Specifying and Verifying Declarative Fluent Temporal Logic Properties of Workflows

Germán Regis[1], Nicolás Ricci[1,2], Nazareno Aguirre[1,2], and Tom Maibaum[3]

[1] Departamento de Computación, FCEFQyN, Universidad Nacional de Río Cuarto, Argentina, {gregis, nricci, naguirre}@dc.exa.unrc.edu.ar
[2] Consejo Nacional de Investigaciones Científicas y Técnicas (CONICET), Argentina
[3] Dept. of Computing & Software, McMaster University, Canada, tom@maibaum.org

Abstract. In this paper, we present a characterization of workflows as labeled transition systems, focusing on an encoding of workflow specifications based on workflow patterns. This encoding models tasks in a convenient way, enabling us to exploit fluent linear time temporal logic formulas for capturing typical constraints on workflows. Fluents enable us to flexibly characterize the activities associated with workflow tasks, and also to easily express a wide range of constraints on workflows. Moreover, our characterization of workflows as labeled transition systems, and the use of fluent linear time temporal logic as a language to express workflow properties, allows us to employ model checking for automatically guaranteeing that a property is satisfied by a workflow, or generating violating workflow executions when such property does not hold.
We use YAWL as a language for expressing workflows. Our characterization of workflows as labeled transition systems is implemented in a tool that translates YAWL models into FSP, and then employs the LTSA tool to automatically verify properties of workflows, expressed as fluent linear time temporal logic properties, on the resulting FSP models.

1 Introduction

The importance of efficiency in companies requires constant improvement to their organizational processes. This has led to the need for expressing such processes, typically referred to as *workflows*, and to the proposal of various workflow languages. Indeed, there exist many workflow languages, differing in their degree of formalization (e.g., informal, only with a formal syntax, with a formal syntax and semantics, etc.), their corresponding approaches for workflow description (e.g., declarative or procedural), their expressiveness (e.g., some support advanced conditional routing and some not), their support for automated analysis, etc. An aspect that we consider particularly important is formal semantics. This aspect is crucial for the analysis of models in the language, and is also strongly related to expressiveness, since more expressive languages are more difficult to fully formalize. Furthermore, expressiveness and automation in analysis are typically conflicting aspects, and the design of a good language involves the search of an adequate balance between these aspects. This applies not only to the language

in which a workflow is expressed, but also to the language used for describing declarative properties of a workflow. The importance of declarative properties of workflows is acknowledged by several researchers (see for instance [10, 19, 14, 21]). In particular, in [19] a declarative approach to business process modeling and execution is proposed, where declarative behavioral properties of workflow models are a central characteristic.

In this paper, we present a characterization of workflows as labeled transition systems, focusing on an encoding of workflow specifications based on workflow patterns. This encoding models tasks in a convenient way, enabling us to exploit *fluent linear time temporal logic* (FLTL) [7], to describe declarative behavioral properties of workflow models. As we show later on, fluents enable us to flexibly characterize the activities associated with workflow tasks, and also to easily express a wide range of constraints on workflows. Our characterization of workflows as labeled transition systems has as an additional motivation (besides enabling for the use of FLTL as a language to express properties of workflows) the possibility of using Model Checking [5] for automatically verifying that a workflow satisfies a given property. Thus, our encoding of workflows as labeled transition systems allows us to use FLTL to express properties of workflows, as well as to automatically verify these properties via model checking on the resulting transition systems, generating violating workflow executions when these properties do not hold. This mechanism for the analysis of declarative properties of workflows is very flexible, as opposed to existing tools for workflow analysis that focus on specific properties such as soundness or deadlock-freedom (e.g., the tool in [2]).

Our approach is in essence language independent, and could in principle be applied to any formal workflow language. In this paper, we choose to use YAWL (Yet Another Workflow Language) [2] models to express workflows. YAWL is a powerful workflow language based on the use of workflow patterns [1], that is supported by an open source toolset, and has a formal semantics based on Petri Nets. It is considered an expressive formalism, as various works dealing with its expressiveness in relation to other business process or workflow languages (e.g., Business Process Modeling Notation, Event-Driven Process Chains, etc) demonstrate [9]. Indeed, the use of YAWL allows us to ensure the applicability of our approach to other workflow languages, in many cases via the use of available automated tools mapping other formalisms into YAWL.

The paper proceeds as follows. In the next section, we discuss workflows, and present the use of the YAWL tool for their specification. We then argue about the importance of being capable of expressing properties of workflows, as well as guaranteeing their validity. In section 3 we provide the formal foundations of our work. In section 4, we propose an automated way of encoding YAWL specifications as Finite State Processes (FSP), characterizing tasks as *fluents*. We show how convenient fluents are for expressing behavioral properties, in the context of fluent temporal logic. In order to do that, we develop in detail a case study, taken from the YAWL toolset, whose complexity enables us to illustrate the advantages of the approach. Finally, we discuss related work in the area and draw some conclusions.

2 Business Processes, Workflows and Patterns

In the last decade, several languages and tools have been developed in order to provide an organized view of the structural behavior of systems. One of the main goals of these languages and tools is to provide a setting for describing and analyzing procedural descriptions (workflows) of the activities that take place on the system. As part of these efforts, the Workflow Pattern Initiative was created with the aim of identifying and providing a conceptual basis for business process specifications. This resulted in the specification of a wide range of workflow patterns (control flow, data, resource, etc.), and the development of a formally founded language (and accompanying toolset), known as YAWL [2].

