Completing Workflow Traces Using Action Languages

Chiara Di Francescomarino[1(✉)], Chiara Ghidini[1(✉)], Sergio Tessaris[2], and Itzel Vázquez Sandoval[2]

[1] FBK-IRST, Via Sommarive 18, 38050 Trento, Italy
{dfmchiara,ghidini}@fbk.eu
[2] Free University of Bozen–Bolzano, piazza Università, 1, 39100 Bozen-Bolzano, Italy
tessaris@inf.unibz.it,
itzel.vazquezsandoval@stud-inf.unibz.it

Abstract. The capability to monitor process and service executions, which has gone to notably increase in the last decades due to the growing adoption of IT-systems, has brought to the diffusion of several reasoning-based tools for the analysis of process executions. Nevertheless, in many real cases, the different degrees of abstraction of models and IT-data, the lack of IT-support on all the steps of the model, as well as information hiding, result in process execution data conveying only incomplete information concerning the process-level activities. This may hamper the capability to analyse and reason about process executions. This paper presents a novel approach to recover missing information about process executions, relying on a reformulation in terms of a planning problem.

Keywords: Business processes · Planning · Execution logs

1 Introduction

In the last decades, the use of IT systems for supporting business activities has notably increased, thus opening to the possibility of monitoring business processes and performing on top of them a number of useful analysis. This has brought to a large diffusion of tools that offer business analysts the possibility to observe the current process execution, identify deviations from the model, perform individual and aggregated analysis on current and past executions, thus supporting process model re-design and improvement.

Unfortunately, a number of difficulties may arise when exploiting information system data for monitoring and analysis purposes. Among these, data may bring only partial information in terms of which process activities have been executed and what data or artefacts they produced, due to e.g., manual activities that are scarcely monitorable and hence not present in within the information system data (*non-observable activities*).

© Springer International Publishing Switzerland 2015
J. Zdravkovic et al. (Eds.): CAiSE 2015, LNCS 9097, pp. 314–330, 2015.
DOI: 10.1007/978-3-319-19069-3_20

To the best of our knowledge, none of the current approaches has tackled the latter problem. Only recently, the problem of dealing with incomplete information about process executions has been faced by few works [1,2]. However, either the proposed approach relies on statistical models, as in [1] or it relies on a specific encoding of a particular business process language, with limited expressiveness (e.g., it cannot deal with cycles), as in [2].

In this paper we tackle the problem of reconstructing information about incomplete business process execution traces, proposing an approach that, leveraging on the model, aims at recovering missing information about process executions using action languages. In order to address the problem we exploit the similarity between processes and automated planning [3], where activities in a process correspond to actions in planning. A (complete) process execution corresponds to a sequence of activities which, starting from the initial condition, leads to the output condition satisfying the constraints imposed by the workflow. Analogously, a total plan is a sequence of actions which, starting from the initial state, leads to the specified goal.

Given a workflow and an observed, incomplete, trace, we provide an algorithm to construct a planning problem s.t. each solution corresponds to a complete process execution and vice versa. In this way, by analysing all the possible plans we can infer properties of the original workflow specification (e.g. the number of valid cases, unused branches, etc.), including all the possible completions of the trace. The advantage of using automated planning techniques is that we can exploit the underlying logic language to ensure that generated plans conform to the observed traces without resorting to an ad hoc algorithm for the specific completion problem. In the literature different languages have been proposed to represent planning problems and in our work we use the language $\mathcal{K}$ based on the Answer Set Programming engine DLV$^{\mathcal{K}}$ (see [4]). This language, in the spirit of the well known $\mathcal{C}$ (see [5]), is expressive enough for our purposes and the integration within an ASP system enables a flexible and concise representation of the problem. On the other hand, the main ideas behind the encoding are general enough to be adapted to most of the expressive planning languages.

We focus on *block structured* workflows which, broadly speaking, means that they are composed of blocks, where every split has a corresponding join, matching its type, and of loops with a single entry and exit points [6]. This assumption rules out pathological patterns that are notoriously hard to characterise (e.g. involving nested OR joins); but they provide coverage for a wide range of interesting use cases [7].

2 A Motivating Example

We aim at understanding how to reconstruct information of incomplete process execution traces, given the knowledge about the process model (which we assume to be correct and complete). The input to our problem consists of: (i) an instance-independent component, the process model, which in this paper is described

using the YAWL language[1] [6]; and (ii) an instance-specific component, that is, the input trace.

Hereafter we assume familiarity with YAWL, a workflow language inspired by Petri Nets, whose main constructs are reported in Figure 1.

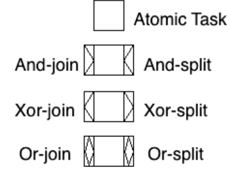


Fig. 1. YAWL

As a simple explanatory example of the problem we want to solve, consider the YAWL process in Figure 2. The process takes inspiration from the procedure for the generation of the Italian fiscal code: the registration module is created (CRM), the personal details of the subject are added (APD) and, before assigning the fiscal code, either the passport/stay permit information ($APPD$) or the parents' data ($APARD$) are added to the module. In the latter case, according to whether the child is foreigner ($foreigner$) or not ($!foreigner$), either the nationality code or the birth APSS code (AAC) is added to the module. Once data have been added ($APDC$), the fiscal code is generated (GFC) and checked (CFC). If the fiscal code is correct ($FCOk$), administrative offices (NA) and users (NU) can be notified (in parallel), otherwise ($!FCOk$), the error reported (RE) and the fiscal code generation procedure iterated until successful. Specifically, for the user notification, according to whether the request is marked as a request for a child or not, either the parents (NP) or the requester (NR) are notified. After the notification the request module is finally registered (RRM).

