Automated Reasoning Support for Process Models using Action Languages

Itzel Vázquez Sandoval

EMCL Master Thesis

Faculty of Computer Sciences

Supervisor: Dr. Sergio Tessaris

September 2014

To those who nourishing the soul, foster a free adventurer mind

... To G. M. C. N. and G.

Declaration

I hereby declare that except where specific reference is made to the work of others, the contents of this thesis are original and have not been submitted in whole or in part to obtain a degree or any other qualification neither in this nor in any other university. This dissertation is my own work and contains nothing which is the outcome of work done in collaboration with others, except as specified in the text.

No other resources apart from the references and auxiliary means indicated in the bibliography were used in the development of the presented work.

Itzel Vázquez Sandoval
September 2014

Acknowledgements

To all my family, ... las palabras salen sobrando.

My sincere gratitude to Sergio Tessaris for his enlightening, precise and always motivating guide during the development of the present work.

My acknowledgement to Chiara Ghidini and Chiara Di Francescomarino for the opportunity to work on this topic, to the EMCL joint commission for the creation of such an interesting and peculiar program, as well as for the granted support, fundamental for the completion of studies; and to everyone in any form involved in the elaboration process of this thesis.

Abstract

By cause of the human-interactive nature of business processes, a diversity of problems emerge from their automation, which has become an increasingly important activity and by consequence extensively effected nowadays. One of the advantages of the referred automation is the capability to monitor process and service executions, for whose analysis several reasoning-based tools have emerged. The capability to analyze and reason about such executions is weakened nevertheless by the detection of incomplete information concerning the process-level activities.

The purpose of the present thesis is to solve the particular problem of reconstructing incomplete observed executions of processes, in order to support the performance of reliable business analysis over monitored behavior. Such analysis permits among others, the enhancement of the process models.

The proposed solution consists in the characterization of the model of the process, along with the recorded trace, as a logical formulation in terms of a planning problem; this representation enables the use of well-founded automated reasoning engines to generate plans conforming to possible sequences of activities that correspond to complete instances of process executions.

Contents

Contents **xi**

1 Introduction **1**

2 Preliminaries **3**
- 2.1 Planning overview . . . 4
- 2.2 The workflow language YAWL . . . 4
  - 2.2.1 Structured workflows . . . 6
  - 2.2.2 YAWL theoretical foundations . . . 8
- 2.3 The planning language $\mathcal{K}$ . . . 10
  - 2.3.1 $\mathcal{K}$ theoretical foundations . . . 11

3 Encoding YAWL workflows into $\mathcal{K}$ programs **13**
- 3.1 $\mathcal{K}$ representation of constructs . . . 13
  - 3.1.1 Extensions . . . 17
- 3.2 Encoding algorithm . . . 18
  - 3.2.1 The background knowledge . . . 18
  - 3.2.2 The $\mathcal{K}$ specification . . . 19
  - 3.2.3 Complexity . . . 20
  - 3.2.4 Simple example . . . 20
- 3.3 Correctness of the encoding . . . 24
  - 3.3.1 Bisimulation between YAWL models and $\mathcal{K}$ programs . 25

4 Restriction of plans to observed executions **33**
- 4.1 Algorithm for inclusion of traces . . . 33
- 4.2 Validity of the algorithm . . . 35

5 Inclusion of branching predicates **41**
- 5.1 Encoding of Decision arcs . . . 42
- 5.2 Encoding behavior from the model . . . 44
- 5.3 Encoding observed evaluations . . . 45

6 Implementation **49**
- 6.1 System overview . . . 49
- 6.2 Input files format . . . 51
- 6.3 Tests and results . . . 52

6.3.1 Settings and test 53
6.3.2 Results 53

7 Related work **55**

8 Conclusions **57**

Bibliography **59**

Appendix A Model description's file format **63**

Appendix B Birth Management Process **67**

Appendix C Architecture of the implementation **75**

Chapter 1

Introduction

The widely adopted use of IT systems nowadays for supporting business activities, has brought to the development of tools and reasoning services, such as ProM, Microsoft's BAM suite, Oracle's BAM and IBM WebSphere's BAM component, among many others, that offer the possibility to observe the evolution of ongoing processes and hence to perform statistical analyzes on current and past executions (business analysis monitoring (BAM)). The obtained results enable to identify misalignments between process models and executions, as well as to discover design bottlenecks, encouraging thus the re-design and possible improvement of the models.

Certain difficulties though are subject to arise when exploiting the monitoring outcome for analysis purposes. The collected data for example, usually does not contain information about their owner process instance, being the correlation of both parts not a trivial task; the fetching of noise material together with the useful information is as well frequently occurring.

Furthermore, in many real cases the different degrees of abstraction between the definition of a process and the elements of the modeling language, the lack of IT-support for all the process activities (e.g. human interactions, which are scarcely traceable) and the concealment of specific parts of the process, result in the retrieval of incomplete information concerning the process-level activities executed and the data by them produced.

Only recently the problem of dealing with incomplete information about process executions has been faced by few works [4, 28]; the proposed approaches however rely, either on statistical models, as in [28] or on a specific encoding of a particular business process language, with limited expressivity, as in [4].

The present work focuses therefore on the last problem; the objective is

thus to completely reconstruct a business process execution, based on the information observed about it by a monitoring system. The proposed approach takes advantage of the well-defined description of a model and captures its dynamics in terms of logical rules of an action language, making possible to perform formal reasoning over them.

The underpinning idea comes from the similarity between processes and automated planning [23], where activities in a process correspond to actions in planning. Business processes are modeled by workflows; a complete process execution corresponds then to a sequence of activities which, starting from the unique initial condition, leads to the output condition of the workflow, satisfying the constraints imposed to traverse the model. Analogously, a total plan is a sequence of actions which, starting from the initial state, leads to the specified goal advancing through states according to preconditions.

Given thus a workflow and an observed sequence of executed tasks (trace), an algorithm to construct a planning problem such that each of its solutions corresponds to a complete process execution and vice versa, is provided. In this way, the analysis of all the possible plans allows to infer properties of the original workflow specification.

The main advantage of using automated planning techniques is that, by ensuring a correct statement of the planning problem, the conformance of generated plans with traces is asserted by the underlying logic language. Moreover, since it is a strong research field, many stable and reliable state-of-the-art planners are available.

Seeking to develop a solid and well-founded procedure, *structured* workflows will be assumed in this thesis. This assumption rules out patterns with notoriously hard to characterize behavior, such as nested OR joins, providing still coverage for a wide range of interesting use cases.

The document is structured as follows: some preliminary notions and the formalization of concepts founding the present work will be introduced in the next chapter; the proposed encoding of a process in terms of a planning problem is presented afterwards, together with a formal proof of the correct simulation of the workflow semantics by the encoding. Chapter 4 introduces an algorithm to generate exactly the plans representing complete executions corresponding to the data in a given trace. A formal proof of its validity is provided as well.

The handling of data interaction within a model is addressed in chapter 5. And later on, an overview of the implementation of the proposed approach as well as a running example are reported. The last chapters concern related work, conclusions and future work.

Chapter 2

Preliminaries

Given a process model and a partial execution trace of such process, the problem to confront is the reconstruction of the information missing in the trace.

Process models typically describe the flow of a process through activities (which can have associated data) and control flow constructs. Different languages and notations are available to describe them, by instance YAWL, BPMN and Petri Nets. The present work considers YAWL [33] as the modeling language.

Action languages are intended to specify state transition systems and a common use is the creation of formal models for the effects of actions on the world [15]; planning languages are in general action languages. Because of its answer set-like semantics giving it a powerful reasoning over incomplete knowledge, the planning language $\mathcal{K}$ is selected here.

The main ideas behind the encoding are however general enough to be adapted, under opportune assumptions, to other languages, by instance to the PDDL standard [20].

The similarity between workflow processes and action languages is supported by the fact that the formal semantics of YAWL is provided in terms of transition systems, where states are defined in terms of conditions that may trigger the execution of activities causing transitions between states.

To establish the context of the topic, general concepts of planning and the formal background of the selected languages will be provided in this chapter.

As an annotation, the notion of observability of tasks will be constantly referred. As observable tasks we denote tasks in a model whose execution

is always tracked; unobservable tasks on the other hand, refer to those ones that depend on factors out the control of a monitoring system and thus its execution can not be recorded. The observability of tasks is determined at design time.

2.1 Planning overview

Automated planning concerns the execution of computational techniques to choose and organize actions by anticipating their expected outcomes, aiming to achieve as best as possible a preset objective [23].

A model for a planning specification is a state-transition system, where states are sets of ground literals in the planning language and the transition between states is decided by actions based on state constraints. The model constitutes the domain of a problem.

A *planning problem* is a triple (Σ, s_0, *goal*), where Σ is the definition of the domain of the problem, s_0 is the initial state and *goal* is a set of ground literals to be satisfied. A solution (*plan*) is a sequence of ground actions that lead from s_0 to a state in Σ where all the elements in *goal* are satisfied.

A *fluent* or *flexible relation* is a predicate whose arguments are susceptible to change over the time; a state is therefore determined by a set of fluents. A *rigid relation* is a predicate not intended to vary over states, representing thus a fact that belongs to the knowledge base.

The purpose of the present work is to specify a planning problem under the assumption of incomplete knowledge, which in sum connotes that the status of fluents might be unknown.

2.2 The workflow language YAWL

YAWL (Yet Another Workflow Language) is a powerful workflow modeling language based on the well-known workflow patterns [30]. It is supported by an open-source environment that handles complex data transformations and full integration with organizational resources and external Web Services. [1]

YAWL's formal operational semantics is founded on Petri Nets[1], Workflow

[1] A Petri net is a bipartite graph where the nodes are either places (circles) or transitions (squares)

nets and the so called Reset nets, which combined create a language powerful enough to model dynamic behaviors of systems. The basic notions of places, transitions and flow relations that define a Petri net are implemented as well in YAWL [33]. In the last however, conditions (places) can be omitted, being implicitly represented by the arc connecting two tasks.

A workflow in YAWL is called a *specification*; each specification is composed by a root workflow net that may contain nested nets. The formed hierarchy obeys to best design practices, being though irrelevant from the semantic perspective. Since this thesis centers in the last, it will be assumed that a specification is composed by a one-level net, which is indeed equivalent to the expansion of all the nested tasks contained in a regular specification.

The set of so called *constructs* defines the basic elements from which any control-flow pattern supported by the language, and hence any specification, can be formed; the elements of such set are:

+ condition
+ input condition
+ AND/OR/XOR join
+ multiple instance task
+ composite multiple instance task
+ atomic task
+ output condition
+ AND/OR/XOR split
+ composite task
+ cancellation region

In view of the fact that each instance of our problem concerns a single execution of a process, the constructs referring to multiple tasks will be disregarded in the present work. The rest of them will be treated in more detail along the document, the particular behavior of each is introduced in the encoding section.

As a remark, due to the non-local behavior of the OR-join construct, derived from the requirement of synchronization depending on possible future execution paths, a broad and deep research has emerged around its semantics, for whose formal definition and posterior implementation, different approaches have been taken aiming to closely match the informal behavior.

A model in YAWL can be approached from three different angles, the control-flow, the data and the resource perspectives. The former concerns the ordering and interaction of tasks in the model. The data perspective approaches the exchange of information, distinguishing two types: the internal data transfer, which regards the exchange of data among elements of the workflow and the external data transfer, which involves communication between a process and its operating environment (users). The last perspective focuses in the organizational and work distribution model to describe the manner in which work items are distributed to users according to roles.

In the present work, our interest is concentrated in the control-flow perspective for the core of the bisimulation and in the internal data transfer from the data perspective, when referring to branching selection.

Some explained examples of processes modeled in YAWL can be found in the section devoted to the encoding.

2.2.1 Structured workflows

Since processes depict real world situations, a powerful modeling language should impose the less possible restrictions for what can be modeled, however, the complexity of reasoning over a language is directly proportional to its expressive power. Some identified and widely discussed problems come out for instance from the presence of the multi-merge pattern [30].

Aiming to provide well founded semantics to those languages, *structured* workflows were introduced, which informally are models where splits and joins are always paired by type into single-entry-single-exit blocks and loops occur only with a single-entry and a single-exit points [33].

A formal specification based on the definition of structured models presented in [18], is then as follows:

Definition 2.2.1. A ***structured model*** (*block*) with entry i and exit o is inductively defined as follows, where $\{c_1, \ldots, c_4\}$ are conditions:

1. Let t be an atomic task. Then, t preceded by c_1 and followed by c_2 is a *block*. The entry of *block* is c_1 and the exit is c_2.
2. SEQUENCE. Let X and Y be *block*s. The concatenation of X and Y, where the exit of X is the entry of Y is also a *block*. The entry of *block* is the entry of X and its exit is the exit of Y.
3. PARALLEL STRUCTURE. Let $X_1 \ldots X_n$ be *block*s, s an AND-split preceded by c_1 and j an AND-join followed by c_2. The structure with c_1 as entry, c_2 as exit, transitions between s and the entries of the X_is and transitions between the exits of the X_is and j is then a *block* as well. The entry of *block* is c_1 and the exit is c_2.
4. DECISION STRUCTURE. Let $X_1 \ldots X_n$ be *block*s, c_1 the unique condition incoming to s, c_2 the single condition following j and either (i)s an OR-split and j an OR-join or (ii)s an XOR-split and j an XOR-join. The structure with c_1 as entry, c_2 as exit, transitions between s and the entries of the X_is and transitions between the exits of the X_is and j,

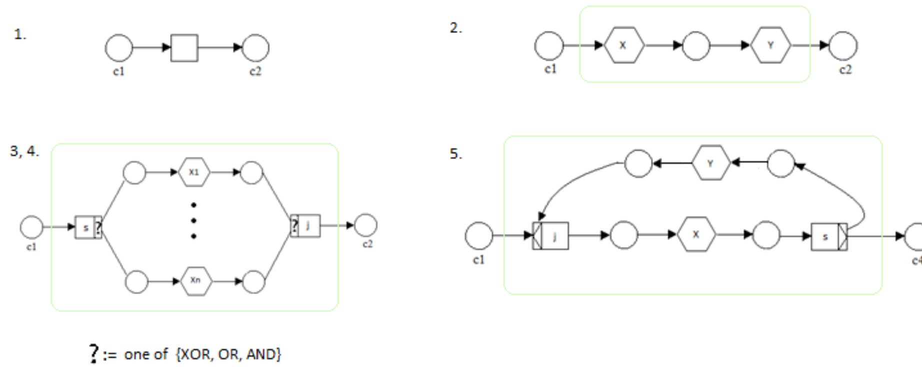


Figure 2.1 Structured model

is a *block*. Predicates can be assigned to the outgoing transitions of s. The entry of this *block* is c_1 and the exit is c_2.

5. STRUCTURED LOOP. Let X and Y be structures, j be an XOR-join preceded solely by two conditions c_1, c_2 and s an XOR-split connected with exactly two conditions c_3, c_4. Then *block* is also the structure with c_1 as entry, c_4 as exit, transitions from j to the entry of X and from the exit of X to s, and where c_3 is the entry of Y and its exit is c_2. X and Y can be null.

Figure 2.1 illustrates each case of the previous definition.

Even if not all processes can be captured as structured models, a considerable proportion of workflows can be transformed into a structured configuration, which promotes readability and makes models less error-prone [11], as well as easier to be analyzed, verified and implemented [18].

An additional number of desirable properties are present in these models, being of our particular interest *safety*, which refers to the presence of at most one instance per task at a time and *deadlock freedom*. Many more can be derived, as it will indeed be done later on in the appropriate section.

Note that the previous formalization is intended to maintain clarity and simplicity of semantics, nevertheless, in practice the conditions joining blocks are *implicit conditions* in a YAWL model and thus are not required to be modeled.

2.2.2 YAWL theoretical foundations

Two assumptions are considered in the solution approach here presented:

1. A single execution instance of the process at a time, and thus safe models
2. Process workflows represented as structured models

Considering them, the original definition of extended workflow nets from [35] is reformulated disregarding the aspects concerning multiple instances and restricting it to convey with def. 2.2.1.

Definition 2.2.2. A ***structured workflow net (SWF-net)*** (aka. SESE net) is a tuple $(C, i, o, T, Z, split, join)$ where:

- C is a set of conditions
- $i \in C$ is the input condition and there is exactly one in C
- $o \in C$ is the output condition and there is exactly one in C
- T the set of tasks in the net
- $Z \subset (C \setminus \{o\} \times T) \cup (T \times C \setminus \{i\})$ is the flow relation. [2]
- $split : T \to \{\text{AND,XOR,OR}\}$ specifies the split behavior of each task t. By default $split(t) =$AND with exactly one $(t, x) \in Z$ for a certain x.
- $join : T \to \{\text{AND,XOR,OR}\}$ specifies the join behavior of each task t. By default $join(t) =$XOR with exactly one $(x, t) \in Z$ for a certain x.
- The graph induced by $C \cup T$ ($\langle C \cup T,\ Z\rangle$) is a structured model.

A *safe workflow model* is a workflow where it is not possible that at some point in time there exist multiple instances of the same task [33]. Derived from the structured restriction, SWF-nets are safe and deadlock free ([35], [32]).

Let $N_1 = (C, init, out, T, Z, split, join)$ be a structured workflow net. The following concepts are then defined, underpinned on formalizations introduced in [35] and adapted to the previously mentioned assumptions.

[2] YAWL's syntax allows the connection from task to task in the model. Since this transition is equivalent to have an implicit condition between the two tasks, the case is covered by this definition.

Definition 2.2.3. A ***workflow state*** σ is a set of pairs such that $\forall c \in C$, $(c,\ f(c)) \in \sigma$, where the function $f : C \to \mathbb{N}$ determines the cardinality of tokens placed in the condition c at a certain moment.

Informally a state represents a multiset of tokens distributed over the conditions in the net at a certain moment in time.

Provided that in the present work safe workflow models are assumed, the elements of a workflow state take the form $(c_x, 0)$ or $(c_x, 1)$, for all $c_x \in C$; therefore, states will be hereafter represented by the conditions where $f(c_x) > 0$. For instance, $\sigma_0 = \{(c_1, 1), (c_2, 0), (c_3, 0), (c_4, 1), (c_5, 1)\}$ will be shortened as $\sigma_0 = \{c_1, c_4, c_5\}$.