2.1 Workflow Specification using YAWL

Yet Another Workflow Language (YAWL) is a language for modeling workflows. YAWL has a formal foundation based on Petri Nets (PN) [8], and its models are specified using *workflow patterns* [1]. In this paper, we concentrate on control flow patterns; these are composed of tasks, conditions and a flow relation between tasks and conditions. The semantics of a given model is influenced by that of PNs, in the sense that a task is enabled when there are enough tokens in its input conditions, according to the pattern behavior. When a task is executed, it takes tokens out of the input conditions and puts tokens in its output conditions. As opposed to the case of PNs, in a YAWL specification one can connect two tasks directly. A distinguishing feature of YAWL is that it provides direct support for the so-called *cancel region* pattern. This pattern enables one to model situations in which a task can have a cancellation set associated with it. When a task is executed, all tasks in its cancellation set are aborted (i.e., disabled if these were not running, canceled if these were in the middle of a process).

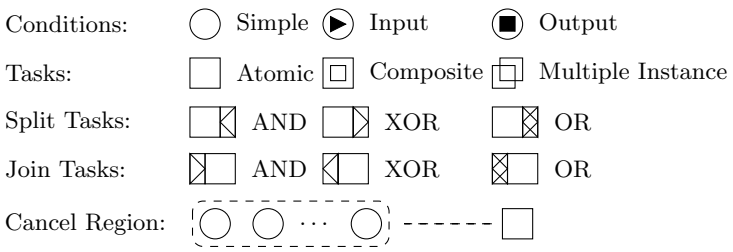

Fig. 1. YAWL Symbols

A workflow specification (control flow perspective) in YAWL is a set of hierarchically organized YAWL nets. Figure 1 shows the symbols corresponding to the elements of the language. A YAWL net is composed of:

- A single *input (start) condition* and a single *output (end) condition*.
- *Tasks*: the language provides three types of tasks, namely: *atomic*, *composite* and *multiple instance*.

- *Atomic* tasks are at the lowest description level of the system.
- *Composite* tasks are associated with corresponding YAWL nets, modeling their behavior. It is assumed that there exists a main YAWL net, which is not associated with any composite task.
- *Multiple instance* (MI) tasks have corresponding lower and upper bounds on the number of instances created when the task is "started up"; MI tasks also have a modifier indicating whether instance creation is *static* or *dynamic* (i.e., indicating whether all instances are created at once, or if these are dynamically created during the execution of the system).

A task T can be related to a *cancel region*, i.e., a set of conditions and tasks that will be aborted when A is completed.

– Specific control flow patterns for the net. The control flow constructs used for pattern definition are those depicted in Fig. 1. Their intended meaning is the following:
  - AND-join: a task associated with this construct starts when all of the incoming branches have been enabled, i.e., all the preceding tasks or associated conditions were completed.
  - OR-join: a task associated with this construct starts when at least one of the incoming branches has been enabled.
  - XOR-join: the associated task starts when exactly one of the incoming branches has been enabled.
  - AND-split: when the incoming branch of this construct is enabled, the thread of control is passed to all of the branches associated with it.
  - OR-split: when the incoming branch of the OR-split is enabled, the thread of control is passed to one or more of the branches following the OR-split, based on the evaluation of conditions associated with each of the outgoing branches.
  - XOR-split: when the incoming branch of the XOR-split is enabled, the thread of control is passed to exactly one of the outgoing branches, based on the evaluation of conditions associated with them.

3 The Formal Framework

3.1 Labeled Transition Systems

Labeled Transition Systems (LTS) are typically used to model the behavior of interacting components [13]. LTS models describe a system as a set of interacting components characterized by states and transitions between them. The transitions represent events in the system, and different components synchronize via shared events. The behavior of the whole system is the result of the parallel composition of its components, understood as the interleaving of the behaviors of the components. Formally, an LTS P is a quadruple $\langle Q, A, \delta, q_0 \rangle$, where: Q is a finite set of states, A is the *alphabet* of P, a subset of the universe *Act* of events; $\delta \subseteq Q \times A \cup \{\tau\} \times Q$ is a labeled transition relation and q_0 is the initial state.

The semantics of an LTS P is its set of *executions*, i.e., the set of sequences of events that P can perform, starting in its initial state and following P's transition relation. For systems with more than a few states, their representation as LTSs becomes impractical. In such situations, a representation of systems as processes in the process algebra FSP, is more convenient [13]. FSP expressions can be automatically mapped into finite LTS, and vice versa.

An FSP specification contains two sorts of process definitions: primitive processes and composite processes. Primitive processes are expressed using event prefix "`->`", choice "`|`" and recursion. Conditional choices can be expressed by means of "`when`" clauses or "`if`" expressions. Both event labels and local process names may be indexed, and primitive processes can be parameterized. As an example, consider the following specification of a simple bounded buffer, and its corresponding LTS.

```
BUFF(N=3) = STATE[0],
STATE[i:0..N] = (
   when (i<N) put[i] ->STATE[i+1]
 | when (i>0) get[i] ->STATE[i-1]).
```

Fig. 2. FSP Buffer specification.

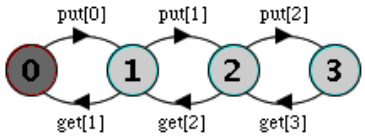

Fig. 3. LTS of the previous buffer.

In Fig. 2, the specification contains a primitive process, parameterized with a bound for the buffer (a default value for the parameter is provided). The possible behaviors of the buffer are specified by means of a primitive process which contains a choice for the two available actions, `put` and `get`. These actions are "multiplied" via indexing, and the resulting behavior is illustrated in the LTS in Fig. 3.