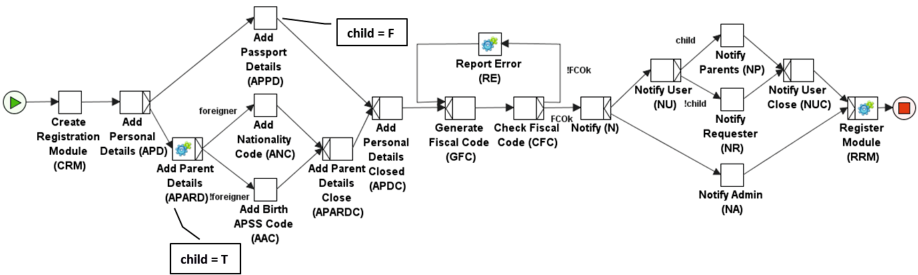


Fig. 2. A process for the generation of the Italian fiscal code

We assume that a run-time monitoring system is able to trace the execution of this process, by logging only the *observable activities* $APARD$, RE and RRM, marked in Fig. 2 with a small gears icon and the data observed by the system. An example of such a logged information (partial trace) is reported in (1). It lists 4 executions of observable activities and the corresponding observed data (enclosed in curly brackets):

$$\boxed{APARD\ \{foreigner : T\},\quad RE,\quad RE,\quad RRM} \tag{1}$$

[1] We use the YAWL modeling language but the approach can be extended to any other block-structured language.

Exploiting the available knowledge about the process model and the observed trace, we would like to know whether it is possible (and how) to reconstruct the complete trace. For instance, by knowing the process control flow in Fig. 2 and the fact that RRM was executed, we can infer that the workflow has been executed from the start until RRM. Thus, taking into account the YAWL semantics, this means that: (i) all the sequential and "parallel" activities CRM, APD, $APDC$, GFC, CFC, N, NU, NUC and NA have been executed; and (ii) exactly one among the mutually exclusive activities (a) $APPD$ or $APARD$ and $APARDC$, (b) ANC or AAC, and (c) NP or NR, have been executed. Moreover, by knowing from (1) that RE has been executed twice, it is possible to understand that the cycle has been iterated two times (and hence GFC and CFC have been executed three times). Similarly, by observing that $APARD$ has been executed, it is possible to understand that the execution also passed through $APARDC$ and not through $APPD$. As a result a possible extension of the trace in (1) is:

$$\boxed{\begin{array}{l} CRM, \quad APD, \quad APARD\ \{foreigner\ :\ T\}, \quad APARDC, \quad APDC, \\ GFC, \quad CFC, RE, \quad GFC, \quad CFC, \quad RE, \quad GFC, \quad CFC, \quad N, \quad NU, \\ NUC, \quad NA \quad RRM \end{array}} \tag{2}$$

However, at this point we are not able to completely reconstruct the trace as we cannot understand which of the alternatives among ANC or AAD, and NP or NR, have been executed. Data, both used for enriching the model and observed in the trace, provides a further source of useful knowledge which can help to discriminate about the missing activities. For example, by observing in the trace the value of the variable $foreigner$ just after the branching activity $APARD$, it is possible to understand that the branch executed by the considered execution trace is the one passing through ANC. Finally, it could happen that some further knowledge is available about data in a workflow, e.g., what are the activities in charge of manipulating those data. For instance, in this example, we could have further knowledge about the variable *child*: we could be aware that *child* is a field of the registration module that is only set by the activity $APARD$ (to *true*) and $APPD$ (to $false$)[2]. This knowledge makes it possible to understand that the "*child*" branch, i.e., the branch passing through the parent notification (NP) should have been executed, thus reconstructing a complete trace, e.g.,

$$\boxed{\begin{array}{l} CRM, \quad APD, \quad APARD\ \{foreigner\ :\ T\}, \quad ANC, APARDC, \quad APDC, \\ GFC, \quad CFC, RE, \quad GFC, \quad CFC, \quad RE, \quad GFC, \quad CFC, \quad N, \quad NU, \quad NP \\ NUC, \quad NA \quad RRM \end{array}} \tag{3}$$

Although in this simple example understanding how to fill "gaps" in the incomplete trace is relatively easy, this is not the case for real world examples. In the next sections we show how to encode general problems in order to be able to automatically reconstruct a partial trace (if the incomplete information

[2] Note that the YAWL model in Figure 2 has been annotated with this additional information.

of the partial trace is compliant) or alternatively, to assess the non-compliance of the incomplete information of the partial trace). The output that we expect is hence either (a) the notification that the partial trace is inconsistent with the process model, or (b) a set of traces that complete the input partial trace (partially or in full).

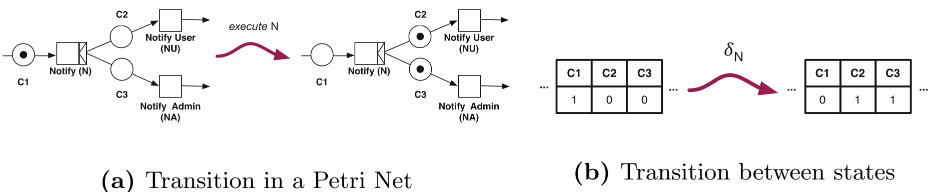
3 The General Approach

IA (complete) process execution can be seen as a sequence of activities which, starting from the initial condition, leads to the output condition satisfying the constraints imposed by the workflow. Similarly, a total plan is a sequence of actions which, starting from the initial state, leads to the achievement of a specified goal.