Definition 2.2.4. Let $t \in T$, $pre, post, \sigma \subseteq C$ and for each element x in $C \cup T$, $preset(y) = \{x|(x, y) \in Z\}$ and $postset(x) = \{y|(x, y) \in Z\}$. There exists a ***binding(t,pre,post,σ)*** iff all of the following conditions hold:

1. $pre \subseteq \sigma$
2. $pre \subseteq preset(t)$
3. $post \subseteq postset(t)$
4. $\sigma \cap post = \emptyset$
5. According to the type of t:

$$
\begin{aligned}
split(t) = AND &\implies post = postset(t)\\
split(t) = OR &\implies post \neq \emptyset\\
split(t) = XOR &\implies |post| = 1\\
join(t) = AND &\implies pre = preset(t)\\
join(t) = OR &\implies pre \neq \emptyset\\
join(t) = XOR &\implies |pre| = 1
\end{aligned}
$$

The introduction of bindings is aimed to express that, when the appropriate incoming conditions to t (the set *pre*) own a token in the state σ, then t can be executed leading to a state where all the conditions in *post* hold a token. Notice that for the OR-join construct, the only assertion is the existence of a token in at least one incoming condition to t previous to its activation. The complete semantics are however captured in the following definition.

Definition 2.2.5. Let σ_1, σ_2 be workflow states of N_1. There is a ***transition*** from σ_1 to σ_2 iff both hold:

1. $\exists\, t \in T$ and $pre, post \subseteq C$ such that $binding(t, pre, post, \sigma_1)$ exists and $\sigma_2 = (\sigma_1 \setminus pre) \cup post$. (Not. $\sigma_1 \triangleright_t \sigma_2$)

2. If $join(t) = OR$ then, for each $\sigma \in reach(\sigma_1)$, there is no $pre' \subseteq \sigma$ s.t. there exists a $binding(t, pre', post, \sigma)$ and $pre \subset pre'$, where $reach(\sigma_1) = \{\sigma \in 2^C \mid \sigma_1 \triangleright_{r_1} \cdots \triangleright_{r_n} \sigma \text{ and } r_i \neq t,\ i \in [0, n]\}$

For notation, $\sigma_1 \xrightarrow{t} \sigma_2 \in N_1$ denotes a transition from σ_1 to σ_2 by the execution of t.

The second point in the above description affirms that it does not exist a bigger set of preconditions containing *pre*, neither in σ_1 nor in any of the states reachable by the execution of one (or more) task(s) except t itself, such that it is capable to activate t.

As an observation, some parts of the original semantics have been disregarded, such as the removal or blocking of useless tokens for instance when an XOR-join receives more than one token. Those situations are though not present in structured workflow models.

Definition 2.2.6. A ***case*** of N_1 is a finite sequence of tasks $\langle t_1, \ldots, t_n \rangle$, $t_i \in T$, for which $\exists\ \sigma_0, \ldots, \sigma_n$ workflow states, such that $\sigma_{j-1} \xrightarrow{t_j} \sigma_j$ is a transition in N_1, for all $j \in [1, n]$ and $\sigma_0 = \{init\}$. A case is ***completed*** iff $out \in \sigma_n$.

Intuitively a case depicts an instance (an execution) of the process described by N_1. A case is generated each time a token is placed in the input condition and it is constructed while tasks are being activated conforming to the semantic rules.

2.3 The planning language $\mathcal{K}$

$\mathcal{K}$ is a logic-based planning language, which thanks to its ASP-like semantics, is well suited for planning under incomplete knowledge [12].

The main elements of $\mathcal{K}$ (and in general of action languages) are *fluents* (see 2.1) and *actions*. The set of fluents depicts the state of the system in a certain moment, which can be changed by means of actions.

A characterizing feature of the language is that it allows the use of default negation besides strong negation, therefore, transitions between states are

determined according to the possessed knowledge, not being necessary to have a value for all the existent fluents. open world assumption

A planning problem specification in $\mathcal{K}$ is similar to a Datalog program; facts are represented by predicate symbols. The evolution of the states in the planning domain is ruled by a set of statements based on preconditions to constraint the execution of actions and postconditions to modify fluents after such executions.

A goal is a conjunction of ground literals and a plan for a goal is a sequence of actions whose execution leads from an initial state to a state where all the literals in the goal are satisfied. Each plan is a solution for the specified planning problem.

2.3.1 $\mathcal{K}$ theoretical foundations

The formalization of the concepts in this section is based on the description of $\mathcal{K}$ provided in [12].

Definition 2.3.1. A $\mathcal{K}$ ***program*** is a tuple $(F, A, R, init_rules, goal)$ such that:

- F is a set of fluent declarations
- A is a set of action declarations
- R is a set of rules, where each rule r is one of:

  a) Causation rule

  `caused` f `if` $b_1, \ldots, b_k,$ $not\ b_{k+1}, \ldots, not\ b_m$
  `after` $a_1, \ldots, a_p,$ $not\ a_{p+1}, \ldots, not\ a_q$

  where $f \in F \cup A \cup \{false\}$, $b_{\{1,\ldots,m\}} \in F$ and $a_{\{1,\ldots,q\}} \in F \cup A$. The if and the $after$ parts are optional; if $m = q = 0$, then the word `caused` is optional as well. Besides,

  $h(r) = \{f\}$ $\quad$ $pre(r) = \{a_1, \ldots, a_q\}$
  $post^+(r) = \{b_1, \ldots, b_k\}$ $\quad$ $post^-(r) = \{b_{k+1}, \ldots, b_m\}$

  b) Executability condition

  `executable` a `if` $b_1, \ldots, b_k,$ $not\ b_{k+1}, \ldots, not\ b_m$

  where $a \in A$, b_is $\in A \cup F$ and $m \geq k \geq 0$. Additionally,
  $h(r) = \{a\}$ $\quad$ $post(r) = \emptyset$
  $pre^+(r) = \{b_1, \ldots, b_k\}$ $\quad$ $pre^-(r) = \{b_{k+1}, \ldots, b_m\}$

not stands for the default negation in the ASP context.

- *init_rules* is the initial set of constraints, described by means of causation rules with $q = 0$
- *goal* is the set of constraints to be satisfied and described by a set of ground fluent literals possibly preceded by the default negation symbol.

$$goal: \ f_1, \ ..., \ f_m, \ not \ f_{m+1}, \ ..., \ not \ f_n?$$

A *literal* is an action, type or fluent predicate symbol $p(t_1, ..., t_n)$, possibly preceded by the strong negation symbol "-". A literal is *ground* if it does not contain variables.

Definition 2.3.2. A ***state*** is a consistent set of ground fluents, i.e., a set S of fluent literals where it holds that: if $x \in S$ then $-x \notin S$ and if $-x \in S$ then $x \notin S$.

An action $a \in A$ is *executable* with respect to a state s iff there exists an executability condition $e \in R$ (2.3.1.b) such that $h(e) = \{a\}$, $pre^+(e) \cap F \subseteq s$ and for each $x \in pre^-(e)$, it holds $x \notin s$ or $-x \in s$.

Definition 2.3.3. A ***legal transition*** is a tuple $[s1, act, s2]$, where $s1$ is a state, act is an executable action with respect to $s1$ and $s2$ is the minimal state such that for every causation rule $r \in R$ (2.3.1.a), whenever the three of $post^+(r) \cap F \subseteq s2$, $pre^+(r) \cap F \subseteq s1$ and $a \in pre^+(r) \cap A$ hold, then $h(r) \neq false \subseteq s2$.

A state s_0 is a *legal initial state* iff it is a minimal set such that for each $ic \in init_rules$, if $post^+(ic) \subseteq s_0$ then $h(ic) \in s_0$. Informally, s_0 is a state determined by a minimal set of literals needed to satisfy all the constraints in *init_rules*.

Definition 2.3.4. A sequence of actions $\langle a_1, \ldots, a_i \rangle$, $i \geq 0$ is a ***plan***, if there exists a sequence of legal transitions $T = \langle [s_0, a_1, s_1], \ \ldots, \ [s_{i-1}, a_i, s_i] \rangle$ such that s_0 is a legal initial state and for $f_i \in goal$, $\{f_1, \ldots, f_m\} \subseteq s_i$ and $\{f_{m+1}, \ldots, f_n\} \cap s_i = \emptyset$.

Definition 2.3.5. A ***partial plan*** is a sequence of actions $\langle a_1, \ldots, a_i \rangle$, $i \geq 0$, for which exists a legal transitions' sequence $T = \langle [s_0, a_1, s_1], \ \ldots, \ [s_{i-1}, a_i, s_i] \rangle$ with s_0 a legal initial state. A partial plan becomes a plan when the goal is satisfied in s_i.

For simplicity and without loss of generality, no parallelism will be assumed in this work, which is nonetheless expressible in the sequential approach, writing by instance all the concurrently executed actions one after the other.

Chapter 3

Encoding YAWL workflows into $\mathcal{K}$ programs

YAWL's control-flow perspective is the focus of attention when pursuing to retrieve missing information about traces. As mentioned before, the set of elements that define the control flow patterns supported by the language is constituted by the so called *constructs*. It is therefore that a process depicted by a YAWL model can be expressed in terms of a $\mathcal{K}$ program, through the proper encoding of each of the constructs in such model.

This section is intended to establish a precise standard representation of structured workflows as planning problems, along with the procedure to generate it.

3.1 $\mathcal{K}$ representation of constructs

Standing on the definition of YAWL's set of constructs provided in [33], as well as in its operational semantics, a case by case encoding into the planning language $\mathcal{K}$ is introduced. As previously mentioned, the proposal relies on the assumption of structured YAWL models.

In consideration of the dynamic context of workflow nets, intuitively a *token* marks a position of the flow in the net; the status of a workflow execution instance is then determined by the set of tokens. The idea underpinning the encoding is to capture the advance of the token through the workflow net by means of the `enabled` fluent and the negation of its stepped into instances.

For simplicity and w.l.o.g. workflows with a Petri Net disposition will be considered, meaning that the elements appear in sequences of task-condition-

task, i.e., no task-task connections occur. [1]

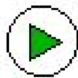
As a general remark, the `not` operator stands for default negation while the symbol `-` is used for strong negation in $\mathcal{K}$.

Condition
To convey with Petri Net semantics, whenever two or more tasks in YAWL are directly connected, the existence of an implicit condition in between is assumed. Conditions can be however explicitly modelled to represent a nondeterministic decision point for advancing the activation token (which becomes just the position of the active token when there is only one next task) in a running case of a YAWL net.

The assumption of having structured models restricts each condition to have exactly one incoming and one outgoing arc (single entry, single exit). Its existence is thus represented as a fact.

```
condition(e1).
```

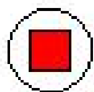
Input condition
This is the starting point of the flow, therefore, the unique condition holding a token at the beginning of the process execution.



```
initially: enabled(start).
```

Remark that this encoding goes along with its corresponding encoding for being a condition.

Output condition
It represents the last possible reachable point; the possession of the token by this condition determines the termination of the process execution.

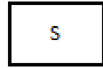


goal: enabled(end)?

Analogous to the input condition, this is joined with its condition's encoding.

[1] This topic is discussed in the "Condition" encoding part and formalized in posterior chapters.

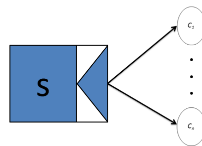
Atomic task



Enacts the default case, which assumes an XOR-join and AND-split with exactly one incoming and outgoing flow elements (arcs) respectively. The encoding is thus covered in the posterior corresponding cases.

And split

S advances a token into all of the conditions from c_1 to c_n.

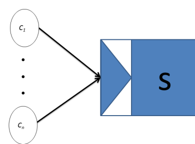



```
actions: S.

always: caused enabled(c1) after S.
        ...
        caused enabled(cn) after S.
```

And join

S is activated when all of the incoming arcs have been enabled, i.e., when there is a token in each of them.

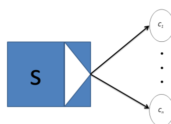



```
actions: S.

always: executable S if enabled(c1),
                          ... ,
                          enabled(cn).
        caused -enabled(c1) after S.
        ...
        caused -enabled(cn) after S.
```

Xor split

The token is passed to exactly one c_i, $i \in [1, n]$, based on the (ordered) evaluation of branching predicates associated to each of the outgoing arcs; the first one found *true* is selected. The below encoding disregards the evaluation, in order to cover as well the possibility of having unobservable activities and thus unknown value of the referred predicates.

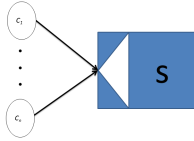



```
caused enabled(c1) if not enabled(c2), ... ,
                      not enabled(cn)
                    after S.
.....
caused enabled(cn) if not enabled(c1), ... ,
                      not enabled(cn-1)
                    after S.
```

The introduction of observed information concerning data variables together with observed tasks is discussed in a future chapter.

Xor join

S can be executed as soon as one c_i, $i \in [1, n]$ is enabled. After its execution, none of the incoming conditions can trigger S again.



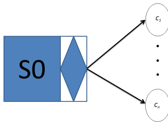

```
actions: S.

always: executable S if enabled(c1).
        ...
        executable S if enabled(cn).
        caused -enabled(c1) after S.
        ...
        caused -enabled(cn) after S.
```

Given the assumption of structured models and the execution of a unique path ensured by the XOR-split encoding, it suffices to inactivate locally the preconditions of S after one incoming arc is true.

Or split

The token is passed to at least one c_i, $i \in [1, n]$, based on the evaluation of the splitting predicates associated to each of the arcs joining them with $S0$. Unlike the XOR case, the token is advanced into all the c_i's whose predicates evaluate to *true*. As in the XOR-split, the encoding considers a generic case.

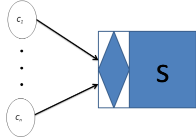



```
total enabled(c1) after S0.
.....
total enabled(cn) after S0.
forbidden not enabled(c1),
          ... , not enabled(cn)
          after S0.
```

The `total` declaration is aimed to simulate the non-determinism of the OR-split. The execution of at least one task is then asserted with the addition of the `forbidden` rule.

Or join

S receives the token only when all the incoming arcs that can be enabled according to previous selective conditions, have been done so. Due to its semantics, whether to enable or not an OR-join cannot be decided locally since reachable active tokens should be considered.




```
actions: S.

always:
   executable S if enabled(c1),
                   not delayed(c2), ... ,
                   not delayed(cn).
   .....
   executable S if enabled(cn),
                   not delayed(c1), ... ,
                   not delayed(cn-1).
   caused -enabled(c1) after S.
   .....
   caused -enabled(cn) after S.

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

Here, `reachable(Y,W)` is a static predicate associated to background knowledge that depends on the topology of the workflow. It expresses that there exists a path that leads from W to Y and is defined for all the conditions Y that have an outgoing arc connected to an OR-join task. The domain of W is the set of conditions located between the corresponding OR-split task *sp* (which exists by the structure of the models) and this OR-join, including the incoming condition to *sp*.

The strategy to compute the ground set of `reachable` elements is described in the next section, as part of the encoding algorithm.

The situation in which one of the c_is is in a waiting status and thus preventing the activation of S, is then mimicked by the fluent `delayed(`c`)`, which is true whenever c is not enabled but one of the currently enabled conditions might lead a token into it.

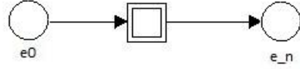
3.1.1 Extensions

The following constructs are not structured workflows by definition, nonetheless, a possible encoding considering general workflows is illustrated.

Composite task

This construct is basically a best practice to encapsulate business activities in the design stage; it can however be integrated into the main net unnesting the elements that compose it, each of which can be represented in the same

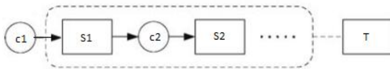
form as the elements in a non composite task. As a remark here, the input and output conditions of the composite task CT are merged with e_0 and e_n respectively; the mapping should hence be represented as follows:




```
always:
    executable CT.firstTask
             if enabled(e0).
    caused enabled(e_n)
          after CT.lastTask
```

Cancellation region

When T completes executing, all of the tasks in execution as well as all tokens residing in conditions into the associated cancellation region (for the same case) are withdrawn. Since no parallel execution is assumed, only conditions need to be considered.




```
always:
 caused -enabled(c1) after T.
 caused -enabled(c2) after T.
```

Multiple and Composite-multiple instances tasks

Since we work under the assumption of single process executions, this construct will be disregarded.

3.2 Encoding algorithm

To represent the model as a planning problem, two files need to be produced:

- A *.plan* file to contain the specifications of the planning problem expressed in $\mathcal{K}$.
- A *.dl* file to introduce background knowledge specified as a stratified Datalog program

3.2.1 The background knowledge

From the control-flow perspective, the `condition` construct has a main role due to the fact that conditions represent decision points to advance the token in an instance of the modeled process. In the case of structured workflow

nets, they indicate the instant previous and posteriors to the execution of tasks and distinguish entries and exits of structured blocks.

For the previous reasons, the background knowledge will be constituted by the proper encoding of every implicit and explicit conditions conforming the encoded model, which in agreement with the previous part connotes that the predicate `condition(`c`).` must be added to the *.dl* file, for every condition c present.

As stated before, the information concerning the reachability of conditions involved in the semantics of an OR-join construct must be added as well to the background knowledge. By the structure of the workflows here considered, extracting this facts is straightforward from the model:

1. For each task t found into an OR decision block b_i (at any nesting level), excluding the closing OR-join, add a $\texttt{reach}_{b_i}(x_j, z_i)$ fact, where x_j represents each element in the preconditions of t and the z_i's are each of the postconditions.

2. To make transitive the reachability relation in the scope of the block, add the rules:

   $\texttt{reachable}_{b_i}\texttt{(X,Y) :- reach}_{b_i}\texttt{(X,Y).}$
   $\texttt{reachable}_{b_i}\texttt{(X,Y) :- reachable}_{b_i}\texttt{(X,Z), reach}_{b_i}\texttt{(Z,Y).}$

3. Encode likewise each OR decision block, labeling the `reach` and `reachable` predicates with an identifier of the block.

The labeling of the `reach` and `reachable` predicates allows to distinguish the search scope of each OR block. An example will be provided in the following sections.

3.2.2 The $\mathcal{K}$ specification

Complying with the definition of a $\mathcal{K}$ program in [19], the *.plan* file is constructed as follows:

1. Each of the predicates used is declared as a fluent.

   $\texttt{fluents: enabled(C) requires condition(C).}$
   $\texttt{\quad\quad\quad\quad delayed}_x\texttt{(C) requires condition(C).}$

   where x represents the identifier of each OR decision block in the model.

2. The identifier t_{id} of each task construct in the model is added to the actions declaration section. Action identifiers derive directly from the names of tasks in the model. The identifier of the input and output conditions will be "*start*" and "*end*" respectively.

   `actions:` t_{id}.

3. The *always* section is constituted with the encoding related to each of the constructs present in the model, excluding the conditions.

4. An `inertial` fluent preserves its truth value through actions while not explicitly changed, in which case, the new value is preserved. Taking advantage of the corresponding $\mathcal{K}$ shortcut and to keep track of flows in the model, include the declaration:

   ```
   inertial enabled(X).
   ```

5. Mindful of the fact that the objective of the encoding points to traces completion and considering that traces of models are given as sequences of tasks, regardless the possible simultaneous executions, the declaration `noConcurrency.` is added as well in the *always* section.

6. The *initially* and *goal* sections are defined respectively through the input and output condition encoding.

3.2.3 Complexity

Inasmuch as each task and each condition are encoded exactly once, the complexity of the presented algorithm is linear in the size of the model; since a fixed number of rules and facts are produced from each YAWL element, the size of the produced encoding is linear as well.

The complexity of computing cases of the modeled process is determined then by the complexity of finding a plan for the corresponding planning problem, which is asserted to be PSPACE-complete [19]. The formal proof of the complexity is considered as future work.

3.2.4 Simple example

The programs resulting of carrying out the algorithm over the model in figure 3.1, are provided for illustration purposes.

The process depicted executes $S0$, which is a multiple choice task, then $S3$, $S1$ or both can be executed. If the upper branch is selected, $S4$ must be eventually executed after $S3$ and then the process can either loop or go to the merging task. If $S1$ is executed then $S2$ must be as well eventually executed before moving to $S5$, which can take place only if all the tasks corresponding to the selected branches have finished. In the loop case, the last task of an iteration must be $S4$. The process finishes after the execution of $S5$.

The labels next to the arcs identify the implicit condition between each pair of tasks.

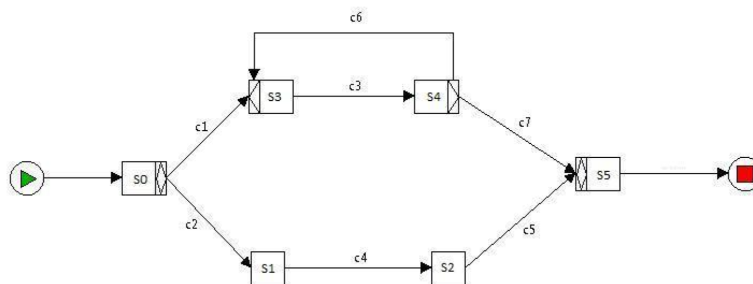


Figure 3.1 A simple example

The encoding