Processes can be composed in a sequential ("`;`") or parallel ("`||`") way. The parallel composition combines the behavior of two processes by synchronizing the events common to their alphabets, and interleaving the remaining events. Continuing with our previous example, consider two processes `PROD` and `CONS`, representing a producer and a consumer, respectively, as specified below. These processes are composed in parallel with a buffer, instantiated with a `Size` (constant declaration); they are synchronized via the relabeling operator "`/`". Relabeling is a relation between actions; in our example, this is used to synchronize all `put` (resp. `get`) actions with the action `produce` (resp. `consume`).

```
PROD = (produce ->PROD).      CONS = (consume ->CONS).
|| BOUNDEDBUFFER = (PROD || BUFF(Size) || CONS)
/{ put[0..Size-1]/produce, get[1..Size]/consume }.
```

3.2 Linear Time Temporal Logic

In order to reason about the behaviors of an LTS, one needs a *logic* in which to express properties of these behaviors. Linear Time Temporal Logic (LTL) [15, 16] is a language that is able to predicate about infinite sequences of states. Each

formula expresses a property of the executions of an LTS. Given a set of atomic propositions $\mathcal{P}$, a well-formed formula is defined inductively using the standard boolean operators and the temporal operators **X** (next) and **U** (strong until), in the following way: *(i)* every $p \in \mathcal{P}$ is a formula, and *(ii)* if ϕ and ψ are formulas, then so are $\neg\phi$, $\phi \vee \psi$, $\phi \wedge \psi$, $\mathbf{X}\phi$, $\phi\mathbf{U}\psi$.

An infinite word $w = x_0x_1x_2\ldots$ over the power set of propositions $\mathcal{P}$ satisfies an LTL formula ϕ, written $w \models \phi$, if the following conditions hold:

- $w \models p \Leftrightarrow p \in x_0$
- $w \models \phi \vee \psi \Leftrightarrow (w \models \phi)$ or $(w \models \psi)$
- $w \models \neg\phi \Leftrightarrow$ not $w \models \phi$
- $w \models \phi \wedge \psi \Leftrightarrow (w \models \phi)$ and $(w \models \psi)$
- $w \models \mathbf{X}\phi \Leftrightarrow w_1 \models \phi$
- $w \models \phi\mathbf{U}\psi \Leftrightarrow \exists i \geq 0 : w_i \models \psi$ and $\forall 0 \leq j \leq i, w_j \models \phi$

where w_1 is the suffix of w resulting from removing the first element in the sequence. The temporal operators **F** (eventually), **G** (always) and **W** (weak until) are defined as follows: $\mathbf{F}\phi \equiv \text{true}\mathbf{U}\phi$, $\mathbf{G}\phi \equiv \neg\mathbf{F}\neg\phi$, and $\phi\mathbf{W}\psi \equiv ((\phi\mathbf{U}\psi) \vee \mathbf{G}\phi)$, where "true $\equiv \phi \vee \neg\phi$".

3.3 Fluent Linear Time Temporal Logic

Fluent Linear Time Temporal Logic (FLTL) is a variant of LTL, that is particularly well suited for describing properties of event-based discrete systems (e.g., LTSs) [13]. Basically, FLTL provides a convenient way of expressing state properties of a labeled transition system, associated with the occurrence of events in the system. More precisely, FLTL extends LTL by incorporating the possibility of describing certain abstract states, called *fluents*, characterized by events of the system. As defined in [17], Fluents are time-varying properties of the world, which hold at particular instants of time if they have been initiated by a triggering event (occurring at some earlier instant in time), and have not been terminated by any terminating event since its initiation. Similarly, a fluent is false at a particular time instant if none of its triggering events ever occurred, or if it has been previously terminated (by one of its associated terminating events) and not yet re-initiated. More formally, $Fl = \langle\{s_1, ..., s_n\}, \{e_1, ..., e_n\}\rangle$*initially* B defines a fluent Fl, where B is a boolean value indicating if the fluent is true or not in the initial state, and $\{s_1, ..., s_n\}$ and $\{e_1, ..., e_n\}$ are disjoint sets of events; when any of the initiating events $\{s_1, ..., s_n\}$ occurs, the fluent starts to be true, and it becomes false again when any of the terminating events $\{e_1, ..., e_n\}$ occurs. If the term *initially* B is omitted then Fl is initially *false*.

LTSA, a tool for the analysis of FSP descriptions, has direct support for fluent-based specifications. Consider as an example the following characterization of the states `full` and `empty`, capturing the obvious associated properties of the bounded buffer:

```
fluent Full = < put[Size-1], get[1..Size]>
fluent Empty = < get[1], put[0..Size-1]> initially True
```

3.4 Model Checking

In the last two decades, the development of algorithmic methods for software and hardware verification has led to powerful analysis mechanisms. One of these is model checking [5]. Model checking provides an automated method for verifying finite state systems, by determining whether or not a property described by a (typically temporal) formula holds on the system's state graph. Various alternative model checking approaches have been proposed, which vary in the representation of the system's state transitions (e.g., explicit state or symbolic), in the logic used for describing properties (e.g., linear time temporal logic, or computation tree logic, etc.), and the language in which systems are actually described (e.g., directly as code in a programming language, or as a model in some more abstract modeling language, etc.). Moreover, tools are available for many of these alternative approaches. In our case, we will use Labelled Transition System Analyzer (LTSA), a verification tool for concurrent systems models. A system in LTSA is modeled as a set of interacting finite state machines. LTSA supports Finite State Process notation (FSP) for concise description of component behavior, and directly supports FLTL property verification. Following the previous examples, we can employ the model checker behind LTSA in order to verify that the buffer cannot simultaneously be empty and full; this is captured by the following FLTL formula: `assert CORRECT_BUFFER = [](!(Full && Empty))`.