By exploiting this similarity, given a workflow (that we assume to be correct and complete, as its actions) and an observed trace, we provide an algorithm to construct a planning problem such that each solution of the planning problem corresponds to a complete process execution and vice versa. In this way, we can (i) either assess the non-compliance of the incomplete information of the partial trace w.r.t. the workflow specification (if no compliant plan is found) or (ii) by analysing all the possible plans we can infer properties of the original workflow specification.

Our encoding includes two stages: firstly the given workflow is encoded into an equivalent planning problem; then further constraints are added to the generated problem to ensure that the only admissible plans are those conforming to the observed traces.

The key of the bisimulation of the workflow processes using an action language lays in the fact that the semantics of YAWL is provided in terms of petri nets transition systems, where states are defined in terms of conditions connected to activities. Conditions may contain one or more tokens and the execution of activities causes transition between states by moving tokens from incoming to outgoing conditions according to their type (AND/OR/XOR join or split); e.g. in Figure 3a the execution of activity N, with an AND split, moves tokens from the input condition to both the output conditions. In a nutshell, the general idea of the bisimulation is to represent the position of tokens by means of states and execution of activities by (possibly non-deterministic) transitions between states (see Figure 3b).



(a) Transition in a Petri Net **(b)** Transition between states

Fig. 3. Encoding YAWL into a plan: an intuitive representation

Block structured workflows are *safe* in the sense that no more than one token accumulates in a single condition; therefore we do not need to keep track of the number of tokens in each condition but we just need to track the presence of a token by means of an appropriate mechanism (that of propositional fluents introduced below).

The execution of a workflow is encoded by using the main elements of an action language, that is *fluents* and *actions*. The formers represent the state of the system which may change by means of actions. Causality statements describe the possible evolution of the states and preconditions associated to actions describe which action can be executed according to the present state. The conditions in the workflow are represented by means of fluents and appropriate causality statements describe the transition by "simulating" the semantics of activities. The possibility of representing partial knowledge and non-determinism in $\mathcal{K}$, introduced in the next section, enables the precise modelling of complex workflow structures like OR-splits and loops.

The conformance to observed traces is enforced by means of additional fluents which, together with causality and pre-condition statements involving observable activities, rule out unwanted plans.

4 Encoding the Problem Using $\mathcal{K}$

We introduce our encoding in two different steps; firstly we provide a general algorithm that, given a block structured workflow, generates a planning problem that *bisimulates* the valid cases. Secondly, we show that given an observed trace we can modify the planning problem in order to exclude plans that are not conforming to the observations. Later we show that additional information about data used by the process can be easily incorporated into the framework, providing additional insight into the observed processes.

For lack of space, in this paper we introduce the main idea behind our technique. For the complete encoding and formal proofs the reader is referred to [8].

4.1 Overview of Action Language $\mathcal{K}$

A planning problem in $\mathcal{K}$ is specified using a Datalog-like language where fluents and actions are represented by literals (not necessarily ground). A problem specification includes the list of fluents, actions, initial state and goal conditions; moreover a set of statements specifies the dynamics of the planning domain using causation rules and executability conditions. The semantics of $\mathcal{K}$ borrows heavily from ASP paradigm. In fact, the system enables the reasoning with partial knowledge and provides both weak and strong negation.

A *causation rule* is a statement of the form

caused f **if** $b_1, \ldots, b_k$, **not** $b_{k+1}, \ldots,$ **not** b_ℓ **after** $a_1, \ldots, a_m$, **not** $a_{m+1}, \ldots,$ **not** a_n.

where f is either a classical literal over a fluent or `false` (representing absurdity), the b_i's are classical literals (atoms or strongly negated atoms, indicated using `-`)

over fluents and background predicates and the a_j's are positive action atoms or classical literals over fluent and background predicates. Informally, the rule states that f is true in the new state reached by executing (simultaneously) some actions, provided that $a_1, \ldots, a_m$ are known to hold while $a_{m+1}, \ldots, a_n$ are not known to hold in the previous state (some of the a_j might be actions executed on it), and $b_1, \ldots, b_k$ are known to hold while $b_{k+1}, \ldots, b_\ell$ are not known to hold in the new state.

An *executability condition* is a statement of the form

executable a **if** $b_1, \ldots, b_k$, **not** b_{k+1}, ..., **not** b_ℓ.

where a is an action atom and $b_1, \ldots, b_\ell$ are classical literals (known as preconditions in the statement). Informally, such a condition says that the action is eligible for execution in a state, if $b_1, \ldots, b_k$ are known to hold while $b_{k+1}, \ldots, b_\ell$ are not known to hold in that state.

Terms in both kind of statements could include variables (starting with capital letter) and the statements must be safe in the usual Datalog meaning w.r.t. the first fluent or action of the statements. Additionally, $\mathcal{K}$ provides some macros to express commonly used patterns. These are internally expanded using the above two statements together with strong and weak negation. For example a fluent can be declared `inertial`, expressing the fact that its truth value does not change unless explicitly modified by an action, or it could be stated that after an action there should be total knowledge concerning a given fluent. For more details the reader should refer to [4].