```
fluents:  enabled(C) requires condition(C).
          delayed(C) requires condition(C).

actions: s0. s2. s1. s5. s3. s4.

always:
executable s0 if enabled(start).
caused -enabled(start) after s0.

total enabled(c1) after s0.
total enabled(c2) after s0.
forbidden not enabled(c1),not enabled(c2) after s0.

executable s1 if enabled(c2).
caused enabled(c4) after s1.
caused -enabled(c2) after s1.

executable s2 if enabled(c4).
caused enabled(c5) after s2.
caused -enabled(c4) after s2.
```

```
caused -enabled(c5) after s5.
caused -enabled(c7) after s5.
executable s5 if enabled(c5),not delayed(c7).
executable s5 if not delayed(c5),enabled(c7).
caused delayed(Y) if not enabled(Y), reachable(Y,W), enabled(W).
caused enabled(end) after s5.

executable s3 if enabled(c1),not enabled(c6).
executable s3 if not enabled(c1),enabled(c6).
caused -enabled(c1) after s3.
caused -enabled(c6) after s3.

caused enabled(c3) after s3.

executable s4 if enabled(c3).
caused -enabled(c3) after s4.

caused enabled(c6) if not enabled(c7) after s4.
caused enabled(c7) if not enabled(c6) after s4.

inertial enabled(X).
noConcurrency.

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

The background knowledge file associated to this process is conformed as follows:

```
condition(start). condition(end).
condition(c1). condition(c2). condition(c3).
condition(c4). condition(c5). condition(c6).
condition(c7).

reach(c1,start). reach(c2,start). reach(c3,c1).
reach(c4,c2). reach(c7,c3). reach(c5,c4).
reach(c6,c3).

reachable(X,Y) :- reach(X,Y).
reachable(X,Y) :- reachable(X,Z), reach(Z,Y).
```

The execution of the $\mathcal{K}$ program with the obtained background knowledge, using the DLV^K planner, gives as solutions by instance exactly the two plans with length four:

```
STATE 0: delayed(c3), delayed(c6), delayed(c4), delayed(c5),
```

```
         delayed(c2), delayed(c7), delayed(c1), enabled(start)
ACTIONS: s0
STATE 1: -enabled(start), enabled(c1), -enabled(c2), delayed(c3),
         delayed(c6), delayed(c7)
ACTIONS: s3
STATE 2: -enabled(c1), enabled(c3), delayed(c6), delayed(c7),
         -enabled(c6)
ACTIONS: s4
STATE 3: -enabled(c3), enabled(c7)
ACTIONS: s5
STATE 4: -enabled(c5), -enabled(c7), enabled(end)
   PLAN: s0; s3; s4; s5
```

```
STATE 0: delayed(c3), delayed(c6), delayed(c4), delayed(c5), delay
ACTIONS: s0
STATE 1: -enabled(start), -enabled(c1), enabled(c2), delayed(c4),
         delayed(c5)
ACTIONS: s1
STATE 2: -enabled(c2), enabled(c4), delayed(c5)
ACTIONS: s2
STATE 3: -enabled(c4), enabled(c5)
ACTIONS: s5
STATE 4: -enabled(c5), -enabled(c7), enabled(end)
   PLAN: s0; s1; s2; s5
```

As a remark, since there is only one OR decision block in the model, the subscript labeling of the reachable predicates was omitted and hereafter, it will be done so whenever this situation occurs. To strictly comply with the algorithms however, the predicates should be taking the label of the block, then, assuming it is b_0, the modifications in the encoding would be:

fluents: delayed$_{b_0}$(C) requires condition(C).

always:
executable s5 if enabled(c5),not delayed$_{b_0}$(c7).
executable s5 if not delayed$_{b_0}$(c5),enabled(c7).
caused delayed$_{b_0}$(Y) if not enabled(Y), reachable$_{b_0}$(Y,W), enabled(W).

And for the background knowledge:

reach$_{b_0}$(c1,start). reach$_{b_0}$(c2,start). reach$_{b_0}$(c3,c1).
reach$_{b_0}$(c4,c2). reach$_{b_0}$(c7,c3). reach$_{b_0}$(c5,c4). reach$_{b_0}$(c6,c3).

reachable$_{b_0}$(X,Y) :- reach$_{b_0}$(X,Y).
reachable$_{b_0}$(X,Y) :- reachable$_{b_0}$(X,Z), reach$_{b_0}$(Z,Y).

3.3 Correctness of the encoding

The encoding proposed in section 3.1 is intended to simulate the behavior of a given YAWL specification by reproducing its semantics in the context of a planning problem. The correctness of such simulation is formally proved in this section, based on the formalisms introduced in the preliminary chapter.

An useful property observed in SWF-nets will be proved first.

Property 3.3.1. In a SWF-net, after the execution of a block there are no tokens left neither in the inner conditions nor in the entry.

Proof. By structural induction considering the cases in def. 2.2.1:

1. The atomic task case is trivial.

2. The token in c_1 is advanced into X and after its execution, to the entry of Y, leaving by induction no tokens left in X. Then it is eventually consumed by Y finishing, again by induction without tokens inside that block and thus no tokens remain in the block sequence.

3. As 4.

4. The token in c_1 is passed to some or all the entries of the X_n's, depending on the case of s; given that SWF-nets are deadlock free, eventually each token is advanced to the exit of the corresponding X_n, leaving by induction X_n cleared; next, according to the case, eventually c_2 is triggered and by semantics, the tokens in all the conditions are consumed and one in c_2 is produced.

5. c_1 activates j passing the token to j_X then by induction X advances it to X_s with no tokens left in X; hereafter, either s passes the token to the exit, in which case the property holds by semantics of s, or it passes it to Y_j again without leaving tokens in Y by induction.

□

As a corollary and since the blocks have a single entry:

Property 3.3.2. In a SWF-net, a block with tokens in the inner conditions has no tokens in its entry.

3.3.1 Bisimulation between YAWL models and $\mathcal{K}$ programs

The correctness of the bisimulation reduces to prove that for each execution instance of a process in a model there exists a plan generated by its corresponding $\mathcal{K}$ encoding and vice versa.

Outline of the proof: The encoding algorithm creates an action and a fact predicate in the $\mathcal{K}$ program for each task and condition in the model; the conditions satisfied at a certain moment of the execution determine states, thus, a task leads from a state x to a state y in the model iff the corresponding action in the encoding does so with the states mapping x and y. This holds because after the execution of an action in the program, the proper postconditions are enabled and all the preconditions are disabled, which simulates the advance of the token in the net. In the case of OR-join tasks, the fact of having structured models guarantees the correct activation of the action. Since this property holds for every pair of states, it is also valid in the initial states of the model and the program, as well as in every step involved in the sequence of tasks/actions going to the respective final states, which again by the mapping function, are equivalent.

Let us start defining a bijective function to match each element in the workflow with an element in $\mathcal{K}$ and formalizing the concerning notion of bisimulation.

Let $N = (C, i, o, T, Z, split, join)$ be a structured workflow net in YAWL, $K_N = (F, A, R, init_rules, goal)$ a $\mathcal{K}$ program and $map : T \cup C \mapsto A \cup F^C$ a bijective function such that for $x \in T$, $map(x) \in A$ and for $x \in C$, $map(x) \in F^C$, for a predefined $F^C \subseteq F$.

As an abuse of notation, $map(\{c_1, \ldots, c_n\}) = \{map(c_1), \ldots, map(c_n)\}$ determines a correspondence between states from a workflow net and states from a $\mathcal{K}$ program, assigning a unique fluent to each condition holding a token in a given workflow state.

Definition 3.3.1. N and K_N are ***bisimilar*** ($N \simeq K_N$) iff there exists a definition of map such that: $T_{N,id} = \langle t_1, \ \ldots \ , t_n \rangle$ is a case in N iff $k_{plan,id} = \langle map(t_1), \ \ldots \ , map(t_n) \rangle$ is a partial plan for K_N.

$T_{N,id}$ is a completed case iff $k_{plan,id}$ is a plan.

To proceed, the next lemma captures the simulation between each pair of states.

Lemma 3.3.1. *Let K_N be the $\mathcal{K}$ encoding obtained from N through the application of rules in section 3.1. Given a definition map_e for the function map and states $s_1, s2, \sigma_1, \sigma_2$ such that $s_1 = map_e(\sigma_1)$ and $s_2 = map_e(\sigma_2)$, then, $\sigma_1 \xrightarrow{h} \sigma_2 \in N$ iff $[s_1, map_e(h), s_2]$ is a legal transition in K_N.*

Proof.
By definition of the encoding procedure, the unique fluents in K_N are `enabled(`X`)` and `delayed(`X`)`; provided that the second is an auxiliary fluent and considering that each action $a \in K_N$ is derived exactly from one task in N, named moreover with the same identifier, let's first define $F^C = \{enabled(x) \mid condition(x)$ is in the background knowledge of $K_N\}$ and

$$map_e(x) = \begin{cases} x & \text{if } x \in T \\ \texttt{enabled(}x\texttt{)} & \text{if } x \in C \end{cases}$$

For simplicity and w.l.o.g., we will use indistinctly x in the subsequent to refer to each, the action and the task. Additionally, the possible presence of *delayed* fluents in s_1 and s_2 will be ignored when referring to the mapping, since it is not part of the defined domain for map_e . Finally, recall two facts:

[F1]: N being structured implies that every task t with a multiple join is followed by a single condition, i.e. $split(t) = AND$ with a sole outgoing edge (def. 2.2.2). Analogously, every multiple split task has exactly one incoming condition.

[R_I]: By definition, `inertial enabled(X)` is part of every $\mathcal{K}$ encoding.

Conscious about this, let's assume $\sigma_1 \xrightarrow{h} \sigma_2 \in N$. Then, by def. 2.2.5, $\exists$ $pre, post \subseteq C$ and *binding(h,pre,post,*σ_1*)* such that $\sigma_2 = (\sigma_1 \setminus pre) \cup post$.

With [F1] in mind and assuming the next given *preset* and *postset* for h (as per def. 2.2.4), we have the following cases and the corresponding encodings derived from the value of h's *split* and *join* functions:

- *AND split*: $preset(h) = \{c_{pre}\},\ postset(h) = \{c_1, \ldots,\ c_n\}$

  ```
  executable h if enabled(c_pre).              (R1)
  caused -enabled(c_pre) after h.
  caused enabled(c_1) after h.
  ...
  caused enabled(c_n) after h.
  ```

  From def. 2.2.4: $post = postset(h)$, $pre = \{c_{pre}\}$ and $\sigma_1 = \{d_1, \ldots, d_m, c_{pre}\}$ with $m \geq 0$; thus, by hypothesis, $\sigma_2 = \{d_1, \ldots, d_m, c_1, \ldots, c_n\}$. This implies that `enabled(`$c_{pre}$`)` $\in s_1$ and thus that h is executable with respect to s_1, due to (R1) and provided that rules of type `nonexecutable`, which might prevent its execution, are never introduced. Then, $map(\sigma_2) = \{$`enabled(`d_1`)`, $\ldots$, `enabled(`d_m`)`, `enabled(`c_1`)`, $\ldots$, `enabled(`c_n`)`, `not enabled(`c_{pre}`)`$\}$ is the minimal state satisfying the causation rules concerning h; therefore $[s_1, h, s_2]$ is a legal transition in K_N (according to def. 2.3.3).

- *AND join.* $postset(h) = \{c_{aft}\},\ preset(h) = \{c_1, \ldots,\ c_n\}$

```
executable h if enabled(c1), ... , enabled(cn).
caused -enabled(c1) after h.
...
caused -enabled(cn) after h.
caused enabled(caft) after h.
```

From def. 2.2.4: *post* $= \{c_{aft}\}$, *pre* $=$ *preset*(h) and $\sigma_1 = \{d_1, \ldots, d_m, c_1, \ldots, c_n\}$ with $n \geq 0$; by hypothesis then $\sigma_2 = \{d_1, \ldots, d_m, c_{aft}\}$. Given that thereupon, $\forall j \in [1, n]$, enabled(c_j) $\in s_1$, h is executable with respect to s_1 and the state after the execution is $\{$enabled(d_1), ..., enabled(d_m), -enabled(c_1), ...,-enabled(c_n), enabled(c_{aft})$\}$ (considering $[R_I]$), which by the semantics of the default negation operator in $\mathcal{K}$, implies $\{$enabled(c_{aft}), enabled(d_1), ..., enabled(d_m), not enabled(c_1), ..., not enabled(c_n) $\}$. This state corresponds to s_2. Hence, $[s_1, h, s_2]$ is a legal transition in K_N.

- *XOR split.* *preset*$(h) = \{c_{pre}\}$, *postset*$(h) = \{c_1, \ldots, c_n\}$

```
executable h if enabled(cpre).
caused -enabled(cpre) after h.
caused enabled(c1) if not enabled(c2), ... , not enabled(cn)
                   after h.
...
caused enabled(cn) if not enabled(c1), ... , not enabled(cn−1)
                     after h.
```

We will consider the general case in which branching selection predicates are not introduced. The management of data will be explained later in this document.

Def. 2.2.4 implies *pre* $= \{c_{pre}\}$, $|$*post*$| = 1$ and *post* $\subseteq$ *postset*(h), then $\sigma_1 = \{d_1, \ldots, d_m, c_{pre}\}$ and $\sigma_2 = \{d_1, \ldots, d_m, c_x\}$ for some $x \in$ *postset*(h), $m \geq 0$. Now, h is executable with respect to s_1 for the same reasons that the AND split case. Different possibilities exist however for the state to be produced, to tell $\{$enabled(d_1), ..., enabled(d_m), enabled(c_i)$\}$ for any $i \in [1, n]$, regarding though that once one enabled(c_i) is selected, the preconditions of the others become unachievable; for this reason, each of the referred states is minimal. Given that moreover, s_2 is one of those states, $[s_1, h, s_2]$ is a legal transition in K_N.

- *XOR join.* *postset*$(h) = \{c_{aft}\}$, *preset*$(h) = \{c_1, \ldots, c_n\}$

```
executable h if enabled(c1).         (R1)
...
executable h if enabled(cn).
caused -enabled(c1) after h.
...
caused -enabled(cn) after h.
```

```
caused enabled(c_aft) after h.
```

From def. 2.2.4, we have $post = \{c_{aft}\}$, $|pre| = 1$ and $pre \subseteq preset(h)$, therefore w.l.o.g. $pre = \{c_1\}$ and $\sigma_1 = \{d_1, \ldots, d_m, c_1\}$ with $m \geq 0$; by hypothesis then $\sigma_2 = \{d_1, \ldots, d_m, c_{aft}\}$. Since $map_e(\sigma_1) = s_1$, `enabled(`c_1`)` $\in s_1$ and thus h is executable with respect to s_1 due to (R1). The state after the execution is {`enabled(`d_1`)`, ..., `enabled(`d_m`)`, `-enabled(`c_1`)`, ..., `-enabled(`c_n`)`, `enabled(`c_{aft}`)`}, which as explained in the AND join case, corresponds to s_2. Hence, $[s_1, h, s_2]$ is a legal transition in K_N.

- *OR split.* $preset(h) = \{c_{pre}\}$, $postset(h) = \{c_1, \ldots, c_n\}$

```
executable h if enabled(c_pre).
caused -enabled(c_pre) after h.
total enabled(c_1) after h.                 (R1)
...
total enabled(c_n) after h.
forbidden not enabled(c_1), ... , not enabled(c_n)
          after h.                                      (R_F)
```

Again let's consider the general case without branching selection predicates.

From 2.2.4, $pre = \{c_{pre}\}$, $\emptyset \neq post \subseteq postset(h)$ and $\sigma_1 = \{d_1, \ldots, d_m, c_{pre}\}$ for $m \leq 0$. Then, w.l.o.g. $\sigma_2 = \{d_1, \ldots, d_m, c_1, \ldots, c_k\}$ with $k \in [1, n]$. h's executability is once again the same as in the AND split case. Regarding the reached state, (R_F) guarantees the existence of one `enabled(`c_i`)`, $i \in [1, n]$; by definition of `total` (see [13]), (R1) is equivalent to have

```
caused enabled(c_1) if not -enabled(c_1) after h.
caused -enabled(c_1) if not enabled(c_1) after h.
```

By semantics of $\mathcal{K}$, the behavior of those rules is to non deterministically assign a truth value to `enabled(`c_1`)`, whenever its value is unknown. As this is analogous for the rest of the `total` rules, the set of achievable states after the execution of h is the power set of $map_e(postset(h))$, where each element is additionally joined with {`enabled(`d_1`)`, ..., `enabled(`d_m`)`}; the element where all `enabled(`c_i`)` are negated is excluded.

