Especificación y Verificación de Propiedades sobre Workflows con Lógica de Fluentes

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

[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

Resumen En el presente trabajo proponemos el uso de una lógica temporal conocida, *fluent linear temporal logic* (FLTL), para expresar formalmente propiedades sobre flujos de trabajo (*workflows*). Creemos que esta lógica es adecuada para dicha tarea, ya que el uso de *fluentes* nos permite caracterizar de manera flexible estados abtractos para representación de actividades y restricciones sobre workflows.
Con el fin de utilizar herramientas automáticas para el análisis de estas propiedades, nos focalizamos en una caracterización de workflows como sistemas de transición de estados etiquetados, que modela las tareas de un modo conveniente y aprovecha las fórmulas de FLTL. Más aún, automatizamos la codificación y empleamos *model checking*, utilizando LTSA (Labeled Transition System Analyzer), para garantizar que un workflow satisface una propiedad dada, o en caso contrario, generar una ejecución que exhiba la violación de la misma.

1. Introducción

La importancia de la eficiencia en las empresas requiere de una mejora constante de sus procesos organizacionales. Esto ha generado la necesidad de describir estos procesos, denominados *workflows* (flujos de trabajo), y ha promovido la creación de una gran cantidad de lenguajes de modelado para tal fin.

Un aspecto que consideramos particularmente importante en los lenguajes para la especificación de workflows es su semántica formal. Este aspecto es fundamental para poder efectuar análisis automáticos sobre los modelos del lenguaje y también está fuertemente relacionado con su expresividad, dado que lenguajes más expresivos son más difíciles de formalizar por completo.

La importancia del análisis de propiedades sobre *workflows* es reconocida por muchos autores (por ejemplo, remitirse a [14,16] entre otros). En este trabajo proponemos como lenguaje formal para expresar propiedades sobre workflows una lógica temporal, específicamente, *fluent linear temporal logic* [7]. Creemos que esta lógica es adecuada para expresar formalmente propiedades sobre workflows. Además, es posible utilizar herramientas de *model checking* para verificar especificaciones en dicha lógica. El Model Checking [5] es una técnica formal conocida que provee un método automatizado para verificar propiedades en sistemas de estados finitos, generalmente expresadas como fórmulas en lógica temporal. Para posibilitar el uso de model checking en la verificación de workflows,

proveemos una caracterización de éstos como sistemas de transición de estados etiquetados (LTS) que nos permite utilizar fluent LTL para capturar propiedades y restricciones de workflows. Como resultado empleamos model checking para garantizar la validez de una propiedad o para generar un contraejemplo en caso de violación de la misma. Este método para el análisis de workflows es muy flexible, contrariamente a otros trabajos, donde centran su análisis en propiedades específicas tales como *soundness*, ausencia de deadlock (por ejemplo, la herramienta discutida en [2]).

Nuestro enfoque es básicamente independiente del lenguaje escogido, y en principio podría ser utilizado para cualquier lenguaje de modelado de worklfows con semántica formal. Elegimos a YAWL (Yet Another Workflow Language) [2] como lenguaje para el modelado de workflows ya que es un poderoso lenguaje con semántica formal (Redes de Petri), basado en el uso de patrones de flujo de trabajo (*workflow patters*)[1] y que posee una herramienta de código libre. Este lenguaje para el modelado de workflows es considerado como un formalismo expresivo, tal como lo demuestran varios trabajos que comparan su expresividad con las de otros lenguajes de modelado de workflows (por ejemplo: Business Process Modeling Notation, Event-Driven Process Chains, etc) [8]. De hecho, la elección de YAWL nos permite garantizar la aplicabilidad de nuestro análisis a otros lenguajes, en muchos casos gracias a la existencia de herramientas de traducción entre estos y YAWL.

