p_2 \wedge p_3)$.

Their request is handled by a Social worker S, who assists the User directly, and a Controller C, who examines the certificates which the User must provide to prove the relevant p_i — which implies revealing the value of p_i to C.

In accordance with their rights to confidentiality, the User does not wish to reveal to C (directly or via deductions) more personal information than is strictly necessary to prove their entitlements. **The User runs a PET** locally, on their device, to compute **(1)** the services to request (s_i), **(2)** which (minimal set of) certificates to provide ($c_i \equiv$ "provide certificate for p_i") **(3)** which data are exposed: this is coded by the variables $q_i \equiv$ "p_i is exposed, either directly by certificate or by deduction according to (φ)". If the user is entitled to nothing, they **abort**. Otherwise they **discuss** the case with their Social worker, who legitimately knows their information (but has no access to the certificates) and **submits** the request for services to the Controller. Separately, the User **uploads** the *necessary* certificates for the Controller. When C has all the required information, they **control** the validity of the request and the certificates provided and render a decision: either **grant** or **reject**.

This workflow involves a few technical points. There are 8 initial states: one for each possible User profile (i.e. $\langle p1, p2, p3 \rangle$ vector). Every other variable is determined by that. The initial formula φ_i could be absent (i.e. $\varphi_{\text{init}} = \top$), since the c_i and q_i are immediately altered by the start action, but in practice we set them all to $\bot$ to accelerate the computation of the initial states, as an implementation detail.

The start action receives the output of the PET (injected tuple), which is, for our purposes, a black box. In practice, we defined it as an injected operator with precomputed outputs:

```
PET(p1,p2,p3) ≜
    \* services s1,2,3, certifs c1,2,3, exposed pi: q1,2,3
    LET P ≜ ⟨p1,p2,p3⟩
    IN CASE P ∈ {⟨⊥,⊤,⊤⟩, ⟨⊤,⊤,⊤⟩} → ⟨⊤,⊥,⊥, ⊥,⊤,⊤, ⊥,⊤,⊤⟩
         □ P = ⟨⊤,⊤,⊥⟩ → ⟨⊤,⊥,⊤, ⊤,⊥,⊤, ⊤,⊤,⊤⟩ ...
```

Its contents are equivalent to [1, Fig. 1] and specify the values of the s_i, c_i, q_i variables given the valuation of the p_i.

Receiving the s_i from the PET may seem redundant, since they are already entirely determined by the p_i via φ, but by doing that we verify that the PET gives the correct services: an error would be caught by the φ-coherence invariant. The "Abort" event is important: making an explicit empty request would leak $p_1 = \bot$.

Note that the action `[c1]C!c1,p1` illustrates both interpretations of "access": `c1` is a resource (piece of paper, scan), whose truth value as a variable is irrelevant to φ, whereas we reason on the value of `p1`.

The invariants to be checked depend on a few trivial injected definitions, e.g. `C` is $\{c_1, c_2, c_3\}$, `S` is $\{s_1, s_2, s_3\}$, `PQ` is the set of the $\langle p_i, q_i \rangle$, etc. They are:

```
I1 ≜ ¬∃ c ∈ C: c ∈ K["S"]
I2 ≜ ∀ A ∈ 𝔸, ⟨c,p⟩ ∈ CP: c ∈ K[A] ⇒ p ∈ K[A]
I3 ≜ ∀⟨p,q⟩ ∈ PQ: q ∈ V ∨ SAFE(p)
I4 ≜ after(K) ⇒ ∀⟨p,q⟩ ∈ PQ: (q ∈ V) = (p ∈ K["C"] ∨ LEAK(p))
```