4 From YAWL Workflows to Labeled Transition Systems

In this section, we present an encoding of YAWL nets into FSP processes. Basically, this encoding, which is fully automated, will allow us to interpret YAWL (procedural) workflows as FSP processes, and thus we will be able to express properties of workflows, using FLTL formulas over their corresponding encoding in FSP. As we mentioned previously, this encoding will enable us to employ the LTSA model checker for *verifying* behavioral properties of task activities of the business process (BP) specifications. The basic intuition behind the encoding of a YAWL net (control flow perspective) into FSP is the following. A system's behavior is characterized by the occurrence of its tasks. In an abstract way, we can capture a task as an entity having some activity in the system between its *start* and *end* events. So, a trace of these events describes a possible execution of the system. In this way, a system's behavior, i.e., all its possible runs, is captured by the set of all its execution traces. These traces are obviously constrained according to the control flow of the system.

According to our previous observation, it is straightforward to see that a task activity can be captured by means of a *fluent*, becoming *true* when its *start* event takes place, and turning back to *false* when its *end* event task occurs. In order to capture the behavior of the workflow's control flow, we will need to introduce appropriate event synchronizations and process compositions, relating the events related to the tasks that conform the workflow. Once we achieve a characterization of workflows as FSP processes, we can express properties of the

workflows by expressing temporal formulas, employing task-related fluents as the basic ingredient.

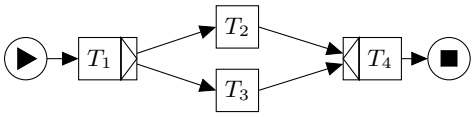

Fig. 4. Simple YAWL net with XOR-split and XOR-join control flow

In order to illustrate the intuition behind our encoding of YAWL into FSP, and our motivation in doing so, let us consider the simple YAWL net shown in Fig.4. According to the YAWL semantics, the set of all possible task occurrences for this net is: $\{T_1T_2T_4, T_1T_3T_4\}$. Each of these corresponds to a trace of events of the system; for instance, [$T_1.start$ $T_1.end$ $T_2.start$ $T_2.end$ $T_4.start$ $T_4.end$] corresponds to the first of the above task occurrences. We will capture the activity of a task straightforwardly via a fluent. For instance, T_2's activity is captured by the fluent $\langle\{T_2.start\}, \{T_2.end\}\rangle$. Now, these fluents can be used in expressing properties of the system's execution, in a declarative way. A basic sample property of the above workflow would be to guarantee that tasks T_2 and T_3 are always run mutually exclusively. This is expressed by the FLTL formula $\mathbf{G}\neg(T_2 \wedge T_3)$.

To formally describe our translation from YAWL into FSP, we consider a formal semantics of YAWL nets [9], given in terms of Reset Petri Nets. Taking into account this semantics, we propose an encoding for tasks and conditions, with a particular treatment for *input* and *output* conditions. For conditions, and due to constraints of finite LTSs (the formalism underlying our approach), we limit their behavior to a *bounded* number of tokens in them. Even though we have this significant limitation, every YAWL model can be encoded as an FSP model. The mismatch between (unbounded) condition tokens and our intrinsically bounded setting will be reflected when analyzing properties of workflows, via false positives reporting deadlocks. However, the analysis is still *conservative*: if no violations to a property are detected, then it is guaranteed that no violations exist.

It is worth mentioning that FSP supports nondeterministic choice, and therefore branching constructs such as non-free choice and deferred choice can be faithfully captured. Also, since our property language is FLTL, there is no need to consider a branching semantics for our processes (nor a bisimulation semantics) for the purpose of property verification: all possible executions (i.e., all possible interleavings of parallel processes) are taken into account by the model checking tool, thus exhaustively covering all behaviours of the system.

In order to represent a net behavior, we specify how to compose tasks and conditions. In this composition we consider the control flow operators associated with the tasks of the net, and provide an encoding for them. Finally, we address especially sophisticated elements of YAWL nets, such as *cancel regions* and *composite tasks*. Multiple instance tasks are simply treated as abbreviations of nets composed of as many instances as the tasks indicate. The dynamic evolution

of multiple instance tasks is characterized via sequential compositions and OR operations.

Definition 1. *A YAWL net is a tuple* $(nid, C, i, o, T, T_A, T_C, M, F, Split, Join, Default, Rem, Nofi)$ *where:*

- *nid is the unique identification of the YAWL net.*
- C *is a set of conditions,* $i \in C$ *and* $o \in C$ *are the input (start) and output (end) conditions, respectively;*
- T *is a set of tasks;* $T_A \subseteq T$ *is the set of atomic tasks, and* $T_C \subseteq T$ *is the set of composite tasks.* $M \subseteq T$ *is the set of multiple instance tasks;*
- $F \subseteq (C \backslash \{o\} \times T) \cup (T \times C \backslash \{i\}) \cup (T \times T)$ *is the control flow relation; every node in the graph* $(C \cup T, F)$ *is on a directed path from* i *to* o*;*
- $Split : T \nrightarrow \{AND, XOR, OR\}$ *specifies the split behavior of each task;*
- $Join : T \nrightarrow \{AND, XOR, OR\}$ *specifies the join behavior of each task;*
- $Default \subseteq F$ *denotes the default arc for the OR-Split, ensuring that at least one outgoing arc is enabled;*
- $Rem : T \nrightarrow \mathbb{P}^+(T \cup C \backslash \{i, o\})$ *specifies the tokens to be removed and the tasks that should be canceled as a consequence of an instance of the task completing its execution;*
- $Nofi : M \to \mathbb{N} \times \mathbb{N}^{inf} \times \mathbb{N}^{inf} \times \{dynamic, static\}$ *specifies the configuration of multiple instance tasks: lower and upper bounds, the threshold for continuation, and its creation's behavior.*