4.2 Encoding of the Workflow

The main elements of a YAWL workflow are activities and conditions: the latter represent the current status by means of those where tokens are present, while the former "activate" according to the state of input conditions and move tokens in output conditions. To each activity X in the workflow is associated an action with the identifier x in the plan, moreover each condition is associated to a unique identifier. In our example this unique identifier is represented by the concatenation of the connected actions. For example, nu_np represents the condition connecting the activities NU and NP in the workflow in Fig. 2. States are represented by the inertial fluent enabled(·) ranging over the set of conditions in the workflow. The fact of the fluent being true corresponds to the presence of a token in the corresponding condition. In this way we can establish a one to one correspondence between planning and workflow states. In the examples below implicit conditions are named using the starting and ending activities. Workflow contains two special conditions called start and end respectively. These are encoded as the initial state and goal specification:

initially: enabled(start).
goal: enabled(end)?

The encoding is based on the activities of the workflow: each activity with input and output conditions translates to a set of $\mathcal{K}$ statements in a modular fashion. The kind of join determines the executability of the corresponding

action according to the input conditions over the enabled(·) fluents. Values of the enabled(·) fluents associated to the output conditions are manipulated according to the kind of split by means of a set of causation rules.

Fig. 2 introduces two kinds of patterns: AND and XOR split/join representing parallelism and decision respectively. Parallelism makes sure that all the alternative branches are processed by activating all the output conditions and waiting for all the input conditions before enabling the closing activity:

caused enabled(n_nu) **after** n.
caused enabled(n_na) **after** n.
executable rrm **if** enabled(nuc_rrm), enabled(na_rrm).

All tokens in input conditions are "consumed" by the activities and this is captured by using strong negation; e.g. for *RRM*:

caused −enabled(nuc_rrm) **after** rrm.
caused −enabled(na_rrm) **after** rrm.

Decision patterns (XOR) select *only one* condition, and the corresponding join expects just one of the input conditions to be activated:[3]

caused enabled(apd_appd) **if not** enabled(apd_apard) **after** apd.
caused enabled(apd_apard) **if not** enabled(apd_appd) **after** apd.
executable apdc **if** enabled(appd_apdc).
executable apdc **if** enabled(apardc_apdc).

All but the OR join can be characterised by *local* properties of the workflow; i.e. it is sufficient to consider input and output conditions associated to the activity. On the other hand, YAWL semantics for the OR join sanctions that the corresponding activity is executable iff there is a token in at least one of the input conditions and *no* tokens can reach empty input conditions later on. This specification is clearly non-local and requires the inspection of the status of conditions not directly connected with the action. However, the restriction to block structured workflows enables us to restrict the actual dependency only to conditions enclosed between the "opening" OR split corresponding to the join. By looking at the network we could determine which conditions might inject tokens into each one of the input conditions; therefore we can prevent the executability of the action unless there are no "active" conditions that might bring a token into any of the empty inputs. To this end we introduce the fluent delayed(·) which identifies such "waiting" conditions:

caused delayed(Y) **if not** enabled(Y), reachable(Y,W), enabled(W).

The information concerning reachability is encoded into the predicate reachable(·,·) which can be pre-computed during the encoding. Given these predicates, the encoding of the OR join for an action S with input conditions $c_1, \ldots, c_n$ correspond to an executability condition of the form:

executable S **if** enabled(c_i), **not** delayed(c_1),..., **not** delayed(c_n).

for each of the input conditions.

[3] Split predicates associated to the edges will be discussed in a later section.

4.3 Encoding of Traces

Activities are divided in observable and non-observable. Traces are sequences of observed activities and generated plans should conform to these sequences in the sense that observable activities should appear in the plan only if they are in the traces and in the exact order.

In order to generate plans in which observable activities appear in the correct order, we introduce a set of fluents (indicated as *trace* fluents) to "inhibit" observable action activation unless it is in the right sequence. The action can be executed only if its corresponding trace fluent is satisfied; moreover trace fluents are set to true in the same order as the observed trace. An additional fluent indicating the end of the trace and included among the goal guarantees that all observed activities are included in the plan.

Trace fluents are in the form observed(·,·) where the first argument is the name of the activity and the second one an integer representing the order in the sequence (observed(end,$k+1$) is the fluent indicating the end of a trace of length k). E.g. a trace $APARD$, RRM for the example corresponds to the sequence of fluents

observed(apard,1), observed(rrm,2), observed(end,3)

The additional integer argument is necessary to account for multiple activations of the same activity in the trace; e.g. if RE has been observed twice there would be a fragment observed(re,n),observed(re,$n+1$) in the sequence of trace fluents.

To ensure that observable activities are included in plans only if required, all the pre-conditions of the executability conditions for observable actions are augmented with the corresponding trace fluents:

executable rrm **if** observed(rrm,N), enabled(nuc_rrm), enabled(na,rrm).

Once an action from the trace is included in the candidate plan, the action must not be repeated and the following activity could be considered; i.e. the corresponding trace fluents should be toggled:

caused observed(rrm,2) **after** observed(apard,1), apard.
caused −observed(apard,1) **if** observed(rrm,2).

Finally the initial status and goal should be modified in order to enable the first observable action and ensure the completion of the whole trace:

initially: enabled(start), observed(apard,1).
goal: enabled(end), observed(end,N)?

