Add Data into Business Process Verification: Bridging the Gap between Theory and Practice

Riccardo De Masellis[1], **Chiara Di Francescomarino**[1], **Chiara Ghidini**[1], **Marco Montali**[2], **Sergio Tessaris**[2]
[1] FBK-IRST, Via Sommarive 18, 38050 Trento, Italy
[2] Free University of Bozen–Bolzano, piazza Università, 1, 39100 Bozen-Bolzano, Italy

Abstract

The need to extend business process languages with the capability to model complex data objects along with the control flow perspective has lead to significant practical and theoretical advances in the field of Business Process Modeling (BPM). On the practical side, there are several suites for control flow and data modeling; nonetheless, when it comes to formal verification, the data perspective is abstracted away due to the intrinsic difficulty of handling unbounded data. On the theoretical side, there is significant literature providing decidability results for expressive data-aware processes. However, they struggle to produce a concrete impact as being far from real BPM architectures and, most of all, not providing actual verification tools. In this paper we aim at bridging such a gap: we provide a concrete framework which, on the one hand, being based on Petri Nets and relational models, is close to the widely used BPM suites, and on the other is grounded on solid formal basis which allow to perform formal verification tasks. Moreover, we show how to encode our framework in an action language so as to perform reachability analysis using virtually any state-of-the-art planner.

1 Introduction

The need to extend business processes with the capability to handle complex data objects has been increasingly recognized both in the BPM and AI areas and has led to significant practical and theoretical advances in the BPM field (Hull 2008; Meyer, Smirnov, and Weske 2011; Reichert 2012; Calvanese, De Giacomo, and Montali 2013; Hull and Motahari Nezhad 2016). On the practical side, several well-established suites, capturing both the process control-flow and its relevant data, are nowadays available as commercial and non-commercial tools. Examples are the Bizagi BPM Suite, Bonita BPM, Camunda and YAWL. Despite different modeling choices all such tools share a common feature: *the way data are modified* is often hidden inside the logic of activities/tasks implemented, e.g., with Java classes, hence resulting in an essentially activity-centric model, where data are introduced in an ad-hoc way as a sort of "procedural attachment" (Calvanese, De Giacomo, and Montali 2013). As a consequence, when coming to the formal verification of data related aspects, these tools either offer only basic features (such as Data Type checks) without considering the interaction between control- and data-flow, or they fail to incorporate data into verification questions, thus producing misleading answers. This does not come as a surprise as, when analyzing the evolution of data in a process which interacts with the external world, unboundedly many new, i.e., *fresh*, data values have (in general) to be considered, making verification of even simple properties undecidable. On the theoretical side there is a significant body of literature on the boundaries of decidability and complexity for the verification of data-aware processes against different formal properties. The problem of these frameworks is that either they are far from the modeling languages used in real BPM suites, and lacking any tool support, or the data model is not adequate for expressive scenarios (See § 2 and § 7).

Copyright © 2017, Association for the Advancement of Artificial Intelligence (www.aaai.org). All rights reserved.

In this paper we aim at bridging the gap between theory and practice by providing a concrete framework, called RAW-SYS (from: Relational-AWare SYStem) for modeling and verifying data-aware processes as represented by well established BPM suites. In particular we provide:

1. a language for modeling the **control-flow**, the **data** and their **interaction** based on the most popular, yet formal, frameworks for modeling these three components, namely Petri Net (PN) (Van Der Aalst 1998), relational models, and actions à la Data Centric Dynamic System (DCDS) (Bagheri Hariri et al. 2013) (§ 4);
2. a reference architecture that mimics how data-aware processes are represented by well established BPM suites (§ 2) and an encoding of such an architecture in RAW-SYS (§ 4);
3. a decidable, yet very expressive, customization of the reference architecture to enable formal verification (§ 5);
4. an actual verification mechanism based on an encoding of RAW-SYS into an action language which allows for exploiting automated planners (§ 6).

2 Motivations and Architecture

Commercial and non-commercial BPM suites, such as the ones listed in Table 1, nowadays support the modeling of both control and data flow and provides consistency and verification support. Focusing on the latter we can identify three different levels at which control and data flow are verified. We illustrate them with the help of the following simple example where activity A is followed by an exclusive choice (XOR-split) that

Tool	Workflow Language	Data Model	Data Handling	Formal Verification
Bonita	BPMN	ER	global,local	*None*
Bizagi	BPMN	ER	global,local	*None*
YAWL	YAWL	XMLschema	global,local	(1)
Camunda	BPMN	*None*	global,local	*None*

Table 1: Tool analysis

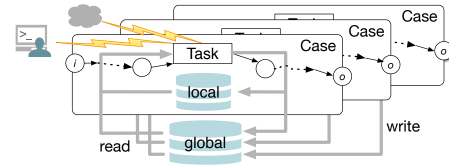
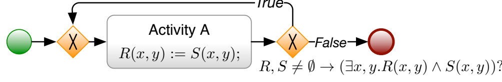


Figure 1: RAW-SYS architecture.

leads the process either to terminate or to re-execute A.



Activity A modifies data by setting relation R to be equal to S and the XOR-split requires conditions (on data) to be specified on each outgoing flow: specifically, whether there is a common tuple to both R and S, if they are not empty. At level (1) verification focuses only on the control flow. In this case the process is considered *sound* as one path exists that leads to termination. At level (2) verification takes into account also the conditions on arcs, e.g., checking whether they are satisfiable. Again the process is considered *sound*. At level (3) verification takes also into account the effects of activities on data: only in this case we can conclude that the process never terminates, as all tuples in S are also in R.

Although the process never terminates, when verified by existing BPM suites such as Bizagi[1] or YAWL (ter Hofstede et al. 2010), this critical issue is not revealed. Indeed YAWL offers verification features limited to the control flow (i.e., at level (1)) and thus it wrongly reports that such a process can always reach the termination state. All other tools in Table 1 instead only offer a simulation environment (i.e., no formal verification) that checks whether the process passes through all the sequence flows, without taking into account data.