`I1` and `I2` are basic sanity-checks on how we wrote the workflow: the Social worker is not supposed to receive any certificate, and it is not possible to give a certificate without also revealing the corresponding personal data. `I3` is the most important: at no point in the process does the PET endanger the user. Data is either flagged as exposed via q_i, or it is safe (does not leak). While this is good, an easy way to satisfy that is for the PET to mark *everything* as exposed. That would certainly keep the user "safe", but is suboptimal. `I4` ensures that the user has *accurate* exposure information, that is to say, that data is marked as exposed *if and only if* it is. But that only applies *after* the Controller has received all information: certificates *and* service requests, which is given by

```
after(K) ≜ K["C"] ∩ C ≠ ∅ ∧ K["C"] ∩ S ≠ ∅
```

Without that precondition, the invariant is incorrect: for instance, in the initial state, q_i are already set, yet the Controller knows nothing. Running the model-checker, this process has 74 states, terminates, and satisfies all our invariants.

6 Conclusion & Future Works

We have introduced DPWN, a formally-specified, tool-supported subset of BPMN capable of specifying agent knowledge and deductive capabilities, and mechanically verifying data security properties (along with control flow properties) through model-checking. Taking deductions into account enables a finer-grained compliance with common regulatory frameworks regarding the "necessary" exposure of data, which we have illustrated through an example. Ongoing work focuses on developing a higher-level user-facing language, offering more palatable abstractions for data than bare Boolean variables: numerical types, comparisons, collections, etc, while continuing to leverage the power of SAT-solver backends. Other perspectives include other modes of deduction, e.g. deductions under full knowledge not only of φ but of the entire workflow, agents with differing individual knowledge of business logic, etc.

References

1. Anciaux, N., Frittella, S., Joffroy, B., Nguyen, B., Scerri, G.: A new PET for data collection via forms with data minimization, full accuracy and informed consent. Published in Proceedings of the 27th International Conference on Extending Database Technology (EDBT), 25th March-28th March, 2024 (2024). https://doi.org/10.48786/edbt.2024.08
2. Baader, F., Calvanese, D., McGuinness, D.L., Nardi, D., Patel-Schneider, P.F.: The description logic handbook: Theory, implementation and applications. Cambridge University Press (2007)
3. Ball, T., Podelski, A., Rajamani, S.K.: Boolean and cartesian abstraction for model checking C programs. In: Margaria, T., Yi, W. (eds.) Tools and Algorithms for the Construction and Analysis of Systems, 7th International Conference, TACAS 2001 Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2001 Genova, Italy, April 2-6, 2001, Proceedings. pp. 268–283. Lecture Notes in Computer Science, Springer (2001). https://doi.org/10.1007/3-540-45319-9_19, https://doi.org/10.1007/3-540-45319-9_19
4. Casciani, A., Agostinelli, S., Lespérance, Y., Marrella, A., Sardiña, S.: Formal semantics for knowledge representation and automated reasoning in bpmn process models. Information Systems **140**, 102718 (2026). https://doi.org/https://doi.org/10.1016/j.is.2026.102718, https://www.sciencedirect.com/science/article/pii/S0306437926000323
5. De Moura, L., Bjørner, N.: Z3: An efficient smt solver. In: International conference on Tools and Algorithms for the Construction and Analysis of Systems. pp. 337–340. Springer (2008)
6. European Parliament and Council of the European Union: Regulation (eu) 2016/679 of the european parliament and of the council of 27 april 2016 on the protection of natural persons with regard to the processing of personal data and on the free movement of such data (general data protection regulation). https://eur-lex.europa.eu/eli/reg/2016/679/oj/eng (2016), consolidated version, applicable from 25 May 2018
7. Fagin, R., Halpern, J.Y., Moses, Y., Vardi, M.Y.: Reasoning about knowledge. Cambridge, MA: The MIT Press (1995)
8. Ganesh, V., Vardi, M.Y.: On the unreasonable effectiveness of SAT solvers. In: Roughgarden, T. (ed.) Beyond the Worst-Case Analysis of Algorithms, pp. 547–566. Cambridge University Press (2020). https://doi.org/10.1017/9781108637435.032, https://doi.org/10.1017/9781108637435.032
9. Graf, S., Saïdi, H.: Construction of abstract state graphs with PVS. In: Grumberg, O. (ed.) Computer Aided Verification, 9th International Conference, CAV '97, Haifa, Israel, June 22-25, 1997, Proceedings. pp. 72–83. Lecture Notes in Computer Science, Springer (1997). https://doi.org/10.1007/3-540-63166-6_10, https://doi.org/10.1007/3-540-63166-6_10
10. Hassen, M.B., Keskes, M., Turki, M., Gargouri, F.: BPMN4KM: Design and implementation of a BPMN extension for modeling the knowledge perspective of sensitive business processes. Procedia Computer Science **121**, 1119–1134 (2017). https://doi.org/https://doi.org/10.1016/j.procs.2017.12.121, https://www.sciencedirect.com/science/article/pii/S1877050917328892, cENTERIS 2017 - International Conference on ENTERprise Information Systems / ProjMAN 2017 - International Conference on Project MANagement / HCist 2017 - International