Since the power set contains all the possible combinations, each element (let it be s_{2i}) is minimal and so each $[s_1, h, s_{2i}]$ is a legal transition in K_N. In particular s_2 is one of the s_{2i}; therefore, $[s1, h, s_2]$ is a legal transition in K_N.

- *OR join.* $preset(h) = \{c_1, \ldots, c_m\}$ and $postset(h) = \{c_{post}\}$

```
executable h if enabled(c_1),
                not delayed(c_2), ... , not delayed(c_m).    (E1)
...
executable h if enabled(c_m),
                not delayed(c_1), ... , not delayed(c_m-1).  (Em)
caused -enabled(c_1) after h.
...
caused -enabled(c_m) after h.
```

```
caused enabled(c_post) after h.

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

By hypothesis and def. 2.2.5:

(i) $\exists$ *pre*, *post* $\subseteq C$ and *binding*$(h, pre, post, \sigma_1)$ s.t. $\sigma_2 = (\sigma_1 \setminus pre) \cup post$

(ii) $\forall\ \delta \in reach(\sigma_1)$, $\nexists\ pre' \subseteq \delta$ s.t. there is a *binding(h, pre, post, δ)* and $pre \subset pre'$

From (i) and def. 2.2.4: $post = \{c_{post}\}$, w.l.o.g. $pre = \{c_1, \ldots, c_k\}$ for some $1 \leq k \leq m$ and $\sigma_1 = pre \cup \{c'_1, \ldots, c'_n\}$ with $n \geq 0$ and $\sigma_2 = \{c'_1, \ldots, c'_n, c_{post}\}$; consequently `{enabled(`$c_1$`), ..., enabled(`$c_k$`)}` $\subseteq s_1$ and thus none of `delayed(`c_1`), ..., delayed(`c_k`)` is caused in that state, since the preconditions of (R_D) don't hold, i.e., `not delayed(`c_j`)` is true in s_1 for all $1 \leq j \leq k$.

Aside, by construction of the encoding, for each condition w in a path that ends in c_r and starts in the OR-split task connected to the entry(e_h) of the block whose exit is c_{post}, there is a `reachable(`c_r, w`)` fact in the knowledge base; that block exists due to the assumption of N being structured; `reachable(`c_r, w`)` is then true in every state of K_N, exactly for all the referred w's.

Given that (ii) holds in σ_1, $\nexists\ pre' \subseteq C$ in none of the states δ reachable from σ_1 by a (sequence of) task(s) different from h, nor in σ_1 itself, such that h can be executed in δ with pre'. This implies that:

- `enabled(`c_r`)`, $k+1 \leq r \leq m$, are not in s_1, i.e., `not enabled(`c_r`)` is true in s_1.
- $\forall\ w$ previously introduced, $w \notin \{c'_1, \ldots, c'_n\}$
- Since h won't be activated, $\forall\ \delta$ holds $pre \subseteq \delta$, then by property 3.3.2, $e_h \notin \delta$, whereby jointly with the previous point derives that $\forall\ \delta, w$, holds $w \notin \delta$.

Provided that $map_e(\sigma_1) = s_1$, then `not enabled(`w`)` is true in s_1 and furthermore it will be so in all the states produced after s_1 from executing an action different to h. Consequently and because `reachable(`c_r, x`)` is false for any other condition x, `delayed(`c_r`)` is not caused in s_1 and hence `not delayed(`c_r`)` is true.

As a result, h is executable with respect to {`enabled(`c_1`)`, ..., `enabled(`c_k`)`, `enabled(`c'_1`)`, ..., `enabled(`c'_n`)` } $= s_1$, for instance with (E1) and being aware of $[R_I]$, its execution leads to the state $s_2 =$ { `enabled(`c'_1`)`, ..., `enabled(`c'_n`)`, `enabled(`c_{post}`)`} $= map_e(\sigma_2)$.

Therefore, $[s_1, h, s_2]$ is a legal transition in K_N.

An atomic h corresponds to the case when $split(h) = AND$ and $join(h) = XOR$ both with cardinality one, which is covered by the previous cases.

For the left hand side direction, let's assume that $[s_1, h, s_2]$ is a legal transition in K_N. This implies that h is executable with respect to s_1 and that $f \in s_2$ for each `caused` f `if` $\bigwedge sss_2$ `after` $\bigwedge sss_1,\ h.$ $\in K_N$, where $sss_1 \subseteq s_1,\ sss_2 \subseteq s_2$.

The existence of the required transition in N is shown as well according to the type of the associated task h. The same *preset* and *postset* of the right hand side part of the proof will be assumed, thus the same encoding for each case.

- *AND split*
  From the assumption, $s_1 = \{$`enabled(`d_1`)`$, \ldots,$ `enabled(`d_y`)`, `enabled(`c_{pre}`)`$\}$ and $s_2 = \{$`enabled(`d_1`)`$, \ldots,$ `enabled(`d_y`)`, `enabled(`c_1`)`$, \ldots,$ `enabled(`c_n`)`$\}$ for some $y \geq 0$. Let $pre = preset(h)$ and $post = postset(h)$, then $pre \subseteq \sigma_1$ and therefore, recalling N is structured, $\sigma_1 \cap post = \emptyset$, otherwise property 3.3.2 would be violated (notice that because map_e is bijective, $\{$`enabled(`d_i`)` $\mid 1 \leq i \leq y\} \cap \{$`enabled(`$c_j$`)`$|1 \leq j \leq n\} = \emptyset$ holds as well in K_N).

  Thence, there exists $binding(h, pre, post, \sigma_1)$, which together with the implication that $\sigma_2 = (\sigma_1 \setminus pre) \cup post$, entails that $\sigma_1 \xrightarrow{h} \sigma_2 \in N$.

- *AND join*
  From the hypothesis, $\{$`enabled(`c_1`)`, $\ldots$, `enabled(`c_n`)`$\} \subseteq s_1$ and $\{$`not enabled(`c_1`)`, $\ldots$, `not enabled(`c_n`)`, `enabled(`c_{aft}`)`$\} \subseteq s_2$ for some $y \geq 0$. Considering then $pre = preset(h)$ and $post = postset(h)$, we derive $pre \subseteq \sigma_1$ and $\sigma_2 = (\sigma_1 \setminus pre) \cup post$. Besides, in agreement with property 3.3.1, $\sigma_1 \cap \{c_{aft}\} = \emptyset$, which by definition of map_e guarantees as a plus that `enabled(`c_{aft}`)` $\notin s_1$.

  Therefore, there exists $binding(h, pre, post, \sigma_1)$, which together with the previous statement about σ_2 implies that $\sigma_1 \xrightarrow{h} \sigma_2 \in N$.

- *XOR split*
  From the encoding and the initial assumption derives that `enabled(`c_{pre}`)` $\in s_1$ and $\{$`-enabled(`c_{pre}`)`, `enabled(`c_x`)`$\} \subseteq s_2$, for some $x \in [1, n]$. Let now $pre = preset(h)$ and $post = \{c_x\}$; then, $post \subseteq postset(h)$ and since $c_{pre} \in \sigma_1$, by property 3.3.2, $\sigma_1 \cap post = \emptyset$, which given the bijectivity of map_e implies in plus that `enabled(`c_x`)` $\notin s_1$. As a consequence, $\sigma_2 = (\sigma_1 \setminus pre) \cup post$.

  Given that all conditions of def. 2.2.4 hold, there is a $binding(h, pre, post, \sigma_1)$ and thus $\sigma_1 \xrightarrow{h} \sigma_2 \in N$.

- *XOR join*
  W.l.o.g. $\{$`enabled(`c_1`)`, $\ldots$, `enabled(`c_k`)`$\} \subseteq s_1$ for some $1 \leq k \leq n$, $\{$`not enabled(`c_1`)`, $\ldots$, `not enabled(`c_n`)`, `enabled(`c_{aft}`)`$\} \subseteq s_2$ is implied; then $\{c_1, \ldots,\ c_k\} \subseteq \sigma_1$; given the structure of N, this implies that an XOR split must have been executed, which guarantees that exactly one c_x, $1 \leq k \leq x$ appears in σ_1. With $pre = \{c_x\}$ and $post = \{c_{aft}\}$, this case holds analogous to the AND join.

- *OR split*
  Being h executable with respect to s_1 implies from the encoding that { -

`enabled(`c_{pre}`)`, (-)?`enabled(`c_1`)`, … , (-)?`enabled(`c_n`)`}[2] $\subseteq s_2$, where at least one of the `enabled(`c_i`)` is positive and that `enabled(`c_{pre}`)` $\in s_1$. Since $s_1 = map_e(\sigma_1)$, $c_{pre} \in \sigma_1$ and $\emptyset \neq \{c_1?, ..., c_n?\} \subseteq \sigma_2$. Let now $post = \sigma_2 \cap postset(h)$; given that N is structured and by property 3.3.2 $\sigma_1 \cap post = \emptyset$, since by definition h is the first task of a block.

Because map_e is bijective, $s_1 \setminus$ {`enabled(`c_{pre}`)`} $= s_2 \setminus${`-enabled(`c_{pre}`)`, (-)?`enabled(`c_1`)`, …, (-)?`enabled(`c_n`)`} and then taking $pre = \{c_{pre}\}$, σ_2 is equal to $\sigma_2 = (\sigma_1 \setminus pre) \cup post$. As a consequence as well, by def. 2.2.4, there is a $binding(h, pre, post, \sigma_1)$ and thus $\sigma_1 \xrightarrow{h} \sigma_2 \in N$.

- *OR join*
  Recall first that (i) since N is structured, there exists an OR-split task h_{split}, which is the first task of the block whose last task is the encoded h. Let $block_h$ denote the mentioned block.

  Then, from hypothesis, {`enabled(`c_{post}`), -enabled(`c_j`)`} $\subseteq s_2$, for all $1 \leq j \leq m$ and at least one of (E1) to (Em) is satisfied in s_1, i.e., $\exists\ c_j$ s.t. `enabled(`c_j`)` and `not delayed(`c_k`)` are in s_1, $\forall\ 1 \leq k \leq m$. (note that `not delayed(`c_j`)` holds too due to (R_D)). Then, $s_1 = \{f_1, \ldots, f_x,$ `enabled(`c_1`)`, … , `enabled(`c_k`)`}, with $1 \leq k \leq m$ and f_i fluents s.t. $\{f_i\} \cap$ {`enabled(`c_j`)`} $= \emptyset$, $i \geq 0$.

  Aware of $[R_I]$, $\{f_1, \ldots, f_y,$ `enabled(`c_{post}`)`} $\subseteq s_2$ is as well implied from $[s_1, h, s_2]$, with $y \leq x$ and $f_1, \ldots, f_y$ all the fluents that in s_1 correspond to the `enabled(X)` predicate.[3]

  Let $pre = \{x \in preset(h) |$ `enabled(`x`)` $\in s_1\}$ and $post = \{c_{post}\}$; we have then $\sigma_1 = \{d_1, \ldots, d_y, c_1, \ldots, c_k\}$ and $\sigma_2 = \{d_1, \ldots, d_y, c_{post}\}$, assuming $f_i = enabled(d_i)$. According thus to def. 2.2.4, $\exists\ binding(h, pre, post, \sigma_1)$ and because $\sigma_2 = (\sigma_1 \setminus pre) \cup post$ as well, hence (1.) from def. 2.2.5 holds.

  Now, since `not delayed(`c_k`)` $\in s_1$, $\forall\ 1 \leq k \leq m$, then for each c_k either:

  a) `enabled(`c_k`)` $\in s_1$, in which case $c_k \in pre$

  b) `not enabled(`c_k`)` is true and [`reachable(`c_k, w`)`, `enabled(`w`)`] is false
     Let's consider $B = \{w|$ `reachable(`c_k, w`)` is in the background knowledge}[4]. $B \neq \emptyset$ due to (i). Since `reachable(`c_k, w`)` is a fact (thus true), then `enabled(`w`)` must be false, i.e., $\forall\ w \in B$, `not enabled(`w`)` holds in s_1.

     Therefrom, $\{c \in block_h | c \in C\} \cap \sigma_1 = pre$, which implies that the only task executable from $block_h$ in σ_1 is h itself. Since by definition h can't be executed in any of the states $\delta \in reach(\sigma_1)$, then $\forall \delta$ holds $pre \subseteq \delta$, which by property 3.3.2 implies that the entry of $block_h$ is not in any δ and thus none of $\{c \in block_h\} \setminus pre$ either.

     Taking so $preset(h_{split}) = \{x\}$ ([F1]), `not enabled(`x`)` remains true in every state previous to the execution of h, which together with the

[2] ? is the operator for regular expressions to denote 0 or 1 occurrence of the operated symbol

[3] The rest of the f_i's can be fluents of the type `delayed(_)`

[4] By construction, those are all the conditions in the path from h_{split}, leading to c_k, including $preset(h_{split})$

construction of the encoding, implies that all of the `not enabled(`w`)` remain true as well.

Recalling that map_e is a bijection between N's states and the states in K_N, from (a) derives that there is no pre' such that $\exists\ binding(h, pre', post, \sigma_1)$ and $pre \subset pre' \subseteq \sigma_1$. And from (b), that none of the elements in $preset(h) \setminus \sigma_1$ can be part of any state δ reachable from σ_1 by a task distinct from h; thus, there is no $pre' \subseteq \delta$ s.t. $pre \subset pre' \subseteq preset(h)$. Consequently, (2.) from def. 2.2.4 holds.

□

At this point , the expected bisimulation of the proposed encoding is asserted by the following theorem.

Theorem 3.3.1. *Let N = (C, start, end, T, Z, split, join) be a structured workflow net in YAWL and K_N the $\mathcal{K}$ encoding obtained from the application of the rules in section 3.1 to N. Then $N \simeq K_N$.*

Proof. Let *map* be the map_e function defined in the proof of lemma 3.3.1. Then,

$\langle t_1, \ \ldots \ , t_n \rangle$ is a case in N

$\overset{def.2.2.6}{\Longleftrightarrow}$ $\exists\ \sigma_0, \ldots, \sigma_n$ workflow states of N, s.t. $\sigma_{j-1} \overset{t_j}{\to} \sigma_j$ is a transition in N, with $j \in [1, n]$ and $\sigma_0 = \{init\}$

$\overset{by\ lemma\ 1}{\Longleftrightarrow}$ $\exists\ s_0, \ldots, s_n$ states of K_N, s.t. $[s_{j-1}, map(t_j), s_j]$ is a legal transition in K_N, with $j \in [1, n]$, $s_j = map(\sigma_j)$ and s_0 is the initial state

$\Longleftrightarrow$ $\langle [s_0, map(t_1), s_1], \ \ldots, \ [s_{n-1}, map(t_n), s_n] \rangle$ is a legal transition sequence in K_N

$\overset{def.2.3.5}{\Longleftrightarrow}$ $\langle map(t_1), \ \ldots \ , map(t_n) \rangle$ is a partial plan in K_N

Hence, $(N \simeq K_N)$ by definition 3.3.1. □

Chapter 4

Restriction of plans to observed executions

Informally a *trace* of a process is the record of a sequence of tasks observed during an execution of such process. Depending on the observability of the tasks in the modelling SWF-net, the trace can contain all of the executed tasks or simply a part of them.

A general algorithm to include information regarding traces into the $\mathcal{K}$ representation of a SWF-net is presented in this section. The intuitive idea is to constraint the execution of observable actions to the detection in the trace of any instance of such action (captured by a ground fluent). The trace sequence is then encoded in such way that each action in it can be executed uniquely if the previous one has been done so. The execution of all the actions is then asserted by the addition of the fluent depicting the observed end of the trace to the goal.

4.1 Algorithm for inclusion of traces

Let P be the encoding in $\mathcal{K}$ of a YAWL specification and T a trace of the process described by P. Then, proceeding as follows, the set of solutions for P can be restricted to contain only those plans p_t such that, for each occurrence of a task in T, the action that corresponds to it in P appears in p_t, maintaining the observed order.

1. Add a new fluent `observed(X,Y)` to P, with X an action and Y an integer. This fluent is intended to represent an action observed in a case, assigning it an identifier. Notice that although Y can be any different integer for each action, an ordered sequence will be used here in order to indicate the execution order. By instance, the elements in the trace *s1, s3, s4, s3, s5, s6* will be represented as *observed(s1,1), observed(s3,2), observed(s4,3), observed(s3,4), observed(s5,5), observed(s6,6)*.

2. Make the new fluent inertial to propagate its value.

```
inertial observed(X,Y).
```

3. For every observable action *act* in P, `observed(`act`,_)`, must be added to the if condition of the respective `executable act` constraints, so that the sequence in $\mathcal{T}$ is respected and since observable actions can be performed only if there is track of their execution.

```
executable act if observed(act,_), ...  .
```

4. To encode the observed sequence of tasks, for each `observed(`x, i`)` add a rule

```
caused observed(z, i+1) after x, observed(x,i).
```

where z denotes the action subsequent to `observed(`x, i`)` in $\mathcal{T}$. If x is the last task in the trace, then add instead

```
caused observed(end, i+1) after x, observed(x,i).
```

5. Disable each action in $\mathcal{T}$ after executed.

```
caused -observed(A,N) if observed(_,M), M=N+1 after
                    observed(A,N).
```

6. The grounded fluent `observed(end,`i`)` introduced above, must be added to the *goal* constraints in order to ensure that all the observed actions form part of the solution.

7. Add to *initially:* the grounding of the fluent associated to is the first action in $\mathcal{T}$. (Ex. observed(t0,1))

Let's consider the simple example from figure (3.1) in section 3.2.4. Assuming that all the activities except $s2$ and $s4$ are observable, for $\mathcal{T} = s0, s3, s1, s5$, the corresponding $\mathcal{K}$ constraints are:

```
fluents:
 observed(X,Y).

always:
   executable s0 if enabled(start), observed(s0,_).
   executable s1 if enabled(c0), observed(s1,_).
   executable s3 if enabled(c3),not enabled(cTo_s3), observed(s3,_).
   executable s3 if not enabled(c3),enabled(cTo_s3), observed(s3,_).
   executable s5 if enabled(c2),not delayed(c4), observed(s5,_).
   executable s5 if not delayed(c2),enabled(c4), observed(s5,_).

   %% trace
   caused observed(s3,2) after s0, observed(s0,1).
```

```
    caused observed(s1,3) after s3, observed(s3,2).
    caused observed(s5,4) after s1, observed(s1,3).
    caused observed(end,5) after s5, observed(s5,4).