Framework	Form. Verif.
CPN	(3^-)
Conceptual WF-Nets	(3^-)
DCDS	(3)

Table 2: Framework analysis

Moving from actual tools to existing theoretical frameworks the situation improves (see Table 2). Colored Petri Nets (CPN) and extensions of Workflow-Nets (WF-Nets) (Sidorova, Stahl, and Trcka 2011) can offer verification support that take into account both conditions on labels and some interaction between activities and data. Nonetheless these frameworks suffer of two problems: first, they do rely on an explicit data model but rather encode data within the Net thus making difficult to model tools which are mostly based on a relational data model; second, decidability is guaranteed only by strongly limiting the number of colors/types (CPNs) or abstracting from actual values (WF-Nets) (see § 7). For the above reasons, they offer a restricted form of (3), whence (3^-), thus posing a serious limitation to process execution which often need unboundedly many new, i.e., *fresh*, data values. Conversely, Data Centric Dynamic System (DCDS) does offer - from a theoretical point of view - a more expressive, yet decidable, reference framework. Thus one may think to provide a specific encoding from each BPM tools to DCDS, in order to perform verification al level (3). Such a solution is however not practical for (at least) two important reasons: on the one hand the formalization of control and data flow in DCDS is provided in terms of a STRIP-like language that is extremely abstract and far from the languages and system architectures used in BPM suites, and on the other hand DCDSs do not have any tool support to concretely perform verification.

[1] http://www.bizagi.com/

This analysis has motivated us towards the introduction of RAW-SYS, a new framework for modeling and verifying data-aware processes as represented by well established BPM suites. To do that RAW-SYS is based on two basic pillars: first, a **conceptual model** close to the ones used by BPM suites and, at the same time, amenable to formal verification; second, a **system architecture** that mimics the way BPM suites deal with processes executions and data.

Concerning the conceptual model, we define RAW-SYS on top of three reference components: *(i)* Petri Net (PN), for specifying the process control-flow; *(ii)* relational models with expressive constraints for describing data; and *(iii)* actions à la DCDS for expressing the interaction between control-flow and data. These components are, undoubtably, among the most popular formal frameworks available in literature. Also, as we can observe in Table 1, PNs and relational models act as suitable formal models for the specific workflows and data models used by BPM suites[2].

Concerning the system architecture, we adopt the one illustrated in Fig. 1. As the majority of BPM suites, we allow for both modeling and executing process (and data) instances. Therefore, we distinguish the intensional level /schema of a RAW-SYS system which we call *model*, from the extensional level/instance, called *snapshot*, which captures the status of the system at a certain time. From a high-level perspective, a RAW-SYS model is made up by a data store and a set of process models. Following a common practice in BPM suites (see column Data Handling in Table 1), the data store is structured in a *global data store*, that is a standard relational database schema with integrity constraints common to all process models and a *local data store*, which is again a full-fledged relational database schema, defined within a single process model. At runtime processes are instantiated in a number of *cases*. Thus a snapshot contains all cases active in a certain moment of time. Note that each case creates an instantiation of the local data store private to the case itself, which is hence used to keep auxiliary information that are

[2] For the usage of PNs to provide a formal semantics of BPMN see (Dijkman, Dumas, and Ouyang 2008); for the integration of XML schemas with relational database systems see (Kappel, Kapsammer, and Retschitzegger 2004).

of interest only as long as the case is active. Conversely, the global data store provides the "stable" memory of the company, and can be accessed by every case. Note also that tasks can interact with external (human or system) agents and the interaction can result in modifying data, including the injection of new, fresh data into the system (e.g, think of a user filling a form with arbitrary data).

We conclude this section presenting a reimbursement ($\mathbb{RB}$) example used throughput the paper to illustrate our work.

Example 1 *Company* `GoodsKit` *is equipped with an informative system which manages, among others, the employees' reimbursement procedure ($\mathbb{RB}$) for their business trips. At runtime, the system deploys an architecture as in Fig. 1 where each process is instantiated by a number of cases. E.g., an $\mathbb{RB}$ case deals with the trip reimbursement of employee* `john` *to* `NewYork`*. As customary, data are contained in a global and a local component: an example of local variable is one that maintains the status of the request.*

3 The Workflow Nets modeling language

Petri Nets (PNs) is a widely-known language for modeling distributed systems that has become the de-facto standard for the formal representation of (the control-flow of) business processes (van der Aalst and Stahl 2011). Structurally, a PN is a directed bipartite graph with two node types, called *places* and *transitions*, connected via directed arcs. Connections between two nodes of the same type are not allowed.

Definition 1 (Petri Net) *A* Petri Net *is a triple $\langle P, T, F\rangle$ where P is a set of* places*; T is a set of* transitions*, such that $P \cap T = \emptyset$; $F \subseteq (P \times T) \cup (T \times P)$ is the flow relation describing the "arcs" connecting places and transitions.*

The *preset* of a transition t is the set of its input places: ${}^\bullet t = \{p \in P \mid (p,t) \in F\}$. The *postset* of t is the set of its output places: $t^\bullet = \{p \in P \mid (t,p) \in F\}$. Intuitively, places represent states/conditions associated to threads (i.e., *tokens*) dynamically moving through the PN, whereas transitions represent atomic tasks/operations used to evolve such tokens from one state to another. To characterize the global configuration of the system, tokens are distributed over places. Technically, this is done by *marking* the net, where a marking is a total mapping $M : P \mapsto \mathbb{N}$ indicating how many tokens are present in each place of the PN.

PNs come with a graphical notation where places are represented by circles, transitions by rectangles, and tokens by full dots within places. Fig. 2 depicts a PN with a marking $M(p_0) = 2$, $M(p_1) = 0$, $M(p_2) = 1$. The preset and poset of t are $\{p_0, p_1\}$ and $\{p_2\}$, respectively. The idea of using PNs to model processes starts from the observation that business processes are basically ordered set of tasks. Thus they can be mapped onto a PN in the following way: tasks are modeled by transitions and precedence relations are modeled by places. However, processes have specific characteristics: they have a clear starting and completion state, with control-flows connecting such two extreme points. This observation resulted in the definition of workflow nets (WF-nets) (Van Der Aalst 1998).

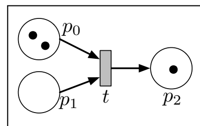
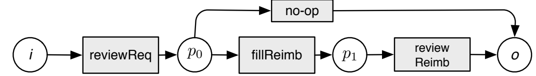


Figure 2: A PN.

Definition 2 (WF-net) *A Petri net $\langle P, T, F\rangle$ is a WF-net if it has a single source place i, a single sink place o, and every place/transition is on a path from i to o, i.e., for all $n \in P \cup T$, $(i, n) \in F^*$ and $(n, o) \in F^*$, where F^* is the reflexive transitive closure of F.*

Example 2 *The WF-net modeling the $\mathbb{RB}$ control-flow is:*



At the start of the process a request is examined. If it is not approved the process terminates; if it is approved, the employee can fill a reimbursement which will be then reviewed before termination. The no-op transition is needed to prevent connections between nodes of the same type.