Let N be a YAWL net. The process representing the *input*(i) and *output*(o) conditions, starting and ending N, is the following:

```
YNET = (i_cond ->o_cond -> YNET).
```

For each $t \in T_A$ (atomic task), we generate an FSP process characterizing its *start* and *end* events:

```
TASK = (start ->end -> TASK).
```

As mentioned before, the encoding of conditions are restricted to a bounded number of tokens. With this limitation, we represent the conditions in a way similar to a bounded buffer, but with two parameters indicating the possible input and output connections. The bound for tokens is the amount of input connections given by default.

```
CONDITION (IN=2,OUT=2) = STATE[0], STATE[i:0..IN] =
(when(i<IN) in[i:1..IN]->STATE[i+1] |when(i>0) out[j:1..OUT]->STATE[i-1]).
```

The input/output connections are encoded as the *in* and *out* actions, and we refer to them as *ports*. Let $tsk_1, tsk_2 \in T_A \wedge (tsk_1 \notin Dom(Split) \wedge tsk_2 \notin Dom(Join))$ be atomic tasks of N, without split and join decorations, respectively; let $c \in C \backslash \{i, o\}$ be a condition with n and m input and output ports, respectively. In order to compose tsk_1 and tsk_2, we have:

– Sequential composition of tsk_1 and tsk_2: achieved by synchronizing $tsk_1.end$ and $tsk_2.start$, by means of relabeling.

```
|| SYSTEM = tsk[1..2]:TASK /{tsk[2].start/tsk[1].end}
```

– Composition of tsk_1 with tsk_2 through c (condition in between two tasks): achieved by connecting the finalization of tsk_1 with some input port of c, and the start of tsk_2 with some output port of c.

```
|| SYSTEM = tsk[1..2]:TASK || c:CONDITION(n,m)
/{c.in[i]/tsk[1].end, tsk[2].start/c.out[j]}
```

where $1 \leq i \leq n$ and $1 \leq j \leq m$.

– Composition with decorations: Consider $T \in T_A \wedge (T \in Dom(Split) \vee T \in Dom(Join))$, i.e., T is an atomic task with some *and* or *join* decoration (AND, OR, XOR). Let us call these decorations *gates*. For each possible gate, we generate a process according to its behavior. These processes are parameterized by the corresponding input and output *ports* (e.g., the process corresponding to a *join* gate may have $2..n$ input ports and only one output). As shown in Fig. 5, if T has some join (j) or split (s) gate associated, the interconnection between T and the gates will be achieved by the synchronization of $J.out$ with $T.start$, and $T.end$ with $S.in$, respectively. Let us consider tsk_1, tsk_2 to be tasks of the system. In order to compose tsk_1 with tsk_2 through T, we synchronize $tsk1.end$ with some input port of J, and $tsk2.start$ with some output port of S.

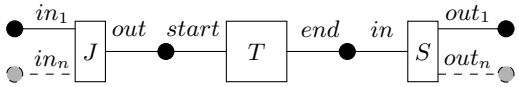

Fig. 5. Task gates configurations.

In order to model the task occurrences in the system, for every task T_i we define a *fluent* of the form $T_i = \langle \{tsk_i.start\}, \{tsk_i.end\} \rangle$. This fluent predicates that T is active between the occurrences of its *start* and *end* events. As an example, the encoding for the YAWL net of Fig. 4 as an FSP process is the following:

```
|| SYSTEM=(YNET ||tsk[1..4]:TASK ||xors: XOR_SPLIT(2) ||xorj:XOR_JOIN(2))
/{ TSK[1].start/i_cond, TSK[4].end/o_cond,
xors.in/TSK[1].end, TSK[2].start/xors.end[1], TSK[3].start/xors.end[2],
xorj.in[1]/TSK[2].end, xorj.in[2]/TSK[3].end, TSK[4].start/xorj.out }.
fluent T[i:1..4] = <{tsk[i].start}, {tsk[i].end}>
```

4.1 Encoding of Gates

For each kind of gate we will generate a corresponding FSP process capturing its behavior. These processes are parameterized by input and output ports. Due to space limitations, we present the encodings only for some gates.

For *AND-split*, *XOR-split* and *OR-split*, the FSP processes are characterized by one input port and $N > 1$ output ones. The processes will be parameterized with N and their encodings depend on the corresponding behavior, e.g., for the AND-split we generate a process of the form:

```
AND_SPLIT_TRIGGER(N=1) = (in ->out[I] ->ANDSPLIT_TRIGGER).
|| AND_SPLIT(N=2) = ( forall [i:1..N] ANDSPLIT_TRIGGER(i)).
```

The `AND_SPLIT` process triggers as many *out* actions as specified by the parameter which shares the *in* action (`forall` is an abbreviation for parallel "||" composition). When the *in* action occurs, *all* the *out* are made available, i.e., the control (token) is passed to all connected output tasks or conditions.