El trabajo continúa de la siguiente manera. En la próxima sección introducimos la noción de workflow, y presentamos a YAWL como lenguaje para la especificación de los mismos. En la sección 3 abordamos los conceptos formales que sirven de fundamento a nuestro trabajo. En la sección 4, proponemos una traducción automática para codificar especificaciones YAWL como LTS, más específicamente mediante FSP (lenguaje para modelar Procesos de Estados Finitos), caracterizando las tareas como *fluentes*. Mostramos cómo los fluentes son convenientes para expresar propiedades de comportamiento. Para esto, desarrollamos un caso de estudio tomado de la herramienta YAWL, cuya complejidad nos permite ilustrar las ventajas de nuestro enfoque. Finalmente, discutimos sobre trabajos relacionados en el area y resumimos conclusiones.

2. Procesos de Negocio, Workflows y Patrones

En la última década, se han desarrollado varios lenguajes y herramientas para proveer una visión organizada del comportamiento estructural de sistemas. Uno de sus los objetivos principales, es proveer un marco para la descripción y el análisis de actividades que forman parte del sistema. Como parte de estos esfuerzos, se creó la *Workflow Pattern Initiative* con el fin de identificar y brindar una base conceptual para la especificación de procesos de negocios. Esto resultó en la definición de una gran variedad de *patrones de flujo de trabajo* y la creación de un lenguaje conocido como YAWL [2].

2.1. YAWL como lenguaje para la especificación de workflows

Yet Another Workflow Language (YAWL) es un lenguaje para el modelado de workflows cuyos modelos son construidos en base a *workflow patterns* [1]. En este trabajo nos concentramos en patrones de flujo de control. Los modelos de YAWL están compuestos por tareas, condiciones y una relación de flujo entre ellos. La semántica de los modelos está basada en la Teoría de Redes de Petri, específicamente, el flujo de trabajo está caracterizado por *tokens* que habilitan la ejecución de una tarea. Cuando una tarea es ejecutada, ésta toma *tokens* de sus condiciones de entrada y coloca *tokens* en sus condiciones de salida. A diferencia de PN, en YAWL es posible conectar dos tareas de manera directa. Una característica distintiva de YAWL, es que provee soporte directo para el patrón denominado *Región de Cancelación* que permite modelar situaciones en donde una tarea t tiene asociado un conjunto de tareas y condiciones que son abortadas cuando t es ejecutada.

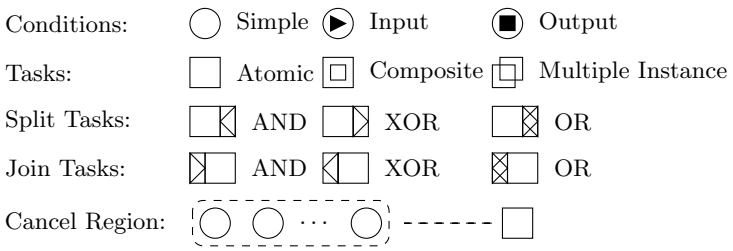


Figura 1. Símbolos YAWL

La especificación de un workflow (la perspectiva de flujo de control) en YAWL es un conjunto jerárquicamente organizado de redes YAWL. La figura 1 muestra los símbolos correspondientes a las construcciones del lenguaje. Una red YAWL está compuesta por:

- Una única *condición de entrada (start)* y una única *condición de salida (end)*.
- *Tareas*: el lenguaje provee tres tipos de tareas: *atómicas*, *compuestas* y *instancia múltiple*. Las primeras constituyen el nivel de descripción más bajo del sistema. Las tareas *compuestas* están asociadas con una red YAWL correspondiente que modela su comportamiento. Las tareas de *instancia múltiple* (MI) poseen una cota superior e inferior sobre el número de tareas creadas cuando la tarea inicia y un modificador indicando si la ceración se lleva a cabo de manera *estática* o *dinámica*. Además, una tarea t puede tener asociada una *región de cancelación*.
- Las construcciones utilizadas para la definición de flujo de control (*compuertas*) son las que se muestran en la figura 1. Su significado es el siguiente:
  - AND-join: una tarea asociada con esta computerta comienza a ejecutarse cuando *todos* los flujos de entrada están habilitados.
  - OR-join: una tarea asociada con esta construcción comienza a ejecutarse cuando *al menos* un flujo de entrada está habilitado.