The semantics of a PN (thus also of a WF-net), and in particular the notion of *valid firing*, defines how transitions route tokens trough the net so that they correspond to a process execution. A firing of a transition $t \in T$ from M to M' is *valid*, in symbols $M \xrightarrow{t} M'$, iff (i) t is enabled in M, i.e., $\{p \in P \mid M(p) > 0\} \supseteq {}^\bullet t$; and (ii) the marking M' satisfies the property that for every $p \in P$: $M'(p) = M(p) - 1$ if $p \in {}^\bullet t \setminus t^\bullet$; $M'(p) = M(p) + 1$ if $p \in t^\bullet \setminus {}^\bullet t$ and $M'(p) = M(p)$ otherwise.

A *case* of a WF-Net is a sequence of valid firings $M_0 \xrightarrow{t_1} M_1, M_1 \xrightarrow{t_2} M_2, \ldots, M_{k-1} \xrightarrow{t_k} M_k$ where M_0 is the marking with a single token in i. We sat that a WF-Net *reaches* a marking M if there exists a finite sequence $t_1, \ldots, t_k$ of transitions such that $M_0 \xrightarrow{t_1} M_1, \ldots, M_{k-1} \xrightarrow{t_k} M$.

The number of tokens flowing through a PN is usually subject to a maximum threshold that is never exceeded. PNs enjoying this property are called "safe".

Definition 3 (k-safeness) *A marking of a PN is k-safe if it assigns no more than k tokens to each place. A PN is k-safe if the initial marking M_0, and all markings reachable from M_0, are k-safe.*

4 The RAW-SYS Framework

We start the formal presentation from the data stores. We fix once and for all a countably infinite set of constants Δ to be used as domain and we observe that technically there is no difference between the global and local data stores.

Definition 4 (Data store) *A (local or global) data store is a tuple $\mathcal{D} = \langle \mathcal{R}, \mathcal{C}, \mathcal{I}_0\rangle$, where:*

- *$\mathcal{R}$ is a* database schema*, i.e., a set of relation schemas;*
- *$\mathcal{C}$ is a set of* safe range FO-constraints[3] *over $\mathcal{R}$, capturing the real-world constraints of the application domain;*
- *$\mathcal{I}_0$ is the* initial database instance *of $\mathcal{D}$, i.e., a data store instance conforming to $\mathcal{R}$, satisfying the constraints $\mathcal{C}$, and made of values in Δ.*

Example 3 *Due to lack of space, we present some relations of the $\mathbb{RB}$ local data store only (underlined attributes are primary keys): CurrReq($\underline{empl, dest}, st$) listing the employee that requested a reimbursement, the trip destination and the*

[3]This is standard. Recall that relational algebra is safe range by construction.

status of the request and $TrvlMax(ma)$ containing the maximum amount estimated by the travels office for the trip.

We now move to processes, which are modeled in a prescriptive fashion leveraging standard WF-nets, but we notably enrich them with data. We call the resulting nets Relational-AWare workflow nets (*RAW-nets*). Intuitively, RAW-nets are standard WF-nets equipped with a (local) data store as in Definition 4. We concentrate on 1-safe nets, which generalize the class of *structured workflows* and are the basis for best practices in process modeling (Kiepuszewski, ter Hofstede, and Bussler 2013). It is important to notice that our approach can be seamlessly generalized to other classes of Petri nets, as long as it is guaranteed that they are k-safe. This reflects the fact that the process control-flow is well-defined. Also, transitions are data-aware: their execution is guarded by queries over the local and global data store, and their effects update the local and the global data store.

Definition 5 (Relational-aware WF-net) *A RAW-net over a global data store $\mathcal{D}_\mathcal{G}$ is a tuple $\langle \mathcal{D}, P, T, F, \mathcal{F}, \mathcal{A}, \mathcal{G} \rangle$ where:*

- *$\mathcal{D}$ is the process local data store (as in Def. 4) with $\mathcal{I}_0 = \emptyset$;*
- *$\langle P, T, F \rangle$ is a WF-net;*
- *$\mathcal{F}$ is a function that associates each transition with a finite set of* functions *from $\bigcup_{n \geq 0}(\Delta^n \mapsto \Delta)$, each representing the interface to a (nondeterministic)* external service*;*
- *$\mathcal{A}$ is a function that associates each transition t with an* action *(see later);*
- *$\mathcal{G}$ is a function that associates each transition with a safe range query over $\mathcal{D} \cup \mathcal{D}_\mathcal{G}$ (the guard).*

Transition actions may include parameters (see later): in that case the guard free variables must match the parameters of the corresponding tasks.

Among the many different data interaction formalisms existing in the literature, we adopt the approach in (Bagheri Hariri et al. 2013; Montali and Calvanese 2016), which allows us to express virtually any pattern of update, including CRUD operations over the data stores, but also bulk operations that simultaneously manipulate large portions of the data store at once. Intuitively, an *action* is described by a set of add and remove operations on data store instances "conditioned" by a domain independent query. More formally an action $\textsc{act}(\mathsf{p}_1, \ldots, \mathsf{p}_\mathsf{n}) : \{e_1, \ldots, e_m\}$ is characterized by: *(i)* the name $\textsc{act}$; *(ii)* a list of parameters $\vec{p} = \mathsf{p}_1, \ldots, \mathsf{p}_\mathsf{n}$ and *(iii)* a set of effect $e_1, \ldots, e_m$. The parameters are substituted with actual values $\vec{\mathsf{d}}$ when the action is invoked. Such values are every answer of of the corresponding guard query given by $\mathcal{G}$ on the local and/or global data source. Each effect e_i is of the form: $Q(\vec{\mathsf{p}}, \vec{x}) \rightsquigarrow$ **add** $A(\vec{\mathsf{p}}, \vec{x}, \vec{y})$ **del** $D(\vec{\mathsf{p}}, \vec{x})$ where Q is a domain independent query over a data schema with open variables $\vec{x}$, while A and D are sets of facts over (possibly another) schema. Intuitively, Q is used to select some values (its answers) that are then used to add and/or remove facts. Notice that added facts A may include Skolem terms $\vec{y}$ that model the interaction with external services: at runtime Skolem terms are substituted by the value returned by issuing the corresponding service call given by $\mathcal{F}$. Some of those services can be guaranteed to return fresh values (i.e. constants not included in the active domain) and this is essential to enable the creation of new objects via new identifiers. The effects e_i are assumed to take place simultaneously.[4]