Notice that, in the XOR and OR split gates, we use state variables in order to encode the corresponding guard conditions. Due to restrictions in the datatypes supported by LTSA, we only consider integer and boolean types. The choices for the out ports are constrained by formulas involving those variables, used as conditional *when* clauses in the obvious way.

The *XOR-join* encoding is simply a choice over its incoming events; once one of them arrives, the outgoing event must occur: `XOR_JOIN (N=2) = in[1..N] -> out -> XOR_JOIN).`

OR-join: Due to its non local semantics, this kind of gate has different interpretations across different business process specification languages. In [9], there is a survey of the *OR-join* semantics in Business Process Modeling Notation (BPMN), in Event-driven Process Chains (EPCs) (see also [11]), etc., and the complications in the analysis of these gates in the presence of cancel regions, loops, or multiple instances. In YAWL, the evaluation of the gate in order to determine if an OR-join can be fired is made via *backward firing* and *coverability* analyses in reset nets. The encoding of the OR-join gates employed to perform our analysis of the models mimic the informal semantic of the OR-join (cf. [9], p. 104), that prioritizes all possible incoming events before firing the out port. In order to encode this gate, the following process is generated:

```
OR_JOIN(N=2) = OR_JOIN_DEF[0], OR_JOIN_DEF[b:0..1] =
               ( in[1..N] -> OR_JOIN_DEF[1] | when (b!=0) out ->OR_JOIN ).
```

where all incoming events are "listened to", and if at least one of them is activated, the outgoing event will be fired. The priority on accept incoming events before firing the output is encoded by means of the *priority* operator of FSP, giving *lower* (>>) priority to the *out* action.

4.2 Cancel Regions and The Encoding of Composite Tasks

Cancel Regions provide the ability of disabling a set of tasks in a process instance. If any of the tasks belonging to this region is already being executed (or is currently enabled), then they are immediately terminated. Cancelation captures the interference of an activity in the execution of others. In order to model this interference in YAWL, a *canceling task* can be associated with a cancel region,

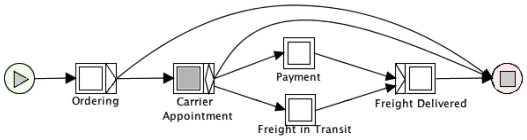

Fig. 6. YAWL net corresponding to the Order Fullfilment Process.

indicating a set of tasks or conditions to be canceled. In order to encode cancel regions in FSP, first we consider an extended version for the encoding of tasks and conditions belonging to a *cancelable region*. For these tasks, instead of the original FSP process, we define a process representing a task that *starts* and, either the task *ends*, or the task can be *canceled*. In a similar way, we define the processes corresponding to the cancelable conditions. In this setting, the *start* action of a canceling task is synchronized with the *cancel* actions of the canceled tasks and conditions.

```
CANCELABLE_TASK = CTASK_INACTIVE,
CTASK_INACTIVE = ( start ->CTASK_ACTIVE | cancel ->CTASK_INACTIVE ),
CTASK_ACTIVE = ( end ->CTASK_INACTIVE | cancel ->CTASK_INACTIVE ).
```

For systems involving *composite tasks*, each of these tasks will have an associated YAWL net specifying its corresponding behavior. So, in order to encode the system, we generate a process CT_i for each net associated with a composite task, following the above procedure. Then, in the net encoding corresponding to the main system, we declare an instance of each CT_i, and we connect them synchronizing their *i_cond* and *o_cond* with the corresponding input and output task or condition. Finally, the activities corresponding to the composite tasks are defined by fluents whose logical values depend on the occurrence of their *i_cond* and *o_cond* actions in the expected way. Note that we can specify the activity of a task t_i belonging to a composite task CT_k on the main process, prefixing the task with the name of the composite task, i.e., $CT_k.t_i$.

5 Case study

We take a case study accompanying the YAWL tool, that we consider to be a complex and complete model, involving all kinds of components of the YAWL language. The sources of the YAWL model can be downloaded[4]. The case study describes the process of order fulfillment followed in a fictitious company, which is divided into the following phases: ordering, logistics (which includes carrier appointment, freight in transit, freight delivered), and payment. The order fulfillment process model is shown in Fig. 6, where each of the above phases is captured by a composite task. Due to space limitations, we only explain in more detail one of the subtasks, the Carrier Appointment process. The YAWL model corresponding to the CA is show in Fig. 7. Basically, the model specifies that after confirmation of a Purchase Order on the previous phase, a *route guide* needs

[4] `http://www.yawlfoundation.org`

to be prepared and the *trailer usage* needs to be estimated. These operations are performed in parallel. If either task takes too long (calculated by the task *Carrier Timeout*), a timeout is triggered, which leads the termination of the overall process. If not, the task *Prepare Transportation Quote* takes place, by establishing the cost of shipment. After this task, a distinction is made among shipments that require a *full truck load (FTL)*, those that require *less than a truck load (LTTL)* and those that simply concern a *single package (SP)*. In order to simplify the view of the model, we depict FTL and LTTL as composite tasks. After the FTL and LTTL, there are subsequent opportunities to modify the appointments information until a *Shipment Notice* is produced; after that, the freight can be picked up. For SP the process is straightforward.