    %% disable executed actions
    caused -observed(A,N) if observed(_,M), M=N+1 after observed(A,N).

    inertial observed(X,Y).

initially:  .... observed(s0,1).
goal: end, observed(end,5)?
```

4.2 Validity of the algorithm

In this section we will prove the soundness and completeness of the algorithm proposed to restrict the set of solutions in agreement with information from traces. Let us formalize some concepts first.

Let M be a SWF-net and tsk the set of tasks in M. Then $\mathcal{O}^{\mathcal{M}} \subseteq tsk$ denotes the set of observable tasks in M and $\mathcal{U}^{\mathcal{M}} \in tsk$ the set of unobservable tasks, such that $tsk = \mathcal{O}^{\mathcal{M}} \cup \mathcal{U}^{\mathcal{M}}$.

Definition 4.2.1. A ***trace*** is a subsequence of observable tasks derived from a case. As an abuse of notation, $T \subseteq T_{case}$ denotes that T is a trace of the case T_{case}.

Let K_M be the $\mathcal{K}$ encoding of a SWF-net M and T an observed trace of the process described by M.

Taking into account the theorem 3.3.1, each T_{case} of M has a corresponding plan P_{case} for K_M, compounded exactly by the same tasks in the same order. Based on this fact, the notation $T \subseteq P_{case}$ will be used as well to symbolize that a subsequence containing the tasks in a trace T is a subsequence of the plan P_{case}.

Let besides $K_M|_T$ designate the encoding resulting from the application of the algorithm in the previous section to K_M and T. The soundness of the algorithm is then asserted by the two following theorems, for which an auxiliary lemma is proved first.

Lemma 4.2.1. *A partial plan for $K_M|_T$ is a partial plan for K_M.*

Proof. In addition to the rule in step 5 of the algorithm, the rules in K_M are extended and modified in $K_M|_T$ as follows, where o_i represents each element in the set of observable tasks and $preconditions_{K_M}(r)$ denotes all the preconditions defined for the same rule in K_M.

```
executable oᵢ if observed(oᵢ,_), preconditions_{K_M}(r)          (R_O)

caused observed(a₂,2) after a₁, observed(a₁,1).
...
caused observed(aₙ,n) after aₙ₋₁, observed(aₙ₋₁,n−1).
caused observed(end,n+1) after aₙ, observed(aₙ,n).
```

Let $p = \langle a_1, \ldots, a_k \rangle$ be a partial plan for $K_M|_T$. Then, $\exists\ seq = \langle\ [s_0, a_1, s1], \ldots, [s_{k-1}, a_k, s_k] \rangle$ a legal transitions sequence in $K_M|_T$, with s_0 the initial state.

We will strengthen the proof by showing as well that
(i) $\forall s_i \in seq,\ s_i \setminus observed(s_i)$ is the corresponding state in K_M.
where $observed(s_i)$ denotes all the instances of the `observed` fluent appearing in s_i.

By induction over the length of p:

k = 1. $[s_0, a_1, s_1]$ is a legal transition in $K_M|_T$.
Then, $preconditions_{K_M}(r)$ are true in s_0, for at least one of the executability rules r with a_1 as head. In addition, observed$(a_1, 1)$ is true as well in case of a_1 being observable.

(i) holds by construction (step 7) and thus, a_1 is executable in K_M.

The rules above have consequences uniquely in the case of a_1 being observable, in which case, an `observed` instance becomes true and observed$(a_1, 1)$ is falsified in s_1. Since the rest of the encoding remains as in K_M and given that the `observed` fluent, which is the only one introduced by the algorithm, is not part of the set of fluents in K_M, (i) holds for s_1.

p is therefore a partial plan in K_M and (i) holds for s_0 and s_1.

I.H. Lets assume now that for $\langle a_1, \ldots, a_{k-1} \rangle$ the lemma holds.

I.S. Then, for $p = \langle a_1, \ldots, a_k \rangle$, $\exists\ \langle [s_0, a_1, s_1], \ldots, [s_{k-1}, a_k, s_k] \rangle$ such that, in particular, $[s_{k-1}, a_k, s_k]$ is a legal transition in $K_M|_T$. Analogous to the base case, this implies that $preconditions_{K_M}(r)$ are true in s_{k-1}, for some r in the executability rules of a_k. Additionally by (R_O), observed(a_k, k) is true if a_k is observable.

Furthermore, by induction, the sequence $\langle [s'_0, a_1, s'_1], \ldots, [s'_{k-2}, a_{k-1}, s'_{k-1}] \rangle$ is a legal transitions sequence in K_M, with $s'_i = s_i \setminus observed(s_i)$, whereby a_k is executable in K_M.

Thus p is a plan for K_M.

Analogous to the base case, the rules introduced in $K_M|_T$ have only consequences over instances of the `observed` fluent and solely if a_k is observable, thus (i) holds for s_k for the same reasons.

□

Theorem 4.2.1. *A plan for $K_M|_T$ is a plan for K_M .*

Proof. Let *plan* be a plan for $K_M|_T$, then, $\exists \langle\, [s_0, a_1, s1], \dots , [s_{n-1}, a_n, s_n]\rangle$ a legal transitions sequence in $K_M|_T$, with s_0 the initial state, such that $\{$`enabled(`end`)`, `observed(`$end, n+1$`)`$\} \in s_n$.

As a direct consequence of the previous lemma, *plan* is a partial plan in K_M; since in addition $x = s_n \setminus observed(s_n)$ is the corresponding last state in K_M, then `enabled(`end`)` $\in x$. Therefore, *plan* is a plan in K_M. □

Next we will prove that observable activities are present in a plan for $K_M|_T$ iff they are part of the trace T and that the order of appearance of the actions is the same in both.

Theorem 4.2.2. *Let P be a plan for $K_M|_T$. Then:*

(a) $\forall o_i \in \mathcal{O}^{\mathcal{M}}$ *:* $o_i \in T$ *iff* $o_i \in P$

(b) $\langle a_x, a_{x+1}\rangle \subseteq T$ *iff* $\langle a_x, n_1, ..., n_k, a_{x+1}\rangle \subseteq P$*, where for all* $k \geq 0$*,* $j \in [1, k]$*,* n_j *is unobservable.*

Proof.

(a) Let's assume that a is an observable action in P and suppose $a \notin T$. Since $a \notin T$, there is no causation rule in $K_M|_T$ with `observed(`a, i`)` in the head, $i \in \mathbb{N}$ introduced by the algorithm. Provided that a is observable, `observed(`a`, _)` is part of the preconditions of every executability rule with a in the head. Thus, from both facts derives that a is not executable with respect to any state and therefore $a \notin P$, which contradicts the assumption. Hence, $a \in T$.

Assume now a as an action in T[1]. Since P is a plan in $K_M|_T$, `observed(`end`,` $n+1$`)` is true in the last state. By step 4 of the algorithm, this implies that the last action in T was executed and thus recursively each of the previous ones, since no other rule can cause the `observed` fluent become true. Given that $a \in T$, again from 4, there is a causation rule with `observed(`a, i`)` in the head, for some $i \in \mathbb{N}$, which requires the execution of a to reach the `observed(`$end, n+1$`)`. Thus, $a \in P$.

(b) Derived from step 1 of the algorithm, remark first that a_k uniquely identifies an (observable) action by the order in which it appears in a trace, being so that a_k denotes the kth action in it.

Let now $\langle a_x, a_{x+1}\rangle \subseteq T$. Then from (a), $a_x, a_{x+1} \in P$. Assuming that $\langle a_x, n_1, ..., n_k, a_{x+1}\rangle \not\subseteq P$, either:

[1]Recall that by definition all the tasks in a trace are observable

- $\exists n_i,\ i \in [1,k]$, such that n_i is observable, i.e. $\langle a_x, n_i, a_{x+1}\rangle \subseteq P$. In this case, since observable actions require a true `observed` instance to be executable (step 3), then `observed(`n_i, i`)` is true in some state. Because solely `observed(`$a_1, 1$`)` is true at the initial state, there must exist a causation rule with `observed(`n_i, i`)` in the head and a_x in the preconditions. Analogously, `caused observed(`a_{x+1}`,x+1) after` n_i`,` `observed(`n_i`,i)` must be in $K_M|_T$. By construction of $K_M|_T$, this implies that $\langle a_x, n_i, a_{x+1}\rangle \subseteq T$, which is absurd since $\nexists i \in \mathbb{N}$ s.t. $x < i < x+1$.
- a_{x+1} appears before a_x, i.e. $\langle a_{x+1}, ..., a_x\rangle$ is in P; therefrom, there exist causation rules in $K_M|_T$ such that `observed(`$a_{x+1}, x+1$`)` is true before `observed(`a_x, x`)`, thus, by construction (step 4), a_{x+1} appears in T before a_x, which is a contradiction.

Hence, the right hand side of (b) holds.

Let's now assume that $\langle a_x, n_1, ..., n_k, a_y\rangle$ as described in (b) is a subsequence of P and suppose that $\langle a_x, a_y\rangle \nsubseteq T$. Given that by hypothesis and (a) a_x and $a_{x+1} \in T$, either:

- exists at least one a_m such that $\langle a_x, a_m, a_y\rangle \subseteq T$, in which case, due to the previous demonstration, $a_m = n_i$ for some $i \in [1,k]$, which together with the fact that n_i is observable, contradicts the assumption.
- $\langle a_y, \ldots, a_x\rangle \subseteq T$. In this case, $y < x$, then by step 4, there are causation rules such that `observed(`a_y, y`)` is true at some point before `observed(`a_x, x`)`. Due to the rule in step 5, `observed(`a_y, y`)` is negated after the execution of the associated action and it is never set true again. Thus a_y is never executed after a_x and so $\langle a_x, ..., a_y\rangle$ can not be in P.

Therefore, the left hand side of (b) holds as well.

□

To prove completeness, the following theorem is formulated.

Theorem 4.2.3. *Let P_T be a plan for K_M such that $T \subseteq P_T$. Then, P_T is a plan in $K_M|_T$.*

Proof. Let's assume that P_T is not a plan in $K_M|_T$. Then, there is no state in $K_M|_T$ such that `enabled(`end`)` and `observed(`end, i`)` are both true, for $i \in \mathbb{N}$. From the fact that P_T is a plan for K_M derives that `enabled(`end`)` is true for some state in K_N; furthermore, by the proof of lemma 4.2.1, there is a corresponding state in $K_M|_T$ where `enabled(`end`)` is true, thus, `observed(`end, i`)` is never true provided that both fluents are inertial and that there is no rule in $K_M|_T$ that negates `observed(`end, i`)`.

By construction, the rule `caused observed(`end, i`) after` x`, observed(`x`,` $i-1$`)` is in $K_M|_T$, where x is the last action in T. Altogether entails that x

and `observed(`$x, i-1$`)` are not satisfied at the same time, then, either x was not executed, `observed(`$x, i-1$`)` is not reachable or both. Again due to lemma 4.2.1's proof, the preconditions inherited from K_M for x, hold in some state, so, x is not executable because `observed(`$x, i-1$`)` is never true.

This behavior repeats recursively yielding to conclude that `observed(`$a, 1$`)` is never true, with a the first action in T, which is impossible inasmuch as by step 7, it is part of the initial state.

Hence, P_T is a plan in $K_M|_T$. □

Chapter 5

Inclusion of branching predicates

As stated by YAWL's semantics, the determination of which conditions to advance the token into, after the execution of an *OR* or an *XOR split* (decision tasks), is ruled by the evaluation of branching predicates [1] (given in terms of XPath [6] boolean expressions) associated to each of the outgoing arcs. Solely the branches (arcs) whose predicates evaluate to *true* are considered to move on and then the advance proceeds according to the respective construct's semantics.

When not specified, YAWL sets the predicate of an arc to the boolean value *true*. The default configuration consists thus in all the predicates *true*, which in case of an *XOR* derives in the activation of the postcondition corresponding to the first evaluated predicate [2], while in the case of an *OR* has the effect of an *AND split* passing the token to all the post conditions. Also, each of this constructs requires always a default flow, which is taken whenever all the other flow predicates evaluate to false [24].

Branching predicates hinge on the data interaction within the process, captured through the so called *net variables*. The value of the variables is determined at execution time and depending on the source, two types are distinguished:

Internal Variables Those whose value is set by the system itself, by instance flag variables indicating that a task was executed, such as “cardSentByPost" being set to true after executing a “Send card" task or “loggedInSite" being enabled each time a system detects a “Sign in" task.

Exogenous Variables Those whose value comes from external sources, such as an user decision or a webservice. Each variable is either input, output or both of a YAWL task, such task will be considered as the *modifier* task of the variable since is the one where the process gets knowledge about its value (either it is required to or observed from the environment).

[1] Named *branching conditions* in YAWL

[2] The order of evaluation of the predicates can be set in the model

So far we have considered a general behavior in SWF-nets by assuming that predicates are not interrelated in a model, in the sense that each predicate is defined it terms of a single and unique variable. It is very likable though that a variable is involved in more than one predicate, whereby its value assignment influences different evaluations and thus the operation of the process.

Here, we aim to introduce the branching predicates into the $\mathcal{K}$ encoding of the model, for which two aspects need to be taken into account:

- the inclusion of the values assigned to the variables involved in those predicates
- the interaction between the given trace and the model, which until this point have been treated separately.

5.1 Encoding of Decision arcs

Predicates are boolean expressions; as such, they can be seen as propositional formulas, where a propositional variable would represent the evaluation of an atomic predicate, which in the Xpath context corresponds to an expression related to comparison operators, such as $! =$ or $\leq$. Therefore, an atomic predicate will be associated by nature with a fluent in $\mathcal{K}$; remark that the fluent represents indeed the evaluation of the expression, not the expression itself, by instance, a fluent f could be associated to $evaluate(age \geq 18)$.

The encoding simulating an arc through which a decision task S and a condition c are connected, is then determined by the form of the associated branching predicate.

Atomic predicate This is the case where a fluent corresponds to the evaluation of a comparison expression or to any of the boolean constants ($true$ or $false$). The encoding for an arc with an atomic predicate *splittingPred* associated is then:

```
fluents:
    splittingPred.

always:
    inertial splittingPred.
    caused enabled(c) if splittingPred after S.
```

The `inertial` declaration is introduced to propagate, once assigned, the value of *splittingPred*.

Complex predicate By definition a boolean expression in Xpath has the structure [6]

```
OrExpr   ::=   AndExpr | OrExpr 'or' AndExpr
AndExpr  ::=   SimpleExpr | AndExpr 'and' SimpleExpr
```

which basically means that is a formula in disjunctive normal form, where each literal is an atomic predicate as described before[3]. From this definition, a fluent is then associated with a `SimpleExpr` and the branching predicates are of the form

$$P = (f_1^1 \wedge ... \wedge f_n^1) \ \vee \ ... \ \vee \ (f_1^k \wedge ... \wedge f_m^k)$$

for some $k, n, m > 0$ and each f_j^i a fluent.

The encoding in this case is:

```
fluents:
    f_1^1.   ...   f_n^1.
    ...
    ...
    f_1^k.   ...   f_m^k.

always:
    inertial f_j^i.

    caused enabled(c) if f_1^1, ... , f_n^1 after S.
    ...
    caused enabled(c) if f_1^k, ... , f_n^k after S.
```

Beware that this encoding extends the core encoding from section 3.1 and does not replace it, provided that the value of the branching predicates might be unknown when the decision after a split task need to be taken, by instance if the split is the first task and it is unobservable. In this case, the general semantics determine the advance of the token, which can be restricted afterwards if more information about the branching predicates is acquainted.

The execution of exactly one path in the *XOR* case relies on the inherited generic encoding in case of unobservability of the task and in the introduction of the observed value of *splittingPred* otherwise, where the exclusive choice has already been handled by YAWL.

As a simple example to clarify the encoding, consider the extract in figure 5.1. The corresponding $\mathcal{K}$ representation is:

```
fluents: a. b. c. d.
actions: t.
```

[3]Conscious of the fact that $\mathcal{K}$ programs are implemented over disjunctive logic semantics, the expression being in DNF represents an advantage for the encoding process

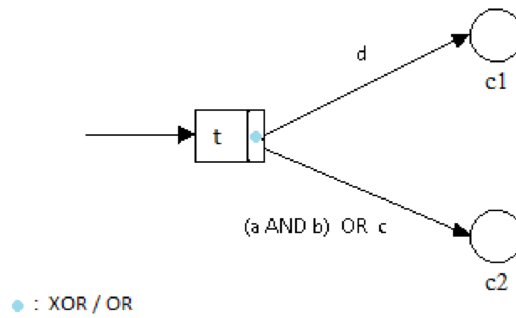


Figure 5.1 Branching predicates

```
always:
   inertial a. inertial b.
   inertial c. inertial d.
   caused enabled(c1) if d after t.
   caused enabled(c2) if a,b after t.
   caused enabled(c2) if c after t.
```

In addition, considering t as an XOR task:

```
caused enabled(c1) if not enabled(c2) after t.
caused enabled(c2) if not enabled(c1) after t.
```

Instead, for t an OR split:

```
total enabled(c1) after t.
total enabled(c2) after t.
forbidden not enabled(c1), not enabled(c2) after t.
```

5.2 Encoding behavior from the model

Each variable appearing in the model is encoded depending on the source of its value.

Case: Internal net variable

For every modifier task t of an internal net variable sP_x, add a rule

caused sP_x after t.

For example `caused` *trainingComplete* `after` *Run_5_km*.

Case: Exogenous net variable

Let sP_x be an exogenous net variable. Then add the corresponding rules, according to the case:

1. sP_x is modified (recall Exogenous Variables) by a non decision task t

   `caused` $modif_sP_x$ `after` t.
   `inertial` $modif_sP_x$.

   where $modif_sP_x$ is a new fluent that has a flag function, indicating when enabled, that the task where the value of sP_x is learned was executed.

2. sP_x is an atomic predicate in the arc connecting a task t with a condition c and does not fulfill (a)

   `caused` sP_x `if enabled(`c`) after` t.

3. sP_x occurs in a complex predicate associated to an arc connecting task t with condition c and does not fulfill neither (a) nor (b).

   `caused` $modif_sP_x$ `if enabled(`c`) after` t.
   `inertial` $modif_sP_x$.

The causation rule in step 2 completes the previous encoding of the arc with atomic predicate, adding the left hand side of the implication: `enabled(`c`)` iff sP_x is true.

5.3 Encoding observed evaluations

The information regarding the evaluation of branching predicates is implicitly encapsulated in the trace records of the observable tasks, being so that the value of fluents linked to branching predicates is introduced to the encoding together with the data concerning the trace.

The following steps extend the trace's inclusion algorithm introduced in section 4, pursuing to add observed data attached to the trace (captured by net variables) and to constraint the computation of plans accordingly.

EXTENSION OF TRACE'S INCLUSION ALGORITHM:

1. For every task t that observes variables sP_1 ... sP_n at step i, add their fluents with the observed value as a postcondition to the already existing causation rule introduced in step 4 of the trace's inclusion algorithm. Then, assuming z to be the action after t in the trace:

   `caused observed(`$z, i+1$`) if` sP_1`, ...,` sP_n `after` t`, observed(`t`,`i`).`

   where each sP_i can be either sP_i or *not* sP_i according to the value in the trace

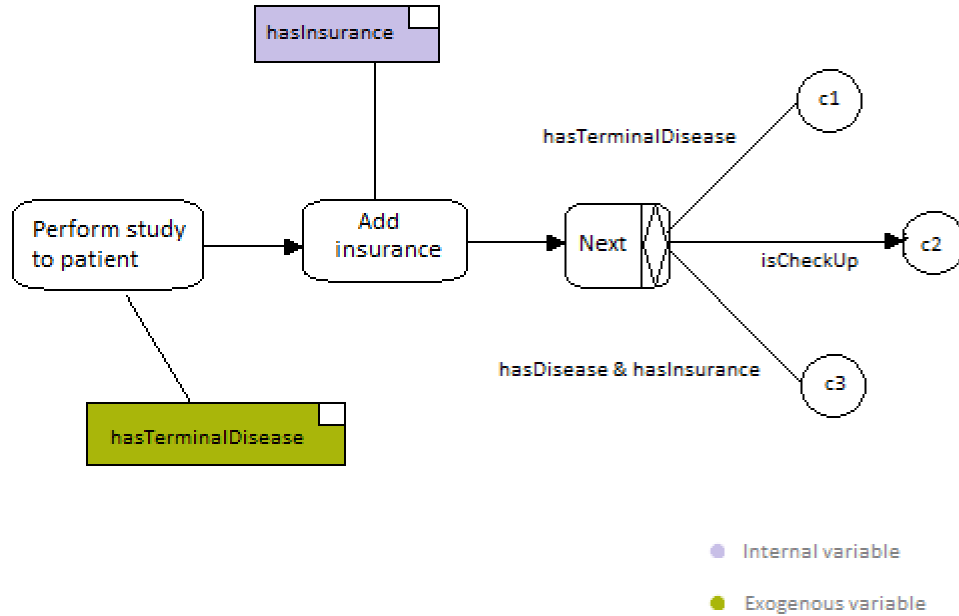


Figure 5.2 Fragment of a hospital process

2. The observed value of a variable from cases 1 or 3 in the case 5.2 of the previous section algorithm, is explicitly caused but conditioned to the execution of the modifier task. Then, considering sP_x observed by task t in step i:

   caused sP_x if $modif_sP_x$ after t, observed(t, i).
   caused $-modifiable_sP_x$ after sP_x.

   where sP_x is in fact a literal, thus it possibly appears as $-sP_x$. The second statement is just to reset the flag fluent once used.

Step 1 guarantees the incorporation of observed evaluations into the encoding by conditioning the execution of the task immediately after the observer in the trace, to fluents having the detected values.

Step 2 asserts the correct selection of paths; for instance if two alternative unobservable activities x and y, from which x is known to learn sP_x's value, are eligible to complete a plan and eventually sP_x is observed to be true in the trace, it allows to infer that, from the two options, x was executed.

Let's consider the fragment in picture 5.2 to illustrate the algorithms. After the performance of a medical study to a patient, an external system informs whether he/she has a terminal disease; the process then adds a mandatory health insurance and the next steps are determined by the clinical state of the patient and his/her purpose of attending the hospital.

Applying the algorithm in section 5.2 to the model: The only internal variable

is set by *Add insurance*, then we add the rule

caused $hasInsurance$ after $AddInsurance$.

$hasTerminalDisease$ corresponds to the case 1 of the exogenous variables, therefore

caused $modif_hasTerminalDisease$ after $PerformStudyToPatient$.
inertial $modif_hasTerminalDisease$.

As well, for the case 2

caused $isCheckUp$ if enabled(c_2) after $Next$.

Instead, $hasDisease$ fits on case 3, thus

caused $modif_hasDisease$ if enabled(c_3) after $Next$.
inertial $modif_hasDisease$.

Encoding now the information from a trace $T := Next[hasInsurance = true, hasDisease = true, \ isCheckUp = false]$, applying step 1 of the extended algorithm

caused observed($end, 2$) if $hasInsurance$, $hasDisease$, not $isCheckUp$
after $Next$, observed($Next$,1).

And from step 2:

caused $hasDisease$ if $modif_hasDisease$
after $Next$, observed($Next$,1).
caused $-modif_hasDisease$ after $hasDisease$.

Chapter 6

Implementation

In order to automatize the reconstruction of traces, fulfilling so the goal established in the introductive chapter, a system implementing the proposed encoding and algorithms was developed. Its general behavior is straightforward from the formulation of the problem and the presented work:

1. Read and parse the process model file given in the specified format
2. Read and parse the trace file in XES format
3. Generate two files containing each:
   - A $\mathcal{K}$ program built by the application of the algorithms for model's encoding and for inclusion of trace and branching predicates
   - The background knowledge extracted from the model as a DLV program
4. Execute the front end system DLV^K with the created files to obtain plans that complete the trace
5. Write the plans in XES format

Diagram 6.1 summarizes the execution sequence integrating the components of the system.

6.1 System overview

The implementation is in Java version 1.6 and requires a distribution of the DLV [14] system. The architecture of the system is depicted in the class diagram of figure 6.2.

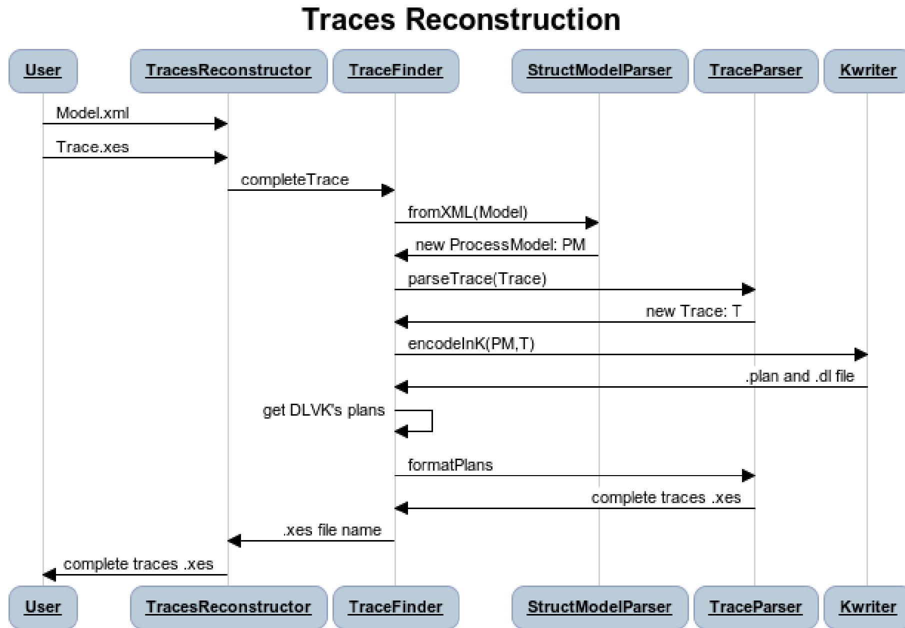


Figure 6.1 Sequence diagram for traces' reconstruction

All the simple objects supporting the system rest in the `basic` package. The `parsing` and `exceptions` packages contain respectively the classes associated to the extraction of model and trace from files, and the exceptions of the system.

The core is implemented in the `encoding` package, where the $\mathcal{K}$ encoding algorithms are performed. The `control` package contains the classes in charge of the interaction among the components and where the main flow described above is executed.

Finally, `ui.TracesReconstructor` is the executable class implementing the command line user interface.

Regarding the search of solutions, *DLV* is a stable state-of-the-art answer set solver for disjunctive extended logic programs (without function symbols) [14]. Taking advantage of their close semantics, a front end for $\mathcal{K}$ has been implemented on top of it, resulting in the DLV^K planner, whose reasoning engine lays over stable models semantics.

A planning problem in $\mathcal{K}$ is a program for DLV^K, therefore, this solver was selected to be integrated in the systems for the generation of plans.

As a last consideration, even though it is an issue of interest, the integration of data extracted from variables is out of the scope of the present thesis. For simplicity thus, the type of the variables associated to the model is assumed to be boolean; this assumption can be easily extended through the implementation of the evaluation of comparison predicates, however, since such functionality is not essential for our primary objective, it will be considered as future work.

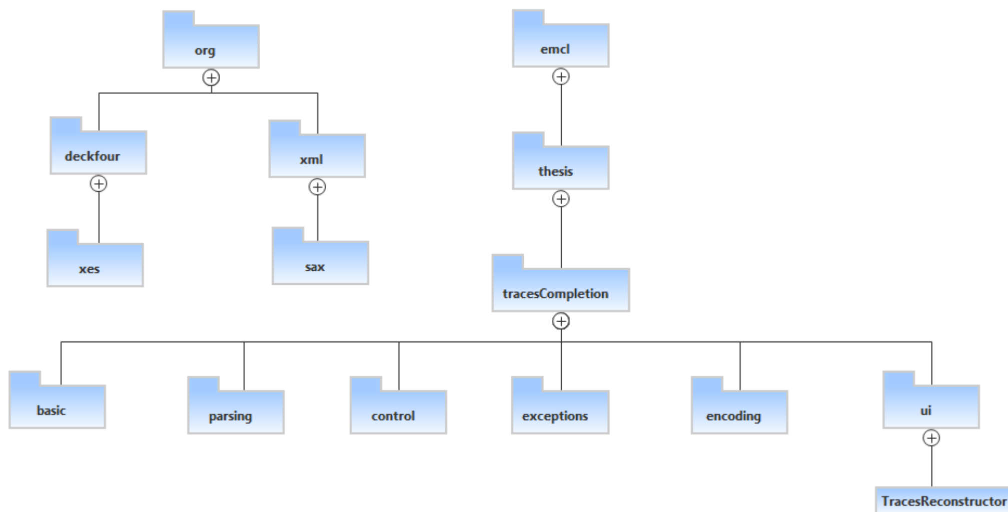


Figure 6.2 System's Packages

6.2 Input files format

XES [16] is an XML-based standard for the representation of event logs, mainly designed for process mining purposes. Since it is widely adopted in such area, this format will be required for the input traces' file. Additional advantages are that, as a standard, it is well documented and many libraries for its integration in particular with Java are available.

The complete documentation can be found in the project site [16]. For illustration purposes though, the next representation belongs to a trace where tasks $Task3$ and $Task6$ were observed and the observation of $Task3$ included the model variables a, b and c.

```
<?xml version="1.0" encoding="UTF-8" ?>
<log xes.version="1.0" xes.features="nested-attributes"
    openxes.version="1.0RC7" xmlns="http://www.xes-standard.org/">
    <extension name="Concept" prefix="concept"
             uri="http://www.xes-standard.org/concept.xesext"/>
       ...

    <global scope="event">
       <string key="concept:name" value=""/>
       ...
    </global>
    <classifier name="Event Name" keys="concept:name"/>
    ...
```