Example 4 *We now show a simple action of the running scenario. Activity $reviewRequest$ examines the employee's request to leave for a business trip and evaluates the maximum reimbursable amount. The (only) effect of the corresponding action $\textsc{rvwReq}()$ (with no parameters) is specified as:*

$$\begin{array}{l} CurrReq(e, d, s) \rightsquigarrow \\ \quad \textbf{\textit{del}}\ \{CurrReq(e, d, s)\} \\ \quad \textbf{\textit{add}}\{CurrReq(e, d, \mathbf{status}()), TrvlMax(\mathbf{ma}())\} \end{array}$$

*In order to update the request status, the tuple representing the current travel request in $CurrReq$ must be first deleted and then added with the new status (recall that additions have higher priority than deletions). Query $CurrReq(e, d, s)$ selects such a tuple, effect **del**$\{CurrReq(e, d, s)\}$ deletes it while **add**$\{CurrReq(e, d, \mathbf{status}())\}$ adds the same tuple but with a new value for the status. Also, the value of the max reimbursable amount is added by the fact $TrvlMaxAmnt(\mathbf{ma}())$. We use of functions $\mathbf{status}()$ and $\mathbf{ma}()$ to model unknown values coming from the external environment, in our case the company travels office*[5]*.*

The valid firing of a transition – in addition to the usual marking conditions on the input places, as specified by the RAW-net – is conditioned by the satisfiability of the guard and the successful application of the action. Note that an action might be unsuccessful because of a violation of a constraint in the local data store.

Finally, a RAW-SYS model simply incorporates the global data store together with a set of processes:

Definition 6 (RAW-SYS model) *A RAW-SYS model is a tuple $\langle \mathcal{D}_G, \mathcal{W} \rangle$ where:*

- *$\mathcal{D}_G$ is a global data store as in Definition 4;*
- *$\mathcal{W}$ is a set of RAW-nets as in Definition 5 over the global data schema $\mathcal{D}_\mathcal{G}$.*

Semantics is provided in term of global states (*snapshots*) that include the set of all active cases as well as the instance of the global data store.

Definition 7 (RAW-SYS snapshot) *A snapshot s of a RAW-SYS model $\langle \mathcal{D}_G, \mathcal{W} \rangle$ is a tuple $\langle \mathcal{I}_S, \mathcal{K} \rangle$ where:*

- *$\mathcal{I}_S$ is an instance of the global data store $\mathcal{D}_G$;*
- *$\mathcal{K}$ is a set of cases, i.e., tuples $\langle w, (M, I) \rangle$ where $w \in \mathcal{W}$ and (M, I) represents the state of the RAW-net w, namely its marking M and the instance I of its local data store.*

The behavior of the system is described by means of all the possible sequences of snapshots evolving from the initial one. The initial snapshot s_0 has an empty set of cases (none of the processes is active), and the global instance $\mathcal{I}_0$ specified, i.e., $s_0 = \langle \mathcal{I}_0, \emptyset \rangle$. It is worth noting that, due to the presence of external service calls and also due to the possibility of nondeterministically spawning new process cases, the execution semantics of a RAW-SYS needs in general to account for an

[4] As in STRIPS, additions have higher priority than deletions.

[5] If we require a function to return a value from a given set, e.g., $\mathbf{status}()$ from $\{\texttt{accp}, \texttt{rejc}\}$, it is enough to include a foreign key constraint to an auxiliary relation containing `accp` and `rejc`.

infinite number of states, as well as truly infinite runs that may visit infinitely many different data store instances.

Sequences of valid snapshots are defined according to two types of transitions that makes the system evolve nondeterministically. In what follows we assume the current snapshot to be $s = \langle \mathcal{I}_S, \mathcal{K} \rangle$.

Creation of a new case: A new case $\langle w, (M_0, I_0) \rangle$ of process $w \in \mathcal{W}$ is created, where M_0 is the initial marking and local data store instance I_0 is empty. The global data store is untouched: the new snapshot is therefore $s' = \langle \mathcal{I}_S, \mathcal{K}' \rangle$ with $\mathcal{K}' = \mathcal{K} \cup \{\langle w, (M_0, \emptyset) \rangle\}$.

Case firing: one of the transitions $t \in T$ of an active case $\langle w, (M, I) \rangle \in \mathcal{K}$ with $w = \langle \mathcal{D}, P, T, F, \mathcal{F}, \mathcal{A}, \mathcal{G} \rangle$ can be executed by picking a suitable parameters substitution $\vec{\mathsf{d}}$ from the answers of query $\mathcal{G}(t)$ evaluated on $\mathcal{I}_S \cup I$ and for the involved service calls $\mathcal{F}(t)$. The corresponding action $\mathcal{A}(t)$ with actual parameters $\vec{\mathsf{d}}$ is fired and possibly updates the local data store instance from I to I' and the global data store from $\mathcal{I}_S$ to $\mathcal{I}'_S$. Also the net marking is updated to M', and we distinguish two cases:

(i) if M' is a final marking for w, then the process terminated and it is removed from the set of active cases, resulting in a new snapshot $s' = \langle \mathcal{I}'_S, \mathcal{I}_A, \mathcal{K}' \rangle$ with $\mathcal{K}' = \mathcal{K} \setminus \{\langle w, (M, I) \rangle\}$;

(ii) otherwise the new snapshot is $s' = \langle \mathcal{I}'_S, \mathcal{I}_A, \mathcal{K}' \rangle$ with $\mathcal{K}' = \mathcal{K} \setminus \{\langle w, (M, I) \rangle\} \cup \{\langle w, (M', I') \rangle\}$.

Notice that in both cases, if I' or $\mathcal{I}'_S$ are not legal instances for $\mathcal{D}$ or $\mathcal{D}_G$, i.e., they do not satisfy the integrity constraints, then the transition cannot be fired.

5 Verification of RAW-SYS models

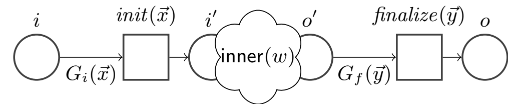
We now consider verification of RAW-SYS models, focusing on fundamental dynamic properties such as reachability as well as model checking against first-order temporal logics. In our setting, reachability amounts to check whether there exists a run of the RAW-SYS under study that starts from the initial state and eventually achieve a state whose data satisfy a boolean query of interest. Since the execution semantics of RAW-SYS gives raise to an infinite-state transition system, verification is much more challenging than in the conventional, finite-state setting (Calvanese, De Giacomo, and Montali 2013). In particular, even the most basic forms of reachability are highly undecidable in the general case. We hence exploit reachability as a test for isolating classes of RAW-SYS: whenever reachability turns out to be undecidable for a class, we consider it not amenable to verification.