With the additional constraints related to the observed trace, the planner will select only plans that conform to the observation among all the possible ones induced by the workflow specification.

4.4 Encoding Information About Data

Although data within YAWL plays a crucial role in analysing process executions, to the best of our knowledge there is no formalisation suitable for automatising reasoning with workflows (see [9]).

As specified in YAWL, the conditions in which tokens are moved after the execution of (X)OR splits depend on the evaluation of the so called *branching*

tests associated to the edges.[4] In order to provide an effective automated reasoning support for workflows manipulating data we considered common usage in use cases [10] and introduced a restricted form of data which enables the analysis of a wide range of workflows. The first restriction is that we focus on boolean variables, that is, with true or false value; and the second is that we restrict branching tests to literals; i.e. a variable or its negation.[5]

Data interacting with processes may arise in different contexts and from disparate sources; we distinguish two kind of variables according to how their value is established: *endogenous* and *exogenous*. The latter ones indicate variables whose value is determined by the environment in which the process actors interact (e.g. a query to a web service) or by events not directly represented within the workflow (e.g. an user action). For example, in Fig. 2, the variable *foreigner* is not "controlled" within the workflow but depends on the context in which the process is executed. Endogenous variables, on the contrary, are those which are completely characterised by the workflow description; i.e. for these variables we know which activities manipulate their value. For example, *CFC* in Fig. 2 sets the flag *FCOk* which signals whether code is not valid, and the value of this flag is used to control the loop in the workflow.

In our work we are not interested in capturing the whole data life cycle of processes but rather in being able to further restrict the set of execution traces to those conform to the observations about the data. E.g. the execution of a specific activity might be incompatible with a branch because of the value of a variable, and our system should be able to take this into account.

Endogenous variables. This kind of variables have a natural encoding within the action language by considering each variable a different inertial fluent which can be modified by actions. Branching tests involving these variables should be added to the causation rules defining the activation of the corresponding condition.

For example, activity *APARD* is known to set the variable *child* to true, therefore the encoding should include the causation rule "**caused** child **after** apard.". Similarly, branching depending on these variables affects the enabling of the corresponding conditions. For example, the condition connecting *NU* and *NP* depends on the truth value of *child* as well:

enabled(nu_nr) **after** nu.

Values of variables might be observed in traces, and in this case, before the advancing of the corresponding trace fluent, the value of the variable should be verified. This is encoded in the planning problem by adding the observed variable to the pre-conditions of the following trace fluent. For example, if in the trace *RRM* observes the value true for *child*, then the corresponding trace fluent should be "advanced" only if that value has been set by an action:

caused observed(end,$n+1$) **if** child **after** observed(rrm,n), rrm.

[4] In the YAWL specification they are indicated as *branching conditions*, we use an alternative term to avoid confusion with the conditions themselves.

[5] This latter restriction can be lifted, although it simplifies the discussion in this paper.

Exogenous variables The behaviour of these variables is different because the process does not control but only accesses their value. In fact, in general, is not even possible to assume that their value would be constant through the complete run of the process. In terms of planning, this means that they cannot be characterised as inertial because their value might change without an explicit causation rule (e.g. external temperature below freezing). However, knowledge about their value might be exploited in specific context and this information could arise from two different sources: traces and branching tests.

In the trace shown in Equation (1), the value of exogenous variable foreigner is observed to be TRUE by the first activity ($APARD$); therefore we know its value right after the execution of the corresponding activity. This can be encoded into the following causation rules involving the fluent:

caused foreigner **after** observed(apard,1), apard.

The addition of the trace fluent is necessary to guarantee that the value is associated to the corresponding observation and not just each time the action is included in the plan (this is relevant for loops). Additional information about the nature of these variables can be used to further refine the encoding. E.g. knowing that the value of the variable would not change during the execution of the process would enable its declaration as inertial and this knowledge could be used in other parts of the workflow.

In general, exogenous variables which are part of a branching test cannot be used to select the right branch as shown in the case of endogenous variables. The reason being the fact that their value cannot be assumed. However, from the non-deterministic selection of a specific condition by means of the search for a valid plan the current value of a variable could be induced. Consider again the variable foreigner from the example in the case that its value would not have been observed in the trace, i.e., there is no information concerning which branch has been really taken). The planner would select non-deterministically between the two fluents apard_anc and apard_aac (see Fig. 2). From the selection of the first one we can assume that the value of variable foreigner must be TRUE. Indeed in any process execution in which that branching has been selected, the value of that variable had to be TRUE. This constraint can be imposed also for the encoding in the planning problem by adding the causation rule:

caused foreigner **if** enabled(apard_anc).

5 Evaluation

The problem of finding a (optimistic) plan for a $\mathcal{K}$ program is PSPACE-complete [11] and this result dominates the complexity of our algorithm. In fact, the proposed encoding generates a $\mathcal{K}$ program whose size is polynomially bound w.r.t. the size of the input problem (workflow and trace). Despite the upper-bound complexity, we are interested in investigating whether the approach is able to reconstruct incomplete traces in real scenarios and what kind of information is worth exploiting for the encoding.