```
    <trace>
         <string key="concept:name" value="1"/>
         <string key="description" value="Example trace"/>
         <event>
            <string key="org:resource" value=""/>
               <container key="Process Data">
                  <string key="a" value="true"/>
                  <string key="b" value="false"/>
                  <string key="c" value="false"/>
                    </container>
                 <string key="concept:name" value="Task3"/>
         </event>
         <event>
            <string key="org:resource" value=""/>
               <container key="Process Data">
               </container>
            <string key="concept:name" value="Task6"/>
         </event>
    </trace>
</log>
```

In relation with the models, considering that this work is based on the assumption of structured workflows, a simplified layout in XML format is proposed for their representation instead of the YAWL's format. The conceiving idea is centered in the inductive definition of SESE blocks.

The DTD specification which files defining models must comply with, is provided in appendix A. As per w3c recommendation, it should be referenced in every model file to verify that it is well formed, minimizing consequently the possibility to reach exceptions at run time.

6.3 Tests and results

A process regarding the registration of a newborn in Italy has been selected to evaluate the performance of the system. This example was created in collaboration with a research group from the Fondazione Bruno Kessler(FBK). The workflow starts when a birth has place, the birth is recorded and when the parents are notified, they proceed to register the child; this can be done either at the municipality or at the hospital itself. Two different flows can occur then depending on the place where the registration took place:

1. Registration at hospital: a log from the Local Registration Office is generated, the data is registered and if the registration has already been done as well in the municipality, then that record is obtained, otherwise, the Local office generates a Receipt.

2. Registration at municipality: a log from the municipality is generated, the data registered and set to the appropriate institutions. A fiscal code can be then generated or not, depending on certain conditions and only if the registration has not been done in the Local Office, a Receipt is generated

The process terminates solely after both registries have been successfully created. The YAWL model representing this process is depicted in figure 6.3.

6.3.1 Settings and test

The only observable actions involved in the process are distinguished with the blue image in the model. The model specification in the format defined in A, as well as one of the tested traces can be found in the appendix B.

The tests were performed in a 64-bit Windows Vista machine with 2 gB of RAM and an Intel Core Duo processor. The generated encoding is also attached in the appendix B.

6.3.2 Results

Different traces were used to verify the behavior of the process in different situations. In every test the obtained plans were aligned to the traces and described valid paths from the input condition to the end.

Both, the standard version and the one considering branching predicates have similar execution times, which depend mainly on the reasoning system (DLV). The length of the solutions though impacts on this time, which however keeps very acceptable considering the size of the model. For plans of length 28, which is the minimal, the approximate execution time is from 2-4 minutes and 7 min for a 35 size plan; a plan of length 40 required though around 40 minutes. In presence of loops the execution of this example took more than 3 hours.

As an observed issue, even though XML is a clean and in this case suitable standard, the specification of models by hand is subject to many errors; a considered future enhancement of the system is therefore the parsing of models created through the use of graphical sources, such as the YAWL editor, increasing consequently the target audience able to use it.

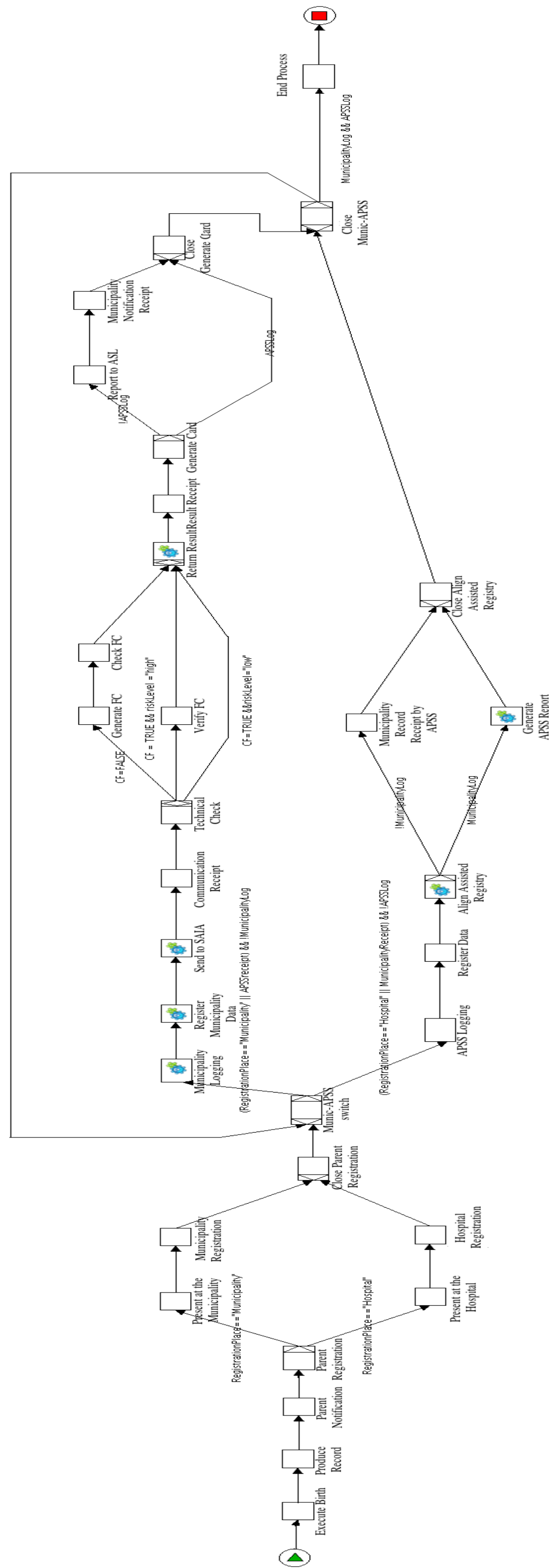


Figure 6.3 Birth Management Process

Chapter 7

Related work

The problem of incompleteness of information associated to process execution traces has been faced mainly in the field of process mining, whose purpose is the analysis of business processes based on event logs; it still represents one of the challenges regarding in particular process discovery and conformance [34]. The goal and preconditions of the present work are however slightly different since a correct model is assumed, while logs derived from monitoring activities can be either incomplete or non-conformant at all.

Regarding process conformance, the alignment between event logs and processes' definitions without [2] and with [8] data, or considering declarative models [9], is approached. These works explore the search space of the set of possible moves, based on the A* algorithm [10], originally proposed to find the path with the lowest cost between two nodes in a directed graph; their aim is to find the best move for aligning the log to the model.

In detail, Adriansyah et al. [2] align Petri Nets and logs identifying skipped and inserted activities in the process model. De Leoni et al. [8] extend the search space by considering also the data values, nevertheless, they are used to weight a cost function for determining the alignment, instead of to drive the reconstruction of the complete trace as in our case.

Other works in the same field have addressed the problem of measuring the extent to which process models capture the observed behavior(fitness). Rozinat et al. [29] use for example missing, remaining, produced and consumed tokens for computing fitness.

The reconstruction of flows of activities of a model, given a partial set of information, can be closely related to several fields of research in which the dynamics of a system are perceived only to a limited extent, being hence needed to reconstruct missing information.

According to their approach, the existing proposals can be divided in those

relying on the availability of a probabilistic model of execution and knowledge (quantitative) and the ones relying on equally likely "alternative worlds" given some observations in time (qualitative).

An instance of the quantitative group is [28], where the authors exploit stochastic Petri nets and Bayesian Networks to recover missing activities, as well as their duration, from process execution traces.

Among qualitative approaches, several works have been developed to efficiently model sets of possible worlds, ranging from tree-based representations (BDDs) [3, 5], to logical formulae whose satisfaction models implicitly represent the worlds [4, 17, 26]. In [4] the issue of reconstructing missing information in execution traces has been tackled as well; their solution however reformulates the problem in terms of a boolean Satisfiability problem (SAT), rather than considering a planning approach and particularly described with an action language.

In [25] a characterization of workflows in terms of labeled transition systems is presented. They consider YAWL as well as the workflows' language; the base of their translation relies though on workflow patterns rather than in the constructs themselves and their goal points to verification of properties of the model. The approach presented in such paper uses fluent linear time temporal logic (FLTL) formulas to capture constraints present on workflows, which can later on be verified through model checking techniques. Alike the proposal here, they consider tasks having a starting and an end event instead of pre and post conditions.

Concerning the employment of automated planning in relation with process models, a similar approach has been developed in [27], where a planning problem is defined in predicate logic terms for the planner IPSS, out from a workflow designed in the tool SHAMASH. The work is however oriented to business process reengineering(BPR), concerning thus about the translation from workflow models into plannning problems so that planning tools semi-automatically generate enhanced business process models. No information about executions is considered.

Planning techniques applied to the creation and reconstruction of process models are also presented in [7], [31] and [22]. In the last by instance, YAWL is customized with *Planlets*, YAWL nets where tasks are annotated with pre-conditions, desired effects and post-conditions, aiming to enable automatic adaptivity of dynamic processes at run-time. The same problem is addressed through the use of continuous planning, in [21], 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. All these papers have a different orientation since their solutions center in the model itself rather than in executions as in the present work.

To the best of our knowledge however, planning approaches have not yet been applied to specifically face the problem of incomplete execution traces.

Chapter 8

Conclusions

The main contribution of the thesis consists in sound and complete algorithms to encode business process models in terms of planning problems, in particular using action languages. Such representation enables among others, the use of reasoning tools based on non monotonic logics, to simulate the behavior of the actual process under the assumption of incomplete knowledge, making thus possible to analyze and learn about the process itself.

YAWL and $\mathcal{K}$ were used to concretize the presented theories, nevertheless, the basis of the techniques relies on the formal semantics of the languages, which at some extent, are the common foundations of most of the process modeling and action languages respectively. Therefore, the core ideas of the algorithms are subject to be applied to other modeling and action languages.

Formal proofs of correctness of the presented algorithms are provided; these algorithms were as well implemented to show and evaluate their feasibility.

Given the similarity of their supporting background, by which workflows can be characterized as transition systems, planning techniques have been applied to tackle different problems in the processes area, such as dynamic recovery or automated verification of processes. The diversity of pursued goals enhances the importance to develop a solid standard formulation that can be adapted to specific objectives, which is the aim of the model encoding algorithm.

A planning approach is particularly suitable to solve the problem of reconstruction of traces, considering the clear intuitive equivalence between the notions of process executions and plans.

By the reason of their desirable properties, the restriction to structured models allows to focus in the semantics of the constructs excluding particular situations originated from the design of the model, such as the presence of deadlocks in blocks opened by an *XOR split* and closed by an *AND join*. Following the presented line the encoding could be though extended; the main challenge consists in the

appropriate handling of loops and the scope of activation of *OR splits*.

Because of the linear complexity of the encoding algorithms, the performance of the system depends on the planning solver's performance. The tests however expose that efficient enough software is available to process real workflows' instances, in this case DLV^K shows to perform well under incomplete knowledge thanks to its non monotonic semantics.

Regarding the limitations of the present work, the data management is still in a preliminary stage; net variables have been considered and tested in a general overview, however, the concrete definition of certain parts and their consequent detailed treatment remain as future work, along with the proof of complexity of the algorithms and the extension to non structured models.

Finally, from the implementation perspective, in order to take advantage of the graphical interface of YAWL, the incorporation of a parser for models described in the YAWL format is as well one of the primary focuses to develop in the future.

Bibliography

[1] *YAWL*. http://www.yawlfoundation.org/. Last checked August 2014.

[2] A. Adriansyah, B. F. van Dongen, and W. M. P. van der Aalst, *Conformance checking using cost-based fitness analysis*, in Proc. of EDOC 2011, 2011, pp. 55–64.

[3] P. Bertoli, A. Cimatti, M. Roveri, and P. Traverso, *Planning in non-deterministic domains under partial observability via symbolic model checking*, in Proc. of the 17th Int. joint conference on Artificial intelligence - Vol. 1, 2001, pp. 473–478.

[4] P. Bertoli, C. Di Francescomarino, M. Dragoni, and C. Ghidini, *Reasoning-based techniques for dealing with incomplete business process execution traces*, in AI*IA 2013: Advances in Artificial Intelligence, vol. 8249 of LNCS, Springer, 2013, pp. 469–480.

[5] A. Cimatti, M. Pistore, M. Roveri, and P. Traverso, *Weak, strong, and strong cyclic planning via symbolic model checking*, Artif. Intell., 147 (2003), pp. 35–84.

[6] J. Clark and S. DeRose, *Xml path language (xpath). version 1.0*, tech. report, 16 November 1999.

[7] C. E. da Silva and R. de Lemos, *A framework for automatic generation of processes for self-adaptive software systems*, Informatica (Slovenia), 35 (2011), pp. 3–13.

[8] M. de Leoni, W. Aalst, and B. Dongen, *Data- and resource-aware conformance checking of business processes*, in Business Information Systems, vol. 117 of LNBIP, 2012, pp. 48–59.

[9] M. de Leoni, F. M. Maggi, and W. M. P. van der Aalst, *Aligning event logs and declarative process models for conformance checking*, in Proc. of BPM'12, 2012, pp. 82–97.

[10] R. Dechter and J. Pearl, *Generalized best-first search strategies and the optimality of a**, J. ACM, 32 (1985), pp. 505–536.

[11] M. Dumas, L. García-Bañuelos, and A. Polyvyanyy, *Unraveling unstructured process models*, in Business Process Modeling Notation - Second International Workshop, BPMN 2010, Potsdam, Germany, October 13-14, 2010. Proceedings, 2010, pp. 1–7.

[12] T. Eiter, W. Faber, N. Leone, G. Pfeifer, and A. Polleres, *Planning under incomplete knowledge*, in Proceedings of the First International Conference on Computational Logic (CL2000), volume 1861 of Lecture Notes in Computer Science, Springer, 2000, pp. 807–821.

[13] T. Eiter, W. Faber, N. Leone, G. Pfeifer, and A. Polleres, *A logic programming approach to knowledge-state planning, ii: The dlvk system*, Artificial Intelligence, 144 (2003), pp. 157 – 211.

[14] T. Eiter, N. Leone, C. Mateis, G. Pfeifer, T. Wien, T. Wien, T. Wien, and F. Scarcello, *The kr system dlv: Progress report, comparisons and benchmarks*, Morgan Kaufmann Publishers, 1998, pp. 406–417.

[15] M. Gelfond and V. Lifschitz, *Action languages*, Electronic Transactions on AI, 3 (1998).

[16] X. group, *Xes homepage*. http://www.xes-standard.org/.

[17] H. A. Kautz and B. Selman, *Pushing the envelope: Planning, propositional logic and stochastic search*, in AAAI/IAAI, Vol. 2, 1996, pp. 1194–1201.

[18] B. Kiepuszewski, A. H. M. ter Hofstede, and C. Bussler, *On structured workflow modelling*.

[19] T. Knowledge System Group, *Dlvk homepage*. http://www.dlvsystem.com/k-planning-system/.

[20] D. L. Kovacs, *Complete bnf description of pddl 3.1 (completely corrected)*, May 2011.

[21] A. Marrella, M. Mecella, and A. Russo, *Featuring automatic adaptivity through workflow enactment and planning*, in CollaborateCom 2011, 2011.

[22] A. Marrella, A. Russo, and M. Mecella, *Planlets: Automatically recovering dynamic processes in yawl*, in OTM Conferences (1), 2012, pp. 268–286.

[23] D. Nau, M. Ghallab, and P. Traverso, *Automated Planning: Theory & Practice*, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2004.

[24] C. Ouyang, *How to manipulate data in yawl*, November 2005.

[25] G. Regis, N. Ricci, N. Aguirre, and T. S. E. Maibaum, *Specifying and verifying declarative fluent temporal logic properties of workflows*, in Formal Methods: Foundations and Applications - 15th Brazilian Symposium, SBMF 2012, Natal, Brazil, September 23-28, 2012. Proceedings, 2012, pp. 147–162.

[26] J. Rintanen, K. Heljanko, and I. Niemelä, *Planning as satisfiability: parallel plans and algorithms for plan search*, Artif. Intell., 170 (2006), pp. 1031–1080.

[27] M. D. Rodríguez-Moreno, D. Borrajo, A. Cesta, and A. Oddi, *Integrating planning and scheduling in workflow domains*, Expert Systems with Applications, 33 (2007), pp. 389–406.

[28] A. Rogge-Solti, R. Mans, W. van der Aalst, and M. Weske, *Improving documentation by repairing event logs*, in The Practice of Enterprise Modeling, vol. 165 of LNBIP, Springer, 2013, pp. 129–144.

[29] A. Rozinat and W. M. P. van der Aalst, *Conformance checking of processes based on monitoring real behavior*, Inf. Syst., 33 (2008), pp. 64–95.

[30] N. Russell, A. H. M. T. Hofstede, and N. Mulyar, *Workflow controlflow patterns: A revised view*, tech. report, 2006.

[31] Y. Shi, M. Yang, and R. Sun, *Goal-driven workflow generation based on AI planning*, in Computer and Computing Technologies in Agriculture IV - 4th IFIP TC 12 Conference, CCTA 2010, Nanchang, China, October 22-25, 2010, Selected Papers, Part III, 2010, pp. 367–374.

[32] A. ter Hofstede and K. B., *A formal analysis of dealock behaviour in workflows*, tech. report, Australia, 1999.

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

[34] W. M. P. van der Aalst and et al., *Process mining manifesto*, in BPM 2011 Workshops, Part I, vol. 99, Springer-Verlag, 2012, pp. 169–194.

[35] W. M. P. van der Aalst and A. H. M. ter Hofstede, *Yawl: Yet another workflow language*, Inf. Syst., 30 (2005), pp. 245–275.

Appendix A

Model description's file format

The Document Type Definition for the files containing descriptions of structured models, accepted as input for the implemented system is specified as follows.

```
<!ELEMENT structuredModel (block)>
<!ELEMENT block (task|block)+>
<!ELEMENT task (predicate|setVar)*>
<!ELEMENT predicate (#CDATA)>
<!ELEMENT setVar (#CDATA)>

<!ATTLIST structuredModel title (#PCDATA) #IMPLIED>
<!ATTLIST block type (simple|seq|parallel|decision|loop) #REQUIRED>
<!ATTLIST block id ID #IMPLIED>
<!ATTLIST task id ID #REQUIRED>
<!ATTLIST task observable xsd:boolean #REQUIRED>
<!ATTLIST task type (A|ANDj|ANDs|XORj|XORs|ORj|ORs) #REQUIRED>
<!ATTLIST predicate toElemId (#PCDATA) #REQUIRED>
<!ATTLIST setVar type (sys|manual) #REQUIRED>
```

Basically a structured model is a (SESE) block and each block is composed by more blocks, tasks or both in the case of structures involving split constructs. The value of the *type* attribute in a *block* element maps each case in the definition of structured workflow 2.2.2:

TYPE	STRUCTURED BLOCK
simple	Atomic task structure
seq	Sequence
parallel	Parallel structure (block with AND constructs)
decision	Decision structure
loop	Structured loop

The following aspects should be considered as well when defining a structured model.

The block *id* attribute is optional except for blocks contained into a decision block, since they need to be referenced by the branching predicates.

For a block where *type* = "*seq*", the order of appearance of inner blocks is assumed to be the order of the sequence.

For blocks that are direct children of a *loop* block, the attribute *position* is mandatory with value either "fwd" or "bwd". This value determines respectively if the block takes the place after the join task (the first in the loop) or if it is the block performed after the decision to do a new iteration.

```
<!ATTLIST block position (fwd|bwd) #REQUIRED>
```

Recall that this last declaration applies uniquely for blocks inside a block with *type* = "*loop*".

The three attributes for a task t are mandatory; *id* is the name of the task. The value of *type* is determined by the construct of t; `A` represents an atomic task and the rest are implicit from the name, where the 's' identifies split tasks and the 'j' join tasks. The value of *observable* determines whether t is observable or not.

The *predicate* element represents a splitting condition for the parent split task. This element can only appear under tasks of type decision. The value of the attribute *toElemId* is the *id* of the element that is reachable whenever the predicate evaluates to true. For tasks closing a block of *type* = "*loop*", the predicate that leads to the exit of the block has *toElemId* = "*exit*".

The content of a predicate is of the form $v_1, ..., v_n| \; ... \; |x_1, ..., x_m$ where the "," stands for a logical "and" and the "|" represents a logical "or".

Finally, the *setVar* element is used to capture the knowledge of a variable being internally modified or getting its value from external sources during the execution of the task where it appears; *type* = "*sys*" is used for internal variables and *type* = "*manual*" for exogenous variables (see 5). The set of *setVar* elements in a task corresponds to the output variables of a task in the specification of a YAWL model.

All the variables involved in the predicates must be declared by the *setVar* element in some task, otherwise they are not recognized as part of the model.

The XML representation of the workflow in figure 3.1 aligned to this format is exemplified next, supposing that variables SP1 and SP2 are set by a user decision.

```
<structuredModel title="simpleExample">
   <block type="decision">
      <task id="s0" observable="true" type="ORs">
         <predicate toElemId="path1">sP1</predicate>
         <predicate toElemId="path2">sP2</predicate>
```

```
        <setVar type="manual">sP1</setVar>
        <setVar type="manual">sP2</setVar>
      </task>
      <block type="seq" id="path2">
           <block type="simple">
              <task id="s1" observable="true" type="A"/>
           </block>
           <block type="simple">
              <task id="s2" observable="false" type="A"/>
           </block>
        </block>
        <block type="loop" id="path1">
           <task type="XORj" id="s3" observable="true" />
           <task type="XORs" id="s4" observable="false">
              <predicate toElemId="s3">back</predicate>
              <predicate toElemId="exit">sP2</predicate>
              <setVar type="manual">back</setVar>
           </task>
        </block>
       <task id="s5" observable="true" type="ORj"/>
    </block>
</structuredModel>
```

Appendix B

Birth Management Process

The model of the process used as a test case is presented here.

```
<structuredModel title="sequenceExample">
<block type="seq">
   <block type="simple">
      <task id="Execute birth" observable="false" type="A"/>
   </block>
   <block type="simple">
      <task id="Produce record" observable="false" type="A"/>
   </block>
   <block type="simple">
      <task id="Parent notification" observable="false" type="A"/>
   </block>
   <block type="decision">
      <task id="Parent Registration" type="XORs" observable="false" >
         <predicate toElemId="M">registryMunicipality</predicate>
         <predicate toElemId="H">registryHospital</predicate>
      </task>
      <block type="seq" id="M" >
         <block type="simple">
            <task id="Present at Municipality" observable="false" type="A"/>
         </block>
         <block type="simple">
            <task id="Municipality Registration" observable="false" type="A"/>
         </block>
      </block>
      <block type="seq" id="H">
         <block type="simple">
            <task id="Present at Hospital" observable="false" type="A"/>
         </block>
         <block type="simple">
            <task id="Hospital Registration" observable="false" type="A"/>
         </block>
      </block>
      <task id="Close Parents Registration" type="XORj" observable="false" />
   </block>
   <block type="loop">
      <task type="XORj" id="MunicAPPSswitch" observable="false" />
      <block type="decision" position="fwd">
         <task type="XORs" id="MunicAPPSstart" observable="false">
            <predicate toElemId="Mlog">registryMunicipality,noMunicLog |
                                       APSSreceipt,noMunicLog</predicate>
            <predicate toElemId="Alog">registryHospital,noAPSSlog |
                                       municipReceipt,noAPSSlog</predicate>
         </task>
```

```
<block type="seq" id="Mlog">
   <block type="simple" >
      <task id="Municipality Logging" observable="true" type="A"/>
   </block>
   <block type="simple" >
      <task id="Register Municipality Data" observable="true" type="A"/>
   </block>
   <block type="simple">
      <task id="Send To SAIA" observable="true" type="A"/>
   </block>
   <block type="simple" >
      <task id="Communication Receipt" observable="false" type="A"/>
   </block>
   <block type="decision">
      <task id="Technical Check" observable="false" type="ORs" >
         <predicate toElemId="cf">notCF</predicate>
         <predicate toElemId="verif">CF,Risk</predicate>
      </task>
      <block type="simple" id="verif">
         <task id="Verify FC" observable="false" type="A"/>
      </block>
      <block type="seq" id="cf">
         <block type="simple" >
            <task id="Generate FC" observable="false" type="A"/>
         </block>
         <block type="simple" >
            <task id="Check FC" observable="false" type="A"/>
         </block>
      </block>
      <task id="Return Result" observable="true" type="ORj" />
   </block>
   <block type="simple">
      <task id="Result Receipt" observable="false" type="A"/>
   </block>
   <block type="decision">
      <task id="Generate Card" type="XORs" observable="false" >
         <predicate toElemId="log">noAPSSlog</predicate>
         <predicate toElemId="GoToClose">APSSlog</predicate>
      </task>
      <block type="seq" id="log">
         <block type="simple" >
            <task id="Report to ASL" observable="false" type="A"/>
         </block>
         <block type="simple">
            <task id="MunicipalityNotificationReceipt"
                  observable="false" type="A"/>
         </block>
      </block>
      <block type="simple" id="GoToClose">
         <task id="Do nothing" observable="false" type="A"/>
      </block>
      <task id="CloseGenerateCard" type="XORj" observable="false" />
   </block>
</block>
<block type="seq" id="Alog">
   <block type="simple">
      <task id="APSSlogging" observable="false" type="A"/>
   </block>
   <block type="simple">
      <task id="RegisterData" observable="false" type="A"/>
   </block>
   <block type="decision">
      <task id="Align Assisted Registry" type="XORs" observable="true" >
         <predicate toElemId="Mrr">noMunicLog</predicate>
         <predicate toElemId="Gmr">municLog</predicate>
      </task>
      <block type="simple" id="Mrr" >
         <task id="MunicipalityRecordReceiptByAPSS" observable="false" type="A"/>
      </block>
```