In this work, we rely on the notion of *state-boundedness*, which has been extensively exploited for providing strong, robust decidability conditions in a plethora of data-aware process frameworks (Belardinelli, Lomuscio, and Patrizi 2012; Bagheri Hariri et al. 2013; De Giacomo, Lesperance, and Patrizi 2012). Essentially, state-boundedness requires to limit, a-priori, the number of objects that can co-exist in the same state. A state-bounded system still accepts unboundedly many different objects to appear within and across runs.

In all such previous works, there is a single, global data store, whose size is subject to the (state) bound. In our setting, there are three information sources to which such notion of boundedness can be applied: the global data store, the local data store, and the number of running cases. We respectively call the three corresponding notions of state boundedness *global size-*, *local size-*, and *case-boundedness*. Two research questions consequently arise: *(1)* Can state-boundedness help towards decidability of verification over RAW-SYSs? *(2)* If so, which of the information sources must necessarily be bounded towards decidability?

Undecidability. We attack the second research question, showing that as soon as one the RAW-SYS information sources is not size-bounded, reachability of a query constituted by an atomic proposition is undecidable even when the modeling elements of the framework are severely restricted. More specifically, we consider a class of RAW-SYS called *isolated*: $\langle \mathcal{D}_G, \mathcal{W} \rangle$ is *isolated* if every RAW-net w in $\mathcal{W}$ satisfies the following two conditions:

1. w has the following shape:



   where $\mathsf{inner}(w)$ is a workflow net.
2. All guards and actions attached to $\mathsf{inner}(w)$ refer only to the local data store of the RAW-net (i.e., they do not read, nor write, $\mathcal{D}_G$).

Intuitively, in an isolated system each case interacts with the global data store only when it is created, for initializing the local data store, or when it completes its execution, for determining which local data have to be persistently stored. All other guards and actions only operate over the local data store, thus limiting the interactions with the other, simultaneously running, cases.

We report in the following the results for each of the three dimensions of state-boundedness: isolated RAW-SYSs where no bound is imposed on the local data stores, on the global data stores and on cases. The proofs of the first two theorems are by reduction from the halting problem for deterministic two-counter machines (2CMs), well-known to be undecidable, which can be simulated by isolated global size- (local-size) and case-bounded RAW-SYSs. The proof of the third theorem requires instead a more convoluted approach.

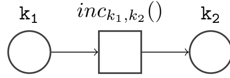
Theorem 1 *Checking reachability of an atomic proposition is undecidable over isolated, global size-bounded and case-bounded RAW-SYSs, where: (i) the global data store contains only propositions; (ii) there is only one RAW-net equipped with a local data store constituted by two unary relations; (iii) there is at most one running case.*

Proof 1 (Proof Sketch) The proof is by reduction from the halting problem for deterministic, two-counter machines (2CMs), well-known to be undecidable (Minsky 1967).

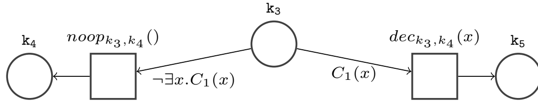
We fix a simple, isolated RAW-SYS $\mathcal{S}$ with the following features: *(i)* the global data source contains only a single proposition *Hit* used as a flag; *(ii)* the single RAW-net w of $\mathcal{S}$ has a data store equipped with two unary relations C_1 and C_2. Net w has a init transition that lets only one single token flow into the internal isolated workflow net, and a finalize transition that raises the *Hit* flag in the global store. The internal workflow net $\mathsf{inner}(w)$, in turn, encodes a 2CM

as follows. The instruction identifiers of the 2CM become places of $\mathsf{inner}(w)$, where the first instruction corresponds to the input place of $\mathsf{inner}(w)$, and the halting state of the 2CM corresponds to its output place. The values of the two counters correspond to the size of the extensions of the two relations C_1 and C_2, which are initially empty. In this light, increment and conditional decrement of the first counter are simulated as follows:

- An increment operation "$\mathtt{k}_1$: c_1++; goto $\mathtt{k}_2$" is simulated in $\mathsf{inner}(w)$ by the net fragment below where $inc_{k_1,k_2}()$: $\{true \rightsquigarrow \textbf{add}\{C_1(\underline{\textbf{newval}}())\}\}$ injects a new value in C_1 through the fresh service call $\underline{\textbf{newval}}()$.



- A conditional decrement operation "$\mathtt{k}_3$: if c_1==0 then goto $\mathtt{k}_4$; else goto $\mathtt{k}_5$" is instead simulated using the net fragment below.



The left branch captures the case where the counter is 0, which only requires to update the program counter, and in fact $noop_{k_3,k_4}()$: $\{\}$. The right branch captures the case where the counter is positive, and must consequently decrease by one unit. This is captured by guard $C_1(x)$, which nondeterministically picks an element from C_1, and by the related action $dec_{k_3,k_4}(x)$: $\{true \rightsquigarrow \textbf{del}\{C_1(x)\}\}$, which removes it. Similar structures are used to simulate increment and conditional decrement for counter 2. It is then easy to see that the 2CM halts if and only if $\mathcal{S}$ reaches a state where *Hit* holds.

Theorem 2 *Checking reachability of an atomic proposition is undecidable over isolated, local size-bounded and case bounded RAW-SYS, where: (i) the global data store contains only propositions and two unary relations; (ii) there is only one RAW-net equipped with an empty local data store; (iii) there is at most one running case.*

Proof 2 (Proof Sketch) The proof is similar to that of Theorem 1, with the difference that the unary relations used to simulate the counters are now in the global data store. The global data store is also equipped with a finite set of atomic propositions, one per state of the 2CM. In a given snapshot, only one of such atomic propositions holds (modeling that the 2CM is in a certain current state). The single RAW-net has a trivial structure with an input and output places connected by a single, no-op transition. Upon creation of an instance of such a RAW-net, it is checked which of such propositions currently holds, triggering a suitable, immediate update of the current state (i.e., substituting the current proposition with the next one, in accordance to the control-state update of the 2CM), and an update on the extension of the two counter relations, with the same strategy discussed in the proof of Theorem 1.

Theorem 3 *Checking reachability of an atomic proposition is undecidable over isolated, local and global size-bounded RAW-SYSs, where: (i) the global and local data stores contain only unary relations, whose extension contains at most one tuple (i.e., they act as registers); (ii) there is only one RAW-net equipped with an empty local data store; (iii) there is at most one running case.*

Proof 3 (Proof Sketch) This case is the trickiest as we cannot anymore exploit the same technique adopted in the previous two cases, because it is not possible anymore to exploit the local/global data store to remember the value of the two counters. Furthermore, when a certain process case becomes running, its evolution cannot be affected by that of other running cases, since it only works on its own local data.