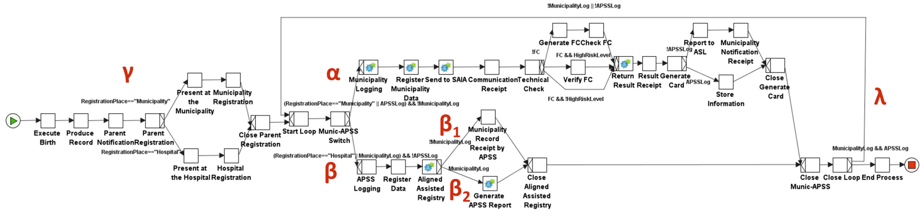


Fig. 4. Birth registration workflow

Specifically, we are interested in answering the following research questions:

RQ1 Is the ASP-based solver able to cope with the planning problems obtained encoding real scenarios?

RQ2 Is encoding information about data worth to be used by the solver to cope with the problem of reconstructing incomplete traces?

Experimental Setting. The process investigated in the experiment is the Italian procedure for the registration of births. The process, which involves several actors such as the public health service (APSS), the municipality, and the central national registry (SAIA), is reported (in the YAWL notation) in Figure 4[6]. It contains 38 activities (6 of which observable), 5 XOR blocks and 1 OR block, and it is enriched with data (specifically, 5 endogenous and 2 exogenous variables). Specifically, after that the activities devoted to prepare the procedure (γ) have been executed, the execution flow can take two alternative paths: the municipality path (path α) or the hospital one (path β), according to whether the parents decide to register the newborn first at the municipality and then the registration is passed to the hospital ($< \gamma, \alpha, \beta >$) or, first at the hospital and then the registration is passed to the municipality ($< \gamma, \beta, \alpha >$). A feedback loop ($\lambda$) allows the flow, once executed one of the two paths, to go back and execute the other one. Moreover, the β path also exhibits a mutually exclusive branch, such that, the subpath β_1 is executed if β is executed as first path, while the subpath β_2 is taken if the path β is executed as second path.

The control that both path α and β are executed exactly once, as well as the choice between the execution of β_1 or β_2 are realized through conditions imposed on data. Hence, based on the observable activities, the model enriched with data and conditions on data (DM) allows for only two possible compliant cases: $< \gamma, \alpha, \beta_2 >$ and $< \gamma, \beta_1, \alpha >$. Note that, when conditions imposed on data are not taken into account (i.e., the only control flow model M is considered), either β_1 or β_2 can be executed without any particular constraint and the feedback loop λ allows for the repetition of α and β. Although a potentially infinite number of different executions can be generated from M, in order to answer the research

[6] The figure is meant to provide an overview of the structure and size of the workflow; details are not essential in this context.

questions, we only focus on all the different incomplete traces (based on the set of the available observable activities) such that each observable activity appears at most once in the trace. The following 7 incomplete traces have been examined:

$$t_1 :< \gamma, \beta_1 >, \quad t_2 :< \gamma, \beta_2 >, \quad t_3 :< \gamma, \alpha >, \quad t_4 :< \gamma, \beta_2, \alpha >,$$
$$t_5 :< \gamma, \beta_1, \alpha >, \quad t_6 :< \gamma, \alpha, \beta_1 >, \quad t_7 :< \gamma, \alpha, \beta_2 > \tag{4}$$

Among these traces, only t_5 and t_7 are compliant with DM. Two different encodings have been investigated: (i) the one that considers only the information about the control flow (the M-based encoding) and the one relying on both control flow and data (the DM-based encoding). They contain about 35 actions and 142 (causation and executable) rules. For each incomplete trace we evaluated (both for the M and DM-encoding) the number of possible solutions (if any), as well as the time required for returning at least one solution of minimum size.

The experimentation has been performed on a pc running Windows 8 with 8GB RAM and a 2.4 GHZ Intel-core i7.

Experiment Results Table 1 reports for each incomplete trace, its size (i.e., the number of observable activities that it contains), the length of the plan (i.e., the size of the complete traces reconstructed by the planner), two metrics related to the planner exploration of the search space (i.e., the number of choice points and the recursion level), the number of alternative solutions (i.e., of the possible complete traces of minimum length) and the time required for reconstructing the missing information with and without using the information about data. Results in the table show that the planner with the DM encoding has correctly returned a complete trace only for t_5 and t_7, while it has classified the other traces as non-compliant. Indeed, the information about data, has a twofold advantage (**RQ2**):

- By constraining the execution flow through the information in the model and in the partial trace, it filters out non-compliant solutions. This also comes out by looking at the number of solutions of minimum length returned with the M and DM-encoding for t_5 and t_7: it is lowered down from 2 (c_9 and c_{13}) to 1 (c_{10} and c_{14}).
- By reducing the search space, it reduces the time required for the exploration. The time required for reconstructing the complete trace with the DM-encoding is almost a quarter of the one needed with the M-encoding.

By inspecting the time required by the planner to find a compliant plan, i.e., to find at least a possible complete trace of minimum size, results show that it depends on both the process model and the incomplete trace. For instance, the time required with the M-encoding seems to vary according to the plan length and, for plans of the same length, on the base of the type of path followed by the trace. Although the time required with the M-encoding can be high (e.g., see c_{10}), overall, the time required with the DM-encoding to find at least one possible complete trace, is of the order of a couple of minutes, which is still acceptable for a real case study (**RQ1**). Moreover, by inspecting the data, we found that for more complex (and hence time-consuming) cases, like c_{10} (which can even take