```
            <block type="simple" id="Gmr">
               <task id="Generate APSS report" observable="true" type="A"/>
            </block>
            <task id="CloseAlignAssisted" type="XORj" observable="false" />
         </block>
      </block>
      <task id="CloseMunicAPSS" type="XORj" observable="false" />
   </block>
   <task id="CloseMunicAPSSloop" type="XORs" observable="false" >
      <predicate toElemId="MunicAPPSswitch">noMunicLog|noAPSSlog</predicate>
      <predicate toElemId="exit">municLog,APSSlog</predicate>
   </task>
  </block>
  <block type="simple">
     <task id="EndProcess" observable="false" type="A"/>
  </block>
</block>
</structuredModel>
```

Additionally, one of the traces:

```
<?xml version="1.0" encoding="UTF-8" ?>
<!-- This file has been generated with the OpenXES library. -->
<!-- XES standard version: 1.0 -->
<!-- OpenXES library version: 1.0RC7 -->
<log xes.version="1.0" >
   <extension name="Lifecycle" prefix="lifecycle"
              uri="http://www.xes-standard.org/lifecycle.xesext"/>
   <extension name="Organizational" prefix="org"
              uri="http://www.xes-standard.org/org.xesext"/>
   <extension name="Time" prefix="time"
              uri="http://www.xes-standard.org/time.xesext"/>
   <extension name="Concept" prefix="concept"
              uri="http://www.xes-standard.org/concept.xesext"/>
   <global scope="trace">
      <string key="concept:name" value=""/>
   </global>
   <global scope="event">
      <string key="concept:name" value=""/>
      <string key="lifecycle:transition" value="complete"/>
   </global>
   <classifier name="Event Name" keys="concept:name"/>
   <classifier name="Resource" keys="org:resource"/>
   <string key="concept:name" value="DEFAULT"/>
   <string key="lifecycle:model" value="standard"/>
   <string key="description" value="Simulated process"/>
   <trace>
      <string key="concept:name" value="1"/>
      <string key="description" value="Simulated process instance"/>
      <event>
         <string key="org:resource" value=""/>
         <string key="lifecycle:transition" value="complete"/>
         <string key="concept:name" value="Align Assisted Registry"/>
         <date key="time:timestamp" value="1970-01-01T01:00:00+01:00"/>
      </event>
      <event>
         <string key="org:resource" value=""/>
```

```
        <string key="lifecycle:transition" value="complete"/>
        <string key="concept:name" value="Generate APSS report"/>
        <date key="time:timestamp" value="1970-01-01T01:00:00+01:00"/>
      </event>
      <event>
        <string key="org:resource" value=""/>
        <string key="lifecycle:transition" value="complete"/>
        <string key="concept:name" value="Municipality Logging"/>
        <date key="time:timestamp" value="1970-01-01T01:00:00+01:00"/>
      </event>
      <event>
        <string key="lifecycle:transition" value="complete"/>
        <string key="concept:name" value="Register Municipality Data"/>
        <date key="time:timestamp" value="1970-01-01T01:00:00+01:00"/>
      </event>
      <event>
        <string key="org:resource" value=""/>
        <string key="lifecycle:transition" value="complete"/>
        <string key="concept:name" value="Send To SAIA"/>
        <date key="time:timestamp" value="1970-01-01T01:00:00+01:00"/>
      </event>
      <event>
        <string key="org:resource" value=""/>
        <string key="lifecycle:transition" value="complete"/>
        <string key="concept:name" value="Return Result"/>
        <date key="time:timestamp" value="1970-01-01T01:00:00+01:00"/>
      </event>
    </trace>
</log>
```


And the background knowledge

```
condition(c32). condition(c33). condition(c30). condition(c31).
condition(c36). condition(c37). condition(c34). condition(c35).
condition(c38). condition(c39). condition(c29). condition(c28).
condition(c27). condition(c22). condition(c21). condition(c20).
condition(c26). condition(c25). condition(c24). condition(c23).
condition(cTo_municAPPSswitch). condition(c6). condition(c5).
condition(c4). condition(c3). condition(c9). condition(c8).
condition(c7). condition(c17). condition(c16). condition(c19).
condition(c18). condition(c1). condition(c2). condition(start).
condition(c0). condition(c11). condition(c10). condition(c13).
condition(c12). condition(c15). condition(c14).

reach(c21,c17). reach(c20,c19). reach(c19,c17). reach(c23,c22). reach(c22,c21).

reachable(X,Y) :- reach(X,Y).
reachable(X,Y) :- reachable(X,Z), reach(Z,Y).
```

The $\mathcal{K}$ program obtained as outcome from the system is:

```
fluents:  enabled(C) requires condition(C).
```

```
         delayed(C) requires condition(C).
         observed(X,Y).

actions: return_Result. parent_Registration. closeGenerateCard. align_Assisted_Registry.
         municAPPSstart. endProcess. execute_birth. check_FC. closeAlignAssisted.
         municipalityNotificationReceipt. generate_APSS_report. municipality_Registration.
         municAPPSswitch. technical_Check. parent_notification. present_at_Municipality.
         verify_FC. produce_record. hospital_Registration. registerData. send_To_SAIA.
         do_nothing. municipalityRecordReceiptByAPSS. communication_Receipt.
         closeMunicAPSSloop.municipality_Logging. report_to_ASL. generate_FC.
         register_Municipality_Data. result_Receipt. present_at_Hospital. closeMunicAPSS.
         aPSSlogging. generate_Card. close_Parents_Registration.

always:
executable technical_Check if enabled(c17).
caused -enabled(c17) after technical_Check.

total enabled(c21) after technical_Check.
total enabled(c19) after technical_Check.
forbidden not enabled(c21),not enabled(c19) after technical_Check.

executable align_Assisted_Registry if enabled(c33), observed(align_Assisted_Registry,_).
caused -enabled(c33) after align_Assisted_Registry.

caused enabled(c35) if not enabled(c37) after align_Assisted_Registry.
caused enabled(c37) if not enabled(c35) after align_Assisted_Registry.

executable send_To_SAIA if enabled(c15), observed(send_To_SAIA,_).
caused enabled(c16) after send_To_SAIA.
caused -enabled(c15) after send_To_SAIA.

executable report_to_ASL if enabled(c26).
caused enabled(c27) after report_to_ASL.
caused -enabled(c26) after report_to_ASL.

executable aPSSlogging if enabled(c31).
caused enabled(c32) after aPSSlogging.
caused -enabled(c31) after aPSSlogging.

executable check_FC if enabled(c22).
caused enabled(c23) after check_FC.
caused -enabled(c22) after check_FC.

executable closeMunicAPSS if enabled(c25),not enabled(c34).
executable closeMunicAPSS if not enabled(c25),enabled(c34).
caused -enabled(c25) after closeMunicAPSS.
caused -enabled(c34) after closeMunicAPSS.

caused enabled(c12) after closeMunicAPSS.

executable register_Municipality_Data if enabled(c14), observed(register_Municipality_Data,_).
caused enabled(c15) after register_Municipality_Data.
caused -enabled(c14) after register_Municipality_Data.

executable municipality_Registration if enabled(c5).
caused enabled(c6) after municipality_Registration.
caused -enabled(c5) after municipality_Registration.

executable verify_FC if enabled(c19).
caused enabled(c20) after verify_FC.
caused -enabled(c19) after verify_FC.

executable municipality_Logging if enabled(c13), observed(municipality_Logging,_).
caused enabled(c14) after municipality_Logging.
caused -enabled(c13) after municipality_Logging.

executable execute_birth if enabled(start).
caused enabled(c0) after execute_birth.
caused -enabled(start) after execute_birth.
```

```
executable communication_Receipt if enabled(c16).
caused enabled(c17) after communication_Receipt.
caused -enabled(c16) after communication_Receipt.

executable parent_Registration if enabled(c2).
caused -enabled(c2) after parent_Registration.

caused enabled(c4) if not enabled(c7) after parent_Registration.
caused enabled(c7) if not enabled(c4) after parent_Registration.

executable hospital_Registration if enabled(c8).
caused enabled(c9) after hospital_Registration.
caused -enabled(c8) after hospital_Registration.

executable do_nothing if enabled(c29).
caused enabled(c30) after do_nothing.
caused -enabled(c29) after do_nothing.

executable municipalityRecordReceiptByAPSS if enabled(c35).
caused enabled(c36) after municipalityRecordReceiptByAPSS.
caused -enabled(c35) after municipalityRecordReceiptByAPSS.

executable generate_FC if enabled(c21).
caused enabled(c22) after generate_FC.
caused -enabled(c21) after generate_FC.

executable result_Receipt if enabled(c18).
caused enabled(c24) after result_Receipt.
caused -enabled(c18) after result_Receipt.

executable produce_record if enabled(c0).
caused enabled(c1) after produce_record.
caused -enabled(c0) after produce_record.

executable endProcess if enabled(c10).
caused enabled(c39) after endProcess.
caused -enabled(c10) after endProcess.

executable close_Parents_Registration if enabled(c6),not enabled(c9).
executable close_Parents_Registration if not enabled(c6),enabled(c9).
caused -enabled(c6) after close_Parents_Registration.
caused -enabled(c9) after close_Parents_Registration.

caused enabled(c3) after close_Parents_Registration.

executable generate_APSS_report if enabled(c37), observed(generate_APSS_report,_).
caused enabled(c38) after generate_APSS_report.
caused -enabled(c37) after generate_APSS_report.

executable present_at_Hospital if enabled(c7).
caused enabled(c8) after present_at_Hospital.
caused -enabled(c7) after present_at_Hospital.

executable municAPPSswitch if enabled(c3),not enabled(cTo_municAPPSswitch).
executable municAPPSswitch if not enabled(c3),enabled(cTo_municAPPSswitch).
caused -enabled(c3) after municAPPSswitch.
caused -enabled(cTo_municAPPSswitch) after municAPPSswitch.

caused enabled(c11) after municAPPSswitch.

executable registerData if enabled(c32).
caused enabled(c33) after registerData.
caused -enabled(c32) after registerData.

executable parent_notification if enabled(c1).
caused enabled(c2) after parent_notification.
caused -enabled(c1) after parent_notification.
```

```
executable closeMunicAPSSloop if enabled(c12).
caused -enabled(c12) after closeMunicAPSSloop.

caused enabled(cTo_municAPPSswitch) if not enabled(c10) after closeMunicAPSSloop.
caused enabled(c10) if not enabled(cTo_municAPPSswitch) after closeMunicAPSSloop.

executable closeAlignAssisted if enabled(c36),not enabled(c38).
executable closeAlignAssisted if not enabled(c36),enabled(c38).
caused -enabled(c36) after closeAlignAssisted.
caused -enabled(c38) after closeAlignAssisted.

caused enabled(c34) after closeAlignAssisted.

executable municipalityNotificationReceipt if enabled(c27).
caused enabled(c28) after municipalityNotificationReceipt.
caused -enabled(c27) after municipalityNotificationReceipt.

executable closeGenerateCard if enabled(c28),not enabled(c30).
executable closeGenerateCard if not enabled(c28),enabled(c30).
caused -enabled(c28) after closeGenerateCard.
caused -enabled(c30) after closeGenerateCard.

caused enabled(c25) after closeGenerateCard.

executable present_at_Municipality if enabled(c4).
caused enabled(c5) after present_at_Municipality.
caused -enabled(c4) after present_at_Municipality.

executable municAPPSstart if enabled(c11).
caused -enabled(c11) after municAPPSstart.

caused enabled(c13) if not enabled(c31) after municAPPSstart.
caused enabled(c31) if not enabled(c13) after municAPPSstart.

executable generate_Card if enabled(c24).
caused -enabled(c24) after generate_Card.

caused enabled(c26) if not enabled(c29) after generate_Card.
caused enabled(c29) if not enabled(c26) after generate_Card.

caused -enabled(c20) after return_Result.
caused -enabled(c23) after return_Result.
executable return_Result if enabled(c20),not delayed(c23), observed(return_Result,_).
executable return_Result if not delayed(c20),enabled(c23), observed(return_Result,_).
caused delayed(Y) if not enabled(Y), reachable(Y,W), enabled(W).
caused enabled(c18) after return_Result.

%trace
caused observed(generate_APSS_report,2)
       after align_Assisted_Registry, observed(align_Assisted_Registry,1).
caused observed(municipality_Logging,3)
       after generate_APSS_report, observed(generate_APSS_report,2).
caused observed(register_Municipality_Data,4)
       after municipality_Logging, observed(municipality_Logging,3).
caused observed(send_To_SAIA,5)
       after register_Municipality_Data, observed(register_Municipality_Data,4).
caused observed(return_Result,6)
       after send_To_SAIA, observed(send_To_SAIA,5).
caused observed(end,7) after return_Result, observed(return_Result,6).

caused -observed(A,N) if observed(_,M), M=N+1 after observed(A,N).

inertial observed(X,Y).
inertial enabled(X).
noConcurrency.

initially: enabled(start). observed(align_Assisted_Registry,1).
goal: enabled(c39), observed(end,6)?
```

Appendix C

Architecture of the implementation

Here we extend the class diagrams of the packages conforming the system.

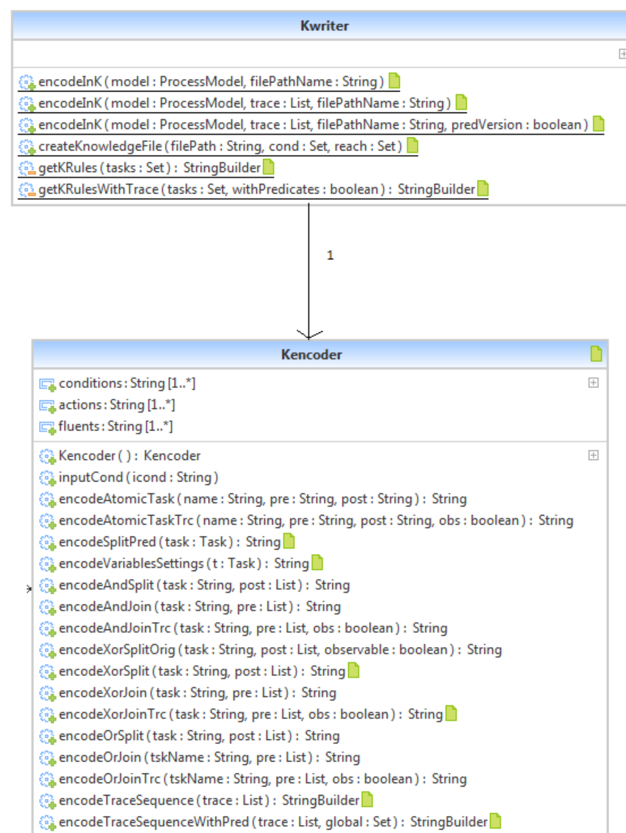


Figure C.1 Package emcl.thesis.tracesCompletion.encoding

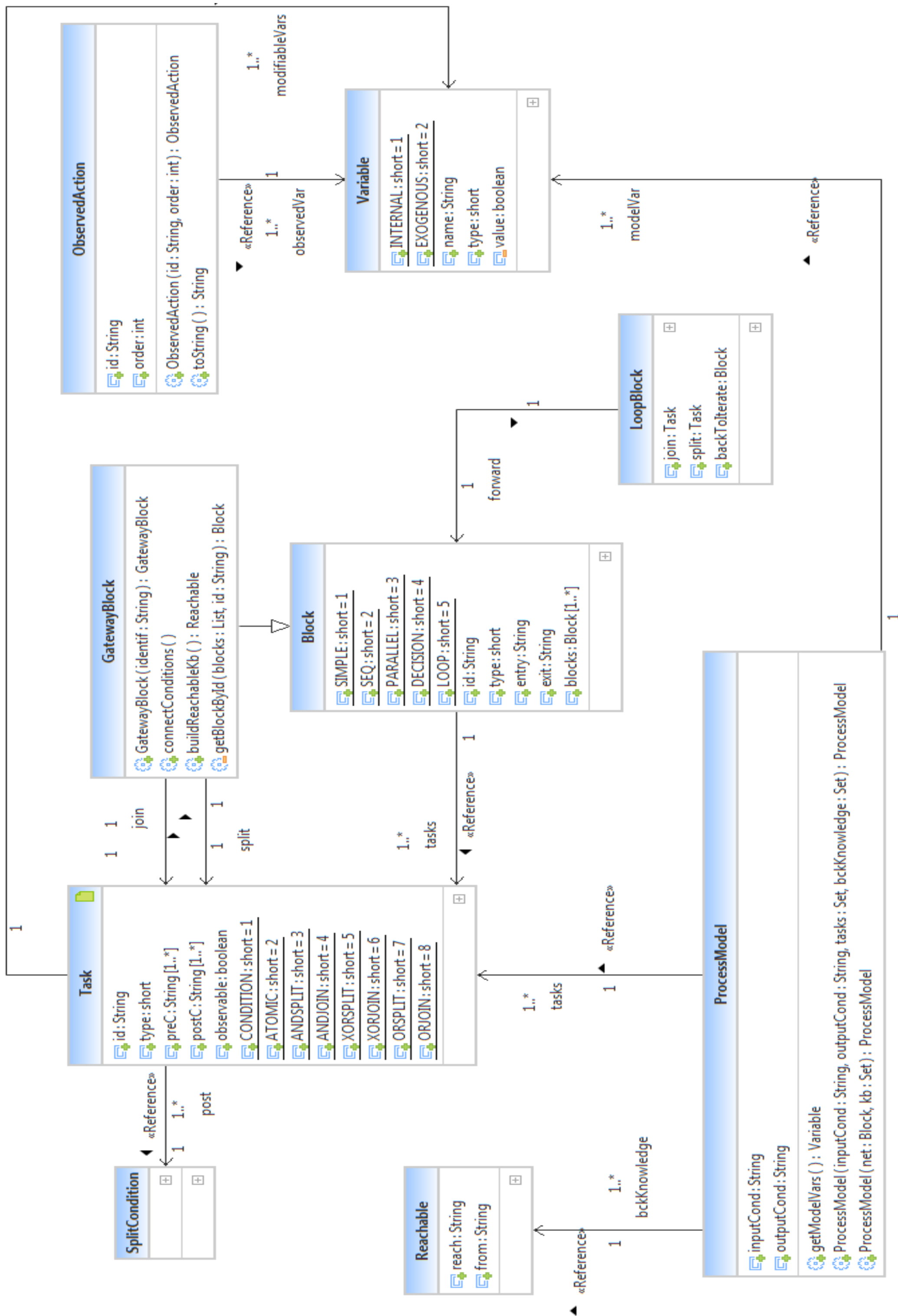


Figure C.2 Package emcl.thesis.tracesCompletion.basic