The encoding of YAWL specifications into FSP processes is fully automated, and a tool called YAWL2FSP was developed for this task. This tool is publicly available[5]. The FSP specification was automatically generated and the resulting LTS for the complete Order Fulfillment net (58 tasks, 30 gates, 36 conditions, 2 cancel regions) was generated in 0.298 seconds, using 28,96 Mbytes of memory, with the tool LTSA. The LTS contains 13164 states and 59722 transitions. The analysis for the system was performed in two phases. First, we verified properties over tasks based on the templates published in the Declare Tool [9], including precedence, non-coexistence, response, etc. Next, and taking advantage of the *fluent* characterizations and FLTL expressiveness, we verified properties of the system involving "sub-traces" of the execution, e.g. activities of a subtask, or properties where the desired behavior is characterized by the occurrence of a disjoint set of events. Due to space limitations, we only report here some of the most relevant properties, and show how these are captured in FLTL:

1. *If timeout occurs in CA, then no shipment notice can be produced.*

```
assert PROPERTY_1 = (CarrierTimeout ->!ProduceShipmentNotice)
```

   Notice that this property uses two fluents, that capture the execution of corresponding atomic actions. The previous section describes the details on how these fluents are defined; for instance, for CarrierTimeout, the fluent is: `fluent CarrierTimeout = <C_A.task[5].start,C_A.task[5].end>`, where `C_A` references the Carrier Appointment net, and `task[5]` represents the FSP process id corresponding to the CarrierTimeout task.
2. *Tasks belonging to different ways of transportation cannot occur simultaneously.* To capture this property we define three fluents, corresponding to the whole activity of the FTL, LTTL or SP ways of transportation. For example, `FullTruckLoad=<C_A.ftl.i_cond,C_A.ftl.o_cond>`, where `ftl` is the FSP id of the translated sub-net, and `i/o.cond` are the *initial* and *end* events, respectively. The property is specified as:

```
assert PROPERTY_2 = !( (FullTruckLoad && LessThanTruckLoad)  ||
       (FullTruckLoad && SinglePackage)           ||
       (SinglePackage && LessThanTruckLoad) )
```

[5] `http://sourceforge.net/projects/yawl2fsp/`

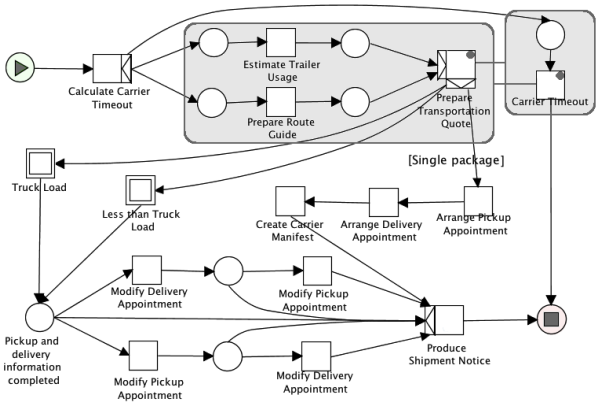

Fig. 7. YAWL net corresponding to Carrier Appointment Process.

3. *If Shipment Notice was produced, necessarily a delivery and pickup appointment were arranged.* We define two fluents, characterizing the arranging appointment activities, i.e., pickup and delivery. These fluents will be activated by any of the corresponding activities of the three possible ways of transportation (FTL, LTTL or SP). Here we can appreciate the flexibility of fluents in order to describe abstract states in the model. As example consider the fluent corresponding to `Delivery Appointment Arranged`, which is enabled by the occurrence of an event corresponding to the main net and events of the FTL and LTTL sub-nets: `<{C_A.task[7].end,C_A.ftl.task[2,3,7].end, C_A.ltl.task[2,5].end},C_A.o_cond>` . The property is expressed as follows:

```
assert PROPERTY_3 = ( ShipmentNoticeProduced ->
       (DeliveryAppointmentArranged && PickupAppointmentArranged))
```

The time consumption associated with the verification of the above properties was: (1) 154ms, (2) 152ms and (3) 185ms, and the memory consumption (1) 11.8MB, (2) 11.9MB and (3) 16.6MB. The encoding and verification were performed using an Intel Core 2 Duo 2.2 Ghz processor, 2 GB 667 Mhz DDR2 SDRAM memory and a Unix based Operating System. Although we are unable to provide a fully-developed example due to space limitations, it is important to notice that in case some property does not hold, the model checker underlying LTSA would provide a trace reproducing the erroneous behavior of the system; this is extremely useful information, that is normally used in order to correct the model, or the corresponding workflow.

6 Related Work and Conclusions

The formal specification and verification of business processes has gained relevance in the last decade, not only in academic settings but also, and most importantly, in industry, where business process optimization is a crucial task. Various languages and methods for business process description have been proposed, most of which were initially informal, but for which different formal characterizations have been proposed. In [18] a general survey for BP specification

can be found and in [4] a survey of formalizations for the Business Process Execution Language (BPEL) is analyzed. Other formal approaches include that presented in [20], where a semantics based on timed automata is proposed for the verification of Product Process Modeling Language (PPML) models. Since our work is essentially a formalization of workflows in terms of labeled transition systems, there exist some relevant related works; in particular, in [10] an automata-based method for formalizing workflow schemas is proposed, but the approach suffers from expressive power limitations, in relation to YAWL (beyond our bounded condition tokens limitation).