Table 1. Birth Management Procedure: statistics and time for trace completion

Check	Trace	With Data	Observable Activities	Plan Length	Choice Points	Recursion Level	Number of Plans	Completion Time
c_1	t_1	NO	1	17	1	1	2	696.0 ms
c_2	t_1	YES	1	non-compl.	-	-	-	-
c_3	t_2	NO	2	17	1	1	2	693.0 ms
c_4	t_2	YES	2	non-compl.	-	-	-	-
c_5	t_3	NO	4	22	1	1	2	1105.0 ms
c_6	t_3	YES	4	non-compl.	-	-	-	-
c_7	t_4	NO	6	31	707831	19	2	326224.0 ms
c_8	t_4	YES	6	non-compl.	-	-	-	-
c_9	t_5	NO	5	31	1050344	20	2	497429.0 ms
c_{10}	t_5	YES	5	31	385711	20	1	154157.0 ms
c_{11}	t_6	NO	5	31	189239	24	2	105574.0 ms
c_{12}	t_6	YES	5	non-compl.	-	-	-	-
c_{13}	t_7	NO	6	31	119850	22	2	84830.0 ms
c_{14}	t_7	YES	6	32	26348	22	1	18600.0 ms

10 minutes for getting a solution), the observability of a single extra activity drastically reduces the time required for the planning. For instance, observing 6 activities rather than only 5 in trace t_5 would allow us to halve the time needed for providing a solution (from 497429 ms to 244603 ms). This seems to suggest that a critical factor in terms of approach scalability is the ratio (and the type) of observable activities.

6 Related Work

The problem of incomplete traces has been faced in a number of works in the field of process mining, where it still represents one of the challenges [12]. Several works [13–15] have addressed the problem of aligning event logs and procedural models, without [13] and with [14] data, or declarative models [15]. All these works explore the search space of the set of possible moves to find the best one for aligning the log to the model. In our case, however, both goal and preconditions are different since we assume that the model is correct. Moreover, differently from [14], data are not used for weighting a cost function, by looking at their values, but rather their existence is exploited to drive the reconstruction of the complete trace.

The key role of data in the context of workflows and their interaction with the control flow has been deeply investigated by the artefact-centric approaches, in which processes are guided by the evolution of business data objects, i.e., artefacts [16]. The Guard-Stage-Milestone (GSM) approach [17] is an example of these approaches. It relies on a declarative description of the artefact life cycles, through a hierarchical structure of stages (sets of clusters of activities equipped with guards, controlling the stage activation, and milestones, determining when the stage goal is achieved). Although, similarly to these approaches, we also focus

on the interaction between data and workflows, we are not interested to the data lifecycle, but rather we aim at exploiting data in order to further restrict the set of plans compliant with the available partial observations.

The reconstruction of flows of activities of a model given a partial set of information on it can be related to several fields of research in which the dynamics of a system are perceived only to a limited extent and hence it is needed to reconstruct missing information. Most of those approaches share the common conceptual view that a model is taken as a reference to construct a set of possible model-compliant "worlds" out of a set of observations that convey limited data. We can divide the existing proposals in two groups: quantitative and qualitative approaches. The former rely on the availability of a probabilistic model of execution and knowledge. For example, in a very recent work about the reconstruction of partial execution traces [1], the authors exploit stochastic Petri nets and Bayesian Networks to recover missing information (activities and their durations). The latter stand on the idea of describing "possible outcomes" regardless of likelihood; hence, knowledge about the world will consist of equally likely "alternative worlds" given the available observations in time. Among these approaches, the same issue of reconstructing missing information has been tackled in [2] by reformulating it in terms of a Satisfiability Modulo Theory (SAT) problem. In this work, the problem is reformulated as a planning problem (specifically in the form of an action language).

Other works focused on the use of planning techniques in the context of workflows [18,19], though with a different purpose (e.g. for verifying workflow constraints, for accomplishing business process reengineering). Planning techniques have also been applied for the construction and adaptation of autonomous process models [20–22]. For example in [21] YAWL is customized with *Planlets*, YAWL nets where tasks are annotated with pre-conditions, desired effects and post-conditions, to enable automatic adaptivity of dynamic processes at runtime. The same problem is addressed using continuous planning, in [22], where workflow tasks are translated into plan actions and task states into causes and effects, constraining the action execution similarly to the approach presented here. However, to the best of our knowledge, planning approaches have not yet been applied to specifically face the problem of incomplete execution traces.

7 Conclusions

The paper aims at supporting business analysis activities by tackling the limitations due to the partiality of information often characterising the business activity monitoring. To this purpose, a novel reasoning method for reconstructing incomplete execution traces, that relies on the formulation of the issue in terms of a planning problem, is presented.

Although preliminary experiments with significantly more complex workflows than the one used in the paper show that the approach can cope with real workflows, we plan to perform an exhaustive empirical evaluation to understand whether the planner can scale up to workflows deployed in practice. Another

aspect to investigate is the different kind of data used in workflows and their interaction with the observed traces in order to discriminate relevant plans by augmenting the workflow with annotations. To this end we plan to consider the recent work on data-centric approaches to business processes (e.g. [16]) to characterise the data involved in process specification. Another line of research we have not yet considered is the analysis of the compatible completed traces. Since there could be several possible completions for an observed trace, it would be interesting to investigate how these can be aggregated and probabilistically ranked.

Acknowledgments. This work is partly funded by the European Union Seventh Framework Programme FP7-2013-NMP-ICT-FOF (RTD) under grant agreement 609190 - "Subject-Orientation for People-Centred Production".