Let w be the single RAW-net of $\mathcal{S}$. The WF-net of w is again the trivial net containing a single, no-op transition. However, it is now associated to two sophisticated updates to be applied when an instance of w is created or terminates its execution,

The two counters are simulated using two "chains" of running cases, where the value of the counter is the length of the chain minus 1. The main difficulty is how to rigidly keep track of the ordering between cases in a chain, and how to properly manipulate the chain, given the fact that the information about the chain itself cannot be stored anywhere, being all the data stores bounded. We attack this problem as follows. First of all, each instance of w exposes itself via a "ticket", i.e., a unique identifier that is made explicit in the global data store (this is necessary, since the internal case identifiers are not visible). The global data store remembers the current, control-state of the 2CM by adopting the same technique as in the proof of Theorem 2. It also remembers the extreme points of the two chains (i.e., the tickets of the instances at their top/bottom).

Since the 2CM is deterministic, we can assume that each control-state has a unique successor state obtained via a counter-increment operation, or two successor states, one achieved when a counter is positive and gets decremented, the other achieved when the same counter is zero. Increment is then simulated by allowing for the creation of a new w-instance only if the current control-state has an increment transition. Upon creation, the instance consumes the information about the instance that is currently at the top of the corresponding chain, remembering the corresponding ticket in a local, "previous ticket" relation. At the same time, it generates a fresh ticket identifying itself, and updates the global data store by declaring that this ticket is now at the top of the chain. This immediately simulates increment, and therefore the very same update also updates the control-state.

Decrement transitions for a counter are simulated by the termination of the running w-instance that is at the top of the corresponding chain. However, termination is not explicitly controllable in the specification: all running w-instances evolve in parallel and in isolation to each other, and consequently there is no explicit way of selecting only the instance at the top of the chain and induce its termination. To enforce this, we leverage the fact that a RAW-net case can properly terminate only if the corresponding update satisfies the constraints of the global data store. In particular, we make sure that whenever a w-instance wants to terminate, a constraint is violated if the ticket of that instance does not correspond to that at the top of the chain. When the top-instance is picked,

it successfully terminates by updating the control-state, and declaring that the top ticket is now the one store in its "previous ticket" relation.

Transitions triggered by a zero test are simulated in a similar way, discriminating them from decrement transitions by simply checking whether the selected, running w-instance is not only at the top of the chain, but also at the bottom (i.e., the chain contains only such an instance).

Decidability for Bounded RAW-SYSs. The undecidability results in Theorem 1–3 show that as soon as one of the three information sources has unbounded size, reachability turns out to be undecidable. We thus analyze the situation where all of them are bounded. We simply refer to such systems as *size-bounded RAW-SYSs*. We stress that such systems are by no means finite-state: they allow for storing unboundedly many data within and across system runs, provided that such data do not accumulate in the same snapshot.

For bounded RAW-SYSs, we prove the following positive result, thanks to a reduction to DCDSs.

Theorem 4 *Checking reachability over size-bounded RAW-SYSs is decidable in PSPACE in the size of the initial global data store.*

Proof 4 (Proof Sketch) The proof is done in three steps: *(1)* we provide a behavior-preserving encoding of RAW-SYSs into DCDSs; *(2)* we argue that if the input RAW-SYS is bounded, then the DCDS obtained via the encoding is state-bounded in the sense of (Bagheri Hariri et al. 2013); *(3)* we formulate reachability over the input RAW-SYS as a verification problem over the corresponding DCDS - decidability is then obtained by (Bagheri Hariri et al. 2013), in which verification over state-bounded DCDSs has been shown to be decidable. The main guideline for the encoding is as follows. The DCDS data component is obtained by combining the global data store together with the local data stores. All such relations are augmented with an additional attribute that explicitly accounts for the "provenance" of each tuple, that is, the case id of the process that generated it (through actions). In addition, an accessory relation is added so as to track the control-flow state of each such instance. The dynamic component is obtained by introducing dedicated actions for the creation/dismissal of cases. In addition, each RAW-net is translated into a set of dedicated actions (one per transition in the net), following the same strategy as in (Hariri et al. 2014). The fact that the obtained DCDS is state-bounded, which is essential for decidability, derives from the size-boundedness of the input RAW-SYS, and from the fact that we focus on bounded Petri nets. As for the complexity, in general verification over state-bounded DCDSs requires to explore, in the worst-case, a number of states that is exponential in the size of the initial database. However, in the case of reachability, this exploration can be done on-the-fly, using space that is polynomial in the size of the initial data store.

Thanks to the reduction to DCDS, we also get a stronger result: RAW-SYSs can be model checked against sophisticated temporal properties expressed in a first-order variant of μ-calculus called $\mu\mathcal{L}_p$ (Bagheri Hariri et al. 2013).

Corollary 1 *Verification of $\mu\mathcal{L}_p$ properties over size-bounded RAW-SYSs is decidable*

We close this section by briefly discussing why size-bounded RAW-SYS are reasonable in practice. Bounding the number of simultaneously running instances can be seen as a sort of "limited resources" assumption: recalling our running example, the `GoodsKit` travel office has only n people taking care of reimbursement procedures, and hence, assuming each person to handle at most c cases in parallel, only $m = n \cdot c$ $\mathbb{RB}$ cases can be running at the same time. Still, un unbounded number of different $\mathbb{RB}$ cases can be executed during `GoodsKit` life. Bounding the size of the local and global data stores reflects the fact that the progression of cases depends only on a limited, i.e., not unbounded, amount of data. As for local data stores, size-boundedness immediately applies to all practical frameworks that rely on "local variables" as case data. Concerning the global data store, *it can still be unbounded in practice*: we only require a bound on the part used to decide about the progression of currently running cases. Therefore we can easily imagine an unbounded "archival memory" that is used only for other forms of analysis, such as reporting, auditing, and mining. The unboundedness of the archival memory does not undermine our decidability results as, from the verification perspective, it can be seen as a "write-only" component.

6 Implementing Verification using Planning

To support automated verification of RAW-SYS models from a practical point of view we encode the model and verification problem using the $\mathcal{C}$ action language (Lifschitz 1999; Gelfond and Lifschitz 1998). We selected this language because it enables a simple and clear encoding of both the dynamic and static (i.e., data constraints) aspects of RAW-SYS; moreover there are different state of the art systems implementing it (e.g. the $\texttt{DLV}^{\mathcal{K}}$ planner (Eiter et al. 2003)). However, the same encoding can be easily adapted to different planners and similarly expressive planning representation languages, e.g., ADL (Calvanese et al. 2016). For the sake of space constraints we will sketch the main ideas of the encoding.