Conference on Health and Social Care Information Systems and Technologies, CENTERIS/ProjMAN/HCist 2017
11. Kheldoun, A., Barkaoui, K., Ioualalen, M.: Formal verification of complex business processes based on high-level petri nets. Inf. Sci. **385**, 39–54 (2017). https://doi.org/10.1016/J.INS.2016.12.044, https://doi.org/10.1016/j.ins.2016.12.044
12. Krishna, A., Poizat, P., Salaün, G.: Checking business process evolution. Sci. Comput. Program. **170**, 1–26 (2019). https://doi.org/10.1016/J.SCICO.2018.09.007, https://doi.org/10.1016/j.scico.2018.09.007
13. Lamport, L.: Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers. Addison-Wesley (2002)
14. López, H.A., Hildebrandt, T.T.: Three decades of formal methods in business process compliance: A systematic literature review. Pre-print on arXiv: https://arxiv.org/abs/2410.10906 (2024). https://doi.org/10.48550/arXiv.2410.10906
15. Meyer, A., Pufahl, L., Fahland, D., Weske, M.: Modeling and enacting complex data dependencies in business processes. In: Daniel, F., Wang, J., Weber, B. (eds.) Business Process Management - 11th International Conference, BPM 2013, Beijing, China, August 26-30, 2013. Proceedings. pp. 171–186. Lecture Notes in Computer Science, Springer (2013). https://doi.org/10.1007/978-3-642-40176-3_14, https://doi.org/10.1007/978-3-642-40176-3_14
16. Object Management Group: https://www.omg.org/spec/BPMN (2013)
17. Pnueli, A.: The temporal logic of programs. Proceedings of the 18th Annual Symposium on Foundations of Computer Science (FOCS 1977) (1977). https://doi.org/10.1109/SFCS.1977.32
18. Poizat, P., Houhou, S., Baarir, S., Quéinnec, P., Kahloud, L.: A first-order logic verification framework for communication-parametric and time-aware bpmn collaborations (2021). https://doi.org/10.1016/j.is.2021.101765
19. Pullonen, P., Matulevičius, R., Bogdanov, D.: PE-BPMN: Privacy-enhanced business process model and notation. In: Carmona, J., Engels, G., Kumar, A. (eds.) Business Process Management. pp. 40–56. Springer International Publishing, Cham (2017)
20. Russell, N., van der Aalst, W.M.P., ter Hofstede, A.H.M.: Workflow Patterns: The Definitive Guide. MIT Press (2016), https://mitpress.mit.edu/books/workflow-patterns
21. Yuan, Y., Panagiotis, M., Leslie, L.: Model checking TLA+ specifications. In: Lecture Notes in Computer Science. vol. 1703, pp. 54–66. Springer (1999). https://doi.org/10.1007/3-540-48153-2_5