References

1. Rogge-Solti, A., Mans, R.S., van der Aalst, W.M.P., Weske, M.: Improving documentation by repairing event logs. In: Grabis, J., Kirikova, M., Zdravkovic, J., Stirna, J. (eds.) PoEM 2013. LNBIP, vol. 165, pp. 129–144. Springer, Heidelberg (2013)
2. Bertoli, P., Di Francescomarino, C., Dragoni, M., Ghidini, C.: Reasoning-based techniques for dealing with incomplete business process execution traces. In: Baldoni, M., Baroglio, C., Boella, G., Micalizio, R. (eds.) AI*IA 2013. LNCS, vol. 8249, pp. 469–480. Springer, Heidelberg (2013)
3. Nau, D., Ghallab, M., Traverso, P.: Automated Planning: Theory & Practice. Morgan Kaufmann Publishers Inc., San Francisco (2004)
4. Eiter, T., Faber, W., Leone, N., Pfeifer, G., Polleres, A.: A logic programming approach to knowledge-state planning, ii: The dlvk system. Artificial Intelligence **144**, 157–211 (2003)
5. Lifschitz, V.: Action languages, answer sets and planning. In: The Logic Programming Paradigm: A 25-Year Perspective, pp. 357–373. Springer Verlag (1999)
6. van der Aalst, W.M.P., ter Hofstede, A.H.M.: Yawl: Yet another workflow language. Inf. Syst. **30**, 245–275 (2005)
7. Kiepuszewski, B., ter Hofstede, A.H.M., Bussler, C.J.: On structured workflow modelling. In: Seminal Contributions to Information Systems Engineering, pp. 241–255. Springer (2013)
8. Vázquez Sandoval, I.: Automated Reasoning Support for Process Models using Action Language. Master's thesis, Computer Science Faculty, Free University of Bozen-Bolzano (2014)
9. Russell, N.C.: Foundations of process-aware information systems. Thesis, Queensland University of Technology (2007)
10. Russell, N., ter Hofstede, A.H.M., Edmond, D., van der Aalst, W.M.P.: Workflow data patterns: identification, representation and tool support. In: Delcambre, L.M.L., Kop, C., Mayr, H.C., Mylopoulos, J., Pastor, Ó. (eds.) ER 2005. LNCS, vol. 3716, pp. 353–368. Springer, Heidelberg (2005)
11. Eiter, T., Faber, W., Leone, N., Pfeifer, G., Polleres, A.: A logic programming approach to knowledge-state planning: Semantics and complexity. INFSYS Research Report INFSYS RR-1843-01-11. TU Wien (2001)

12. van der Aalst, W., et al.: Process mining manifesto. In: Daniel, F., Barkaoui, K., Dustdar, S. (eds.) BPM Workshops 2011, Part I. LNBIP, vol. 99, pp. 169–194. Springer, Heidelberg (2012)
13. Adriansyah, A., van Dongen, B.F., van der Aalst, W.M.P.: Conformance checking using cost-based fitness analysis. In: Proc. of EDOC 2011, pp. 55–64 (2011)
14. de Leoni, M., van der Aalst, W.M.P., van Dongen, B.F.: Data- and resource-aware conformance checking of business processes. In: Abramowicz, W., Kriksciuniene, D., Sakalauskas, V. (eds.) BIS 2012. LNBIP, vol. 117, pp. 48–59. Springer, Heidelberg (2012)
15. de Leoni, M., Maggi, F.M., van der Aalst, W.M.P.: Aligning event logs and declarative process models for conformance checking. In: Barros, A., Gal, A., Kindler, E. (eds.) BPM 2012. LNCS, vol. 7481, pp. 82–97. Springer, Heidelberg (2012)
16. Cohn, D., Hull, R.: Business artifacts: A data-centric approach to modeling business operations and processes. Bulletin of the IEEE Computer Society Technical Committee on Data Engineering **32**, 3–9 (2009)
17. Hull, R., Damaggio, E., Fournier, F., Gupta, M., Heath III, F.T., Hobson, S., Linehan, M., Maradugu, S., Nigam, A., Sukaviriya, P., et al.: Introducing the guard-stage-milestone approach for specifying business entity lifecycles (invited talk). In: Bravetti, M. (ed.) WS-FM 2010. LNCS, vol. 6551, pp. 1–24. Springer, Heidelberg (2011)
18. Regis, G., Ricci, N., Aguirre, N.M., Maibaum, T.: Specifying and verifying declarative fluent temporal logic properties of workflows. In: Gheyi, R., Naumann, D. (eds.) SBMF 2012. LNCS, vol. 7498, pp. 147–162. Springer, Heidelberg (2012)
19. Rodríguez-Moreno, M.D., Borrajo, D., Cesta, A., Oddi, A.: Integrating planning and scheduling in workflow domains. Expert Systems with Applications **33**, 389–406 (2007)
20. da Silva, C.E., de Lemos, R.: A framework for automatic generation of processes for self-adaptive software systems. Informatica (Slovenia) **35**, 3–13 (2011)
21. Marrella, A., Russo, A., Mecella, M.: Planlets: automatically recovering dynamic processes in yawl. In: Meersman, R., et al. (eds.) OTM 2012, Part I. LNCS, vol. 7565, pp. 268–286. Springer, Heidelberg (2012)
22. Marrella, A., Mecella, M., Russo, A.: Featuring automatic adaptivity through workflow enactment and planning. In: CollaborateCom 2011 (2011)