States are represented by means of so called *fluents* and the dynamics of the planning domain is specified by means of *actions* and *rules*. Actions may have preconditions, and rules, defining how fluents change, may mention them. This allow rules to be used as action postconditions.

The data component is encoded using a fluent for every relation name, and the initial data instance is the planner initial state. Integrity constraints of the data model are encoded by means of denial constraints; that is, rules with $false$ in the head. To ensure separation between different processes, each relation includes an additional attribute holding the process id and queries and constraints are modified in such a way that processes can only access local data, i.e., tuples marked with the corresponding process id. Processes control-flow, defined by means of workflow nets, is encoded using an additional fluent (mrk) representing the *marking* (similar to what done in (Di Francescomarino et al. 2015)).

A case transition associated to a guard/action pair are encoded with (*i*) a $\mathcal{C}$ action having the guard and the specific

marking as precondition and (*ii*) rules asserting straight facts if added by the effects and asserting negated facts if deleted by the effects (*iii*) rules taking care of updating the mrk fluent.

Functional terms are not directly available in most of the implementations of $\mathcal{C}$-based action languages, therefore we simulate them by means of predicates and denial constraints enforcing functionality. Moreover, nondeterministic selection of the value is forced by means of default negation through a commonly used ASP pattern of rules. Using the results outlined in § 5 we can restrict to a domain of constants for the planning problem including those appearing in the initial instance, actions, and the (finite) set of constants obtained by abstracting the infinite elements from the original domain. The finiteness of the domain is guaranteed by the fact that the model is state-bounded.

Soundness and completeness of such an encoding w.r.t. the reachability problem are inspired by the results presented in (Calvanese et al. 2016). In particular, we establish a correspondence between state transitions in RAW-SYS and states of the action language specification.

The verification of reachability properties of the model is encoded with a query describing the goal state for the planner. These properties can be related to the state of the (global) data store as well as more general conditions over the dynamic of the system, e.g., the verification that there are no running processes can be performed by checking that no places are in the mrk relation (i.e. there are no tokens).

An experiment with a concrete planner. As a proof of the feasibility of the proposed approach we investigated the applicability of RAW-SYS with a concrete reasoning system ($\mathtt{DLV}^{\mathcal{K}}$) in the realistic setting of the $\mathbb{RB}$ example.

Specifically, we tested the encoding of $\mathbb{RB}$ on different scenarios in which we varied the following two dimensions: *(i)* the size of the set of values of the active domain of employees and destinations ($\mathbf{ADOM}$); specifically, $\mathbf{ADOM} \in \{5, 10, 20\}$ *(ii)* the number of pending requests ($\mathbf{PR}$); specifically $\mathbf{PR} \in \{2, 5, 10\}$. Considering the range of these parameters, we obtain a total of 8 scenarios per query. We focus on the following two existentially quantified queries (i.e., goals in the planning encoding):

Query	ADOM	PR	Time(ms)
Q1	5	2	482.2
		5	499.2
	10	2	3278.8
		5	3950.4
		10	3840.3
	20	2	41660.1
		5	64515.9
		10	50019.9
Q2	5	2	982.2
		5	9805.8
	10	2	7143.2
		5	7577.3
		10	62415.5
	20	2	90709.1
		5	98616.2
		10	116554.4

Table 3: Performance results

Q1. a pending request has been processed;
Q2. a pending request has been accepted;

The experimentation has been performed on a pc running Windows 8 with 8GB RAM and a 2.4 GHZ Intel-core i7 and the results averaged on 10 iterations. Table 3 shows the performance of RAW-SYS on the $\mathbb{RB}$ example: the time required by RAW-SYS for achieving the goal ranges from about 0.5s (482.2 ms) for **Q1** with very small active domains, to about a couple of minutes (116554.4 ms) for **Q2** with a size of active domains (ADOM) and pending requests (PR) which is realistic for a small enterprise. An execution time of two minutes looks reasonable for a verification system dealing with complex data that has not been optimized, thus giving the glimpse of the applicability of the approach to realistic scenarios. Moreover the required time seems to be not particularly affected by the number of pending requests (PR), while it seems to strongly depend on the size of ADOM.

7 Related Work and Concluding Remarks

On the theoretical side, a number of works exist both in the area of data-aware processes and of (variants of) PNs. Unfortunately, when combining processes and data, verification problems suddenly become undecidable (Calvanese, De Giacomo, and Montali 2013). We can divide this literature in two streams. In the first stream, variants of PNs are enriched, by making tokens able to carry various forms of data, and by making transitions aware of such data, such in CPNs (van der Aalst and Stahl 2011) or data variants such as (Structured) Data Nets (Badouel, Hélouët, and Morvan 2015; Lazić et al. 2007), ν-PNs (Rosa-Velardo and de Frutos-Escrig 2011) and Conceptual WF-nets with data (Sidorova, Stahl, and Trcka 2011). For full CPNs, reachability is undecidable and usually obtained by imposing finiteness of color domains. Data variants instead weaken data-related aspects. Specifically Data Nets and ν-PNs consider data as unary relations, while semistructured data tokens are limited to tree-shaped data structures. Also, for these models coverability is decidable, but reachability is not. The work in (Sidorova, Stahl, and Trcka 2011) considers data elements (e.g., *Price*) that can be used on transitions' preconditions. However, reasoning does not consider data values (e.g., 50$) but only whether the value is "defined" or "undefined". The second stream contains proposals that take a different approach: instead of making the control-flow model increasingly data-aware, they consider standard data models and make them increasingly "dynamics-aware". Notable examples are relational transducers (Abiteboul et al. 2000), active XML (Abiteboul, Segoufin, and Vianu 2009), the artifact-centric paradigm (Gerede, Bhattacharya, and Su 2007; Damaggio, Deutsch, and Vianu 2011; Bagheri Hariri et al. 2013), and DCDSs (Bagheri Hariri et al. 2013). Such works differ on the limitations imposed to achieve decidability, but they all lack an intuitive control-flow perspective. RAW-SYS instead directly combines a control-flow model based on PNs and standard data models (à la DCDS) as first class citizens.

As future work, we plan to set up an extensive experimental evaluation optimizing our $\mathtt{DLV}^{\mathcal{K}}$ encoding, as well as to tackle verification properties beyond reachability, allowed by the result of DCDS state boundedness, through an encoding in state-of-the-art model checkers.