- XOR-join: una tarea asociada con esta construcción comienza a ejecutarse cuando *exactamente un* flujo de entrada está habilitado.
- AND-split: cuando una tarea asociada a esta construcción es completada, el control pasa a *todos* los flujos de control salientes (en paralelo).
- OR-split: cuando una tarea asociada a esta construcción es completada, el control pasa a *uno o más* flujos de control salientes, basándose en la evaluación de condiciones lógicas asociadas a cada uno de ellos.
- XOR-split: cuando una tarea asociada a esta construcción es completada, el control pasa a *exactamente un* flujo de control saliente, basándose en la evaluación de las condiciones lógicas asociadas a cada uno de ellos.

3. Fundamentos Formales

3.1. Sistemas de Transición de Estados Etiquetados

Los Sistemas de Transición de Estados Etiquetados (LTS) son usados típicamente para modelar el comportamiento de sistemas con componentes que interactúan [11]. Estos sistemas describen modelos como un conjunto de estados y transiciones entre ellos. Las transiciones representan eventos del sistema que pueden ser compartidos (sincronizables). El comportamiento general del sistema es el resultado de la composición paralela de sus componentes, entendiendo ésta como el *interleaving* del comportamiento de sus componentes. Formalmente, un LTS P es una 4-upla $\langle Q, A, \delta, q_0 \rangle$, donde: Q es un conjunto finito de estados, A es el *alfabeto* de P (un subconjunto del universo de eventos Act); $\delta \subseteq Q \times A \cup \{\tau\} \times Q$ es una relación de transición etiquetada y q_0 es el estado inicial. Su semántica viene dada por su conjunto de *trazas*, es decir, el conjunto de secuencias de eventos que el mismo puede ejecutar, comenzando en el estado inicial y siguiendo su relación de transición. Para sistemas que poseen una cierta cantidad de estados es conveniente representarlos mediante álgebras de procesos, en nuestro caso, FSP [11]. Las expresiones en FSP pueden ser transformadas de modo automático en LTS y viceversa.

Una especificación FSP contiene dos tipos de definiciones de procesos: primitivos y compuestos. Los primeros se definen mediante el uso de prefijado de eventos "`->`", elección "`|`" y recursión. Las estructuras condicionales pueden ser expresadas mediante cláusulas "`when`" o expresiones "`if`". Tanto los nombres de eventos, como los nombres de procesos, pueden estar indexados, y los procesos primitivos pueden ser parametrizados. A manera de ejemplo, considérese la siguiente especificación de un buffer (Fig.2), cuyo comportamiento está especificado por un proceso primitivo que contiene una elección entre dos acciones posibles, `put` (poner) y `get` (quitar). Estas acciones están "multiplicadas" por medio de indexación. El LTS correspondiente se muestra en (Fig.3).

Los procesos pueden ser compuestos de manera secuencial ("`;`") o paralela ("`||`"). La composición en paralelo combina el comportamiento de dos procesos, sincronizando los eventos comunes de sus alfabetos o aquellos relacionados mediante el operador de renombramiento "`/`' , y haciendo *interleaving* sobre el resto de sus acciones.

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

Figura 2. Especificación de un buffer en FSP.

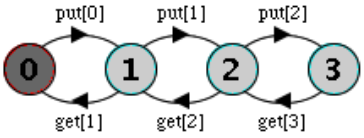


Figura 3. LTS del buffer.

3.2. Lógicas Temporales

Para poder razonar sobre el comportamiento de un LTS, se necesita una lógica capaz de expresar propiedades del mismo, como por ejemplo la Lógica Temporal Lineal (LTL) [12]. En ella, cada fórmula expresa una propiedad del comportamiento de un LTS. Formalmente, dado un conjunto de letras proposicionales $\mathcal{P}$, una fórmula bien formada se define inductivamente utilizando los operadores booleanos estándar y los operadores temporales **X** (next) y **U** (strong until), de la siguiente manera: *(i)* toda letra proposicional $p \in \mathcal{P}$ es una fórmula, y *(ii)* si ϕ y ψ son fórmulas, también lo son $\neg\phi$, $\phi \vee \psi$, $\phi \wedge \psi$, $\mathbf{X}\phi$, $\phi\mathbf{U}\psi$. Los operadores temporales **F** (eventually), **G** (always) y **W** (weak until) se definen de la siguiente manera: $\mathbf{F}\phi \equiv \text{true}\mathbf{U}\phi$, $\mathbf{G}\phi \equiv \neg\mathbf{F}\neg\phi$, y $\phi\mathbf{W}\psi \equiv ((\phi\mathbf{U}\psi) \vee \mathbf{G}\phi)$.

La Lógica Temporal Lineal de Fluentes (FLTL) es una variante de LTL, que resulta muy útil para de describir propiedades sobre sistemas discretos basados en eventos [11]. Básicamente, FLTL extiende a LTL incorporando la posibilidad de describir ciertos estados abstractos, llamados *fluentes*, caracterizados por eventos del sistema, específicamente, son verdaderos en un momento particular si han sido iniciados por un evento disparador, y no han sido terminados por ningún evento de terminación desde entonces. Formalmente, $Fl = \langle S, E \rangle$ *initially* B define un fluente Fl, en donde B es un valor lógico indicando su validez en el estado inicial del sistema y, S y E son dos conjuntos disjuntos de eventos que lo activan o desactivan respectivamente. Si el término *initially* B se omite, Fl es inicialmente considerado como *falso*. Como ejemplo, considérese la siguiente caracterización de los estados `full` y `empty`, que capturan las propiedades obvias del buffer presentado anteriormente:

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

Con el objetivo de analizar propiedades de modelos de manera automática utilizamos Labelled Transition System Analyzer (LTSA). Ésta es una herramienta para la verificación de modelos de sistemas concurrentes que soporta especificaciones FSP con precencia de fluentes. Siguiendo con los ejemplos previos, podemos verificar que el buffer no puede estar vacío y lleno simultáneamente mediante la fórmula FLTL: `assert CORRECT_BUFFER = [](!(Full && Empty)).`

4. Caracterización de Modelos YAWL en FSP y especificación de propiedades

Teniendo en cuenta el objetivo de especificar y verificar propiedades FLTL sobre workflows, presentamos una caracterización de redes YAWL como especificaciones FSP. Básicamente, la intución que subyace a la traducción entre YAWL y FSP es la siguiente: el comportamiento de un sistema está caracterizado por la ocurrencia de sus tareas. De modo abstracto, podemos capturar la noción de tarea como una entidad activa en el sistema entre las ocurrencias de sus eventos *start* y *end*. Así, una ejecución posible del sistema está reflejada por una traza de ocurrencia de dichos eventos, respetando ésta las restricciones de flujo de control. El comportamiento del sistema es el conjunto de todas las posibles trazas mencionadas.

De acuerdo con nuestra observación previa, es sencillo ver que una tarea puede ser modelada con el uso de un fluente, que será verdadero tras la ocurrencia de su evento *start*, y falso ante su evento *end*. Para poder capturar el flujo de control y así el comportamiento del sistema, necesitaremos introducir sincronizaciones apropiadas entre eventos y composiciones de procesos.

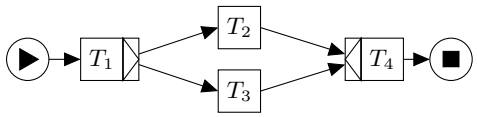


Figura 4. Red YAWL simple con compuertas XOR-split y XOR-join.

Para ilustrar la intución detrás de nuestra traducción de YAWL a FSP, considérese la red YAWL que se muestra como ejemplo en la figura 4. De acuerdo con la semántica de YAWL, el conjunto de todas las trazas de ejecución del sistema es: $\{T_1T_2T_4, T_1T_3T_4\}$. Cada una de éstas se corresponde con una traza de eventos del sistema, por ejemplo $[T_1.start\ T_1.end\ T_2.start\ T_2.end\ T_4.start\ T_4.end\ ]$ para la primera. De esta manera, la actividad de T_2 estará capturada por el fluente $\langle\{T_2.start\}, \{T_2.end\}\rangle$. Esta flexibilidad de descripción de los fluentes nos permite expresar propiedades del workflow de manera sencilla y potente. Como ejemplo podemos expresar la exclusión mutrua entre T_2 y T_3 mediante la fórmula: $\mathbf{G}\neg(T_2 \wedge T_3)$.

Para poder describir formalmente nuestra traducción de YAWL a FSP, debemos considerar la semántica formal de las redes YAWL [8]. Teniendo en cuenta dicha formalización, comenzamos presentando una traducción para las tareas y condiciones. Para las condiciones, debido a la finitud de los LTS, limitamos el comportamiento a una cantidad *acotada* de tokens en éstas. Si bien esta limitación es importante, todos los modelos YAWL siguen siendo traducibles a una especificación FSP con la desventaja de reportes de falsos positivos (contraejemplos que no pueden reproducirse en el modelo original). De todos modos el análisis sigue siendo conservativo.

Una vez caracterizadas las tareas y condiciones, para poder representar el comportamiento de una red, necesitamos especificar cómo componer dichas cons-

trucciones reflejando los patrones de flujo de control asociados a las compuertas. Finalmente, tratamos casos de mecanismos sofisticados de YAWL, tales como las regiones de cancelación y las tareas compuestas.

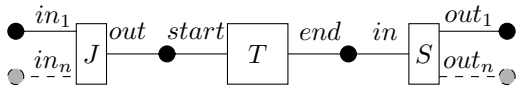
Sea N una red YAWL. El proceso que representa las condiciones *input*(i) y *output*(o) de N, es el siguiente: `YNET = (i_cond->o_cond->YNET).`

Las tareas atómicas son caracterizadas por procesos FSP que especifican sus eventos *start* y *end*: `TASK = (start ->end ->TASK).`

Como se mencionó anteriormente, la codificación de las condiciones está acotada, su caracterización como proceso FSP es similar a la de un buffer acotado (ver Fig. 2), indicando mediante dos parámetros las posibles entradas y salidas de la condición. Su tamaño por defecto es la cantidad de conexiones entrantes.

Las conexiones (*puertos*) de entrada/salida de los procesos generados son codificadas con los eventos *in* y *out*. Consideremos ahora tsk_1, tsk_2, dos tareas atómicas de N sin compuertas asociadas, y sea c una condición con n y m puertos de entrada y salida respectivamente. Para componer tsk_1 y tsk_2 secuencialmente sincronizamos $tsk_1.end$ y $tsk_2.start$ por medio de renombramiento. Para componer tsk_1 y tsk_2 a través de c, sincronizamos $tsk_1.end$ con algún puerto de entrada de c, y $tsk_2.start$ con algún puerto de salida de c.

Para la traducción de tareas en precencia de compuertas, considérese t, una tarea atómica con alguna compuerta *join* o *split* (AND, OR, XOR). Para cada compuerta posible, generamos un proceso que especifica su comportamiento. Estos procesos están parametrizados por la cantidad de puertos de entrada o salida según corresponda, por ejemplo, una compuerta*split* tendrá un sólo puerto de entrada. Como se muestra en el sigiente diagrama, si T tiene alguna compuerta join (J) o split (S) asociada, su interconexión es obtenida sincronizando $J.out$ con $T.start$, y $T.end$ con $S.in$, respectivamente.



La ocurrencia de tareas T_i en el sistema es caracterizada por un *fluente* de la forma $T_i = \langle \{tsk_i.start\}, \{tsk_i.end\} \rangle$. A manera de ejemplo, consideremos la siguiente codificación de la red YAWL de la figura 4 como un proceso FSP:

```
|| 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. Codificación de las Compuertas, Regiones de Cancelación y Tareas Compuestas

Para cada compuerta generamos un proceso FSP que representa su comportamiento. Estos procesos están parametrizados por la cantidad de puertos de entrada y salida. Por razones de espacio sólo presentaremos la codificación de algunas compuertas. Para *AND-split* generamos el siguiente proceso:

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

Este proceso dispara tantos eventos *out* como son especificados por el parámetro (`forall` es una abreviación para la composición paralela "||").

La compuerta *OR-join* tiene diferentes interpretaciones en los diversos lenguajes de modelado de procesos de negocios. En [8], se puede encontrar una reseña sobre la semántica del *OR-join* en ellos. Nuestro enfoque utilizada la semántica informal del OR-join (cf. [8], p. 104), que prioriza todas los posibles eventos de entrada antes de habilitar la salida.

```
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 ).
```

Aquí todos los eventos de entrada son "escuchados", si alguno de ellos es activado, se disparar el evento de salida. La prioridad sobre los eventos de entrada se logra aplicando el operador de *prioridad* ($>>$) al evento *out*.

Para las *regiones de cancelación*, consideramos una versión de codificación extendida de las tareas y condiciones que pertenezcan a la misma. La extensión contempla la posibilidad de ocurrencia del evento *cancel*, el cual se sincroniza con el evento *end* de la tarea cancelante.

Para los sistemas con *tareas compuestas*, cada una de estas tendrá una red YAWL asociada con su correspondiente codificación FSP que estará sincronizada (`i_cond, o_cond`) con la red principal.

5. Caso de Estudio

Como caso de estudio tomamos un modelo que provee la herramienta YAWL[3], ya que consideramos apropiado debido a su tamaño y complejidad. Éste involucra todas las construcciones del lenguaje, lo cual lo hace útil para probar todos los aspectos de la traducción a FSP.

El caso de estudio, *Order Fulfillment*, describe el proceso de cumplimiento de una orden en una empresa ficticia, el cual está dividido en las siguientes etapas: Realización de la Orden (*Ordering*), logística y Pago (*Payment*). La logística incluye los procesos de Asignación de Transporte (*Carrier Appointment*), Mercancía en Transito (*Freight in Transit*) y Mercancía Despachada (*Freight Delivered*). El modelo YAWL de la orden se muestra en la figura 5, en donde cada una de las fases mencionadas es capturada por una tarea compuesta.

Con el objetivo de automatizar la traducción, se desarrolló la herramienta "YAWL2FSP". Con ella se generó el FSP correspondiente a *Order Fulfillment*. Luego, mediante la LTSA, se obtuvo el LTS correspondiente en 0.298 segundos y requirió 28,96 Mb. de memoria. Éste contiene 13164 estados y 59722 transiciones. Finalmente se verificaron algunas propiedades interesantes que, por razones de espacio, sólo explicamos su significado en términos del modelo YAWL de modo

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

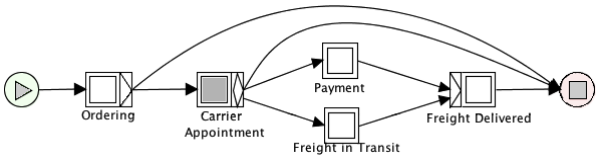


Figura 5. Red YAWL correspondiente al cumplimiento de una orden.

breve. La herramienta desarrollada y los detalles de las propiedades pueden ser consultados en `http://sourceforge.net/projects/yawl2fsp/`.
1) *Si el tiempo máximo expira en la designación de un transporte, entonces el paquete no es enviado.* `CarrierTimeout ->!ProduceShipmentNotice`

Donde, el fluente que representa la actividad de *Carrier Timeout* está definido como: `fluent CarrierTimeout = <C