We have presented an encoding of YAWL (procedural) workflows into FSP processes. This encoding, which can be performed automatically and has been implemented in a tool, models tasks in a convenient way, enabling us to exploit fluent linear time temporal logic formulas for capturing typical constraints on workflows, and to use the model checker behind LTSA for verifying such constraints. The encoding adequately maps YAWL constructs to FSP elements, in order to make intensive use of *fluents*, in particular to capture workflow tasks, and their properties. Workflows, and in particular those based on a control-flow pattern, are inherently event-based, and thus using state-based formal languages such as LTL makes it more complicated to express declarative properties. FLTL, on the other hand, allows one to more naturally describe execution situations in workflows, via abstract activating/disabling events, as our encoding and examples in this paper illustrate.

We are currently conducting some experiments regarding a comparison of ease of use of LTL vs. FLTL for the specification of properties of workflows. In this respect, our work is twofold: we are working on a tool for automatically translating Declare constraint models to FLTL formulas, in order to verify those constraints over a procedural YAWL workflow, and we are developing a front-end (graphical tool) to assist the end user in the description of properties via FLTL and to represent violation executions when counterexamples are reported.

We have chosen to base our work on YAWL because it has a formal foundation, and it supports a wide range of workflow patterns, providing an expressive environment for BP specification. As we mentioned, the YAWL toolset provides the verification of some properties of workflows such as soundness and deadlock-freedom [3], but it does not provide a suitable flexible language for declaratively expressing other behavioral properties of its models. In this respect, the Declare tool might be applicable, but only to monitor executions of YAWL models, or analyzing the consistency of different declarative, linear temporal logic, constraints on a procedural YAWL workflow. In particular, Declare does not provide features for the verification of properties of executions. In this aspect, works closer to our approach are those presented in [12, 6], where the SPIN model checker is used to automatically verify properties of YAWL models. However, in these works, standard LTL is employed as a property language, which is better suited for state-based settings but less appropriate for event-based frameworks, as is the case of workflow descriptions [7].

Acknowledgements. The authors would like to thank the anonymous referees for their helpful comments. This work was partially supported by the Argentinian Agency for Scientific and Technological Promotion (ANPCyT), through grants PICT PAE 2007 No. 2772 and PICT 2010 No. 1690, and by the MEALS project (EU FP7 programme, grant agreement No. 295261).

References

1. W. M. P. van der Aalst, A. H. M. ter Hofstede, B. Kiepuszewski, A. P. Barros, *Workflow Patterns*, Distributed and Parallel Databases, vol.14, pp. 5-51, 2003.
2. W. M. P. van der Aalst and A. H. M. ter Hofstede, *YAWL: yet another workflow language*, Inf. Syst., vol.30, p.p. 245-275, 2005.
3. W. M. P. van der Aalst et al. *Soundness of workflow nets: classification, decidability, and analysis*, Formal Asp. Comput., vol. 23, num. 3, pp. 333-363, 2011.
4. F. van Breugel and M. Koshkina. *Models and Verification of BPEL*. http://www.cse.yorku.ca/˜franck/research/drafts/tutorial.pdf, 2006.
5. E. Clarke, O. Grumberg and D. Peled, *Model Checking*, MIT Press, 2000.
6. F. Rabbi, H. Wang, W. MacCaull, *YAWL2DVE: An Automated Translator for Workflow Verification*, SSIRI, pp. 53-59, 2010.
7. D. Giannakopoulou and J. Magee, *Fluent model checking for event-based systems*, ESEC / SIGSOFT FSE, pp. 257-266, 2003.
8. C. Girault, R. Valk, *Petri Nets for Systems Engineering: A Guide to Modeling, Verification, and Applications*, Springer-Verlag, 2002.
9. A. H. M. ter Hofstede, W. M. P. van der Aalst, M. Adams, N. Russell, *Modern Bussiness Process Automation*, Springer, 2010.
10. C. T. Karamanolis, D. Giannakopoulou, J. Magee and S. M. Wheater, *Model Checking of Workflow Schemas*, EDOC, pp.170-181, 2000.
11. E. Kindler, *On the semantics of EPCs: Resolving the vicious circle*, Data Knowl. Eng., vol. 56, num. 1, pp. 23-40, 2006.
12. N. Leyla, A. S. Mashiyat, H. Wang, W. MacCaull, *Towards workflow verification*, CASCON, pp. 253-267, 2010.
13. J. Magee and J. Kramer, *Concurrency: State Models and Java Programs*, John Wiley & Sons, 1999.
14. F. M. Maggi, M. Montali, M. Westergaard W. M. P. van der Aalst, *Monitoring Business Constraints with Linear Temporal Logic: An Approach Based on Colored Automata*, BPM, pp. 132-147, 2011.
15. Z. Manna and A. Pnueli, *The Temporal Logic of Reactive and Concurrent Systems - Specification* -, Springer, 1991.
16. Z. Manna and A. Pnueli, *Temporal Verification of Reactive Systems -Safety-*, Springer, 1995.
17. R. Miller and M. Shanahan, *The Event Calculus in Classical Logic - Alternative Axiomatisations*, Linkoping Electronic Articles in Computer and Information Science, Vol. 4, No. 16, pp. 1-27, 1999.
18. S. Morimoto, *A Survey of Formal Verification for Business Process Modeling*, (ICCS 2008), pp.514-522, 2008.
19. M. Pesic, H. Schonenberg, W. M. P. van der Aalst *Declarative Workflow*, Modern Business Process Automation, pp. 175-201, 2010.
20. G. Regis, N. Aguirre, T. S. E. Maibaum, *Specifying and Verifying Business Processes Using PPML*, ICFEM, pp. 737-756, 2009.
21. P. Y. H. Wong, J. Gibbons, *Property specifications for workflow modelling*, Sci. Comput. Program, vol. 76, nro. 10, pp. 942-967, 2011.