Acknowledgement. This research has partially been carried out within the Euregio IPN12 KAOS, which is funded by the "European Region Tyrol-South Tyrol-Trentino" (EGTC) under the first call for basic research projects.

References

Abiteboul, S.; Vianu, V.; Fordham, B. S.; and Yesha, Y. 2000. Relational transducers for electronic commerce. *Journal of Computer and System Sciences* 61(2):236–269.

Abiteboul, S.; Segoufin, L.; and Vianu, V. 2009. Modeling and verifying Active XML artifacts. *Bull. of the IEEE Computer Society Technical Committee on Data Engineering* 32(3):10–15.

Badouel, E.; Hélouët, L.; and Morvan, C. 2015. Petri nets with semi-structured data. In *Proc. of 36th International Conference on Application and Theory of Petri Nets and Concurrency*.

Bagheri Hariri, B.; Calvanese, D.; De Giacomo, G.; Deutsch, A.; and Montali, M. 2013. Verification of relational data-centric dynamic systems with external services. In *32nd Symposium on Principles of Database Systems (PODS '13)*, 163–174. ACM.

Belardinelli, F.; Lomuscio, A.; and Patrizi, F. 2012. An abstraction technique for the verification of artifact-centric systems. In *Proc. of 13th Int. Conf. on Principles of Knowledge Representation and Reasoning, KR 2012*, 319–328.

Calvanese, D.; Montali, M.; Patrizi, F.; and Stawowy, M. 2016. Plan synthesis for knowledge and action bases. In *Proc. of the 25th Int. Joint Conf. on Artificial Intelligence (IJCAI 2016)*. AAAI Press.

Calvanese, D.; De Giacomo, G.; and Montali, M. 2013. Foundations of data-aware process analysis: A database theory perspective. In *32nd Symposium on Principles of Database Systems (PODS '13)*, 1–12. ACM.

Damaggio, E.; Deutsch, A.; and Vianu, V. 2011. Artifact systems with data dependencies and arithmetic. In *Proc. of the 14th Int. Conf. on Database Theory (ICDT 2011)*, 66–77.

De Giacomo, G.; Lesperance, Y.; and Patrizi, F. 2012. Bounded situation calculus action theories and decidable verification. In *13th Int. Conf. on Principles of Knowledge Representation and Reasoning, KR 2012*, 467–477.

Di Francescomarino, C.; Ghidini, C.; Tessaris, S.; and Sandoval, I. V. 2015. Completing workflow traces using action languages. In *Proc. of 27th Int. Conf. on Advanced Information Systems Engineering (CAiSE 2015)*, volume 9097 of *LNAI*, 314–330. Springer.

Dijkman, R. M.; Dumas, M.; and Ouyang, C. 2008. Semantics and analysis of business process models in bpmn. *Information and Software Technology* 50(12):1281–1294.

Eiter, T.; Faber, W.; Leone, N.; Pfeifer, G.; and Polleres, A. 2003. A logic programming approach to knowledge-state planning, II: The DLVK system. *Artificial Intelligence* 144(1-2):157–211.

Gelfond, M., and Lifschitz, V. 1998. Action Languages. *Electronic Transactions on AI* 2(3-4):193–210.

Gerede, C. E.; Bhattacharya, K.; and Su, J. 2007. Static analysis of business artifact-centric operational models. In *IEEE Int. Conf. on Service-Oriented Computing and Applications, SOCA 2007*, 133–140. IEEE Computer Society.

Hariri, B. B.; Calvanese, D.; Montali, M.; and Deutsch, A. 2014. State-boundedness in data-aware dynamic systems. In *Proc. of 14th Int. Conf. on the Principles of Knowledge Representation and Reasoning, KR 2014*. AAAI Press.

Hull, R., and Motahari Nezhad, H. R. 2016. Rethinking BPM in a cognitive world: Transforming how we learn and perform business processes. In *14th Int. Conf. on Business Process Management, BPM 2016*, volume 9850 of *LNCS*, 3–19. Springer.

Hull, R. 2008. Artifact-centric business process models: Brief survey of research results and challenges. In *Proceedings of the OTM 2008 Confederated International Conferences*, volume 5332 of *LNCS*, 1152–1163. Springer.

Kappel, G.; Kapsammer, E.; and Retschitzegger, W. 2004. Integrating xml and relational database systems. *World Wide Web* 7(4):343–384.

Kiepuszewski, B.; ter Hofstede, A. H. M.; and Bussler, C. J. 2013. On structured workflow modelling. In *12th Int. Conf on Advanced Information Systems Engineering (CAiSE 2000)*, volume 1789 of *LNCS*. Springer. 431–445.

Lazić, R.; Newcomb, T.; Ouaknine, J.; Roscoe, A. W.; and Worrell, J. 2007. Nets with Tokens Which Carry Data. In *28th Int. Conf. on Applications and Theory of Petri Nets and Other Models of Concurrency, (ICATPN 2007)*, volume 4546 of *LNCS*, 301–320. Springer.

Lifschitz, V. 1999. Action languages, answer sets and planning. In *The Logic Programming Paradigm: a 25-Year Perspective*. Springer Verlag. 357–373.

Meyer, A.; Smirnov, S.; and Weske, M. 2011. Data in business processes. Technical Report 50, Hasso-Plattner-Institut for IT Systems Engineering, Universität Potsdam.

Minsky, M. L. 1967. *Computation: Finite and Infinite Machines*. Prentice-Hall, Inc.

Montali, M., and Calvanese, D. 2016. Soundness of data-aware, case-centric processes. *International Journal on Software Tools for Technology Transfer* 18(5):535–558.

Reichert, M. 2012. Process and data: Two sides of the same coin? In *Proceedings of the OTM 2012 Confederated International Conferences*, volume 7565 of *LNCS*, 2–19. Springer.

Rosa-Velardo, F., and de Frutos-Escrig, D. 2011. Decidability and complexity of petri nets with unordered data. *Theoretical Computer Science* 412(34):4439 – 4451.

Sidorova, N.; Stahl, C.; and Trcka, N. 2011. Soundness verification for conceptual workflow nets with data: Early detection of errors with the most precision possible. *Information Systems* 36(7):1026–1043.

ter Hofstede, A. H. M.; van der Aalst, W. M. P.; Adams, M.; and Russell, N., eds. 2010. *Modern Business Process Automation - YAWL and its Support Environment*. Springer.

van der Aalst, W., and Stahl, C. 2011. *Modeling Business Processes: A Petri Net-Oriented Approach*. MIT Press.

Van Der Aalst, W. M. P. 1998. The application of petri nets to workflow management. *Journal of Circuits, Systems and Computers* 08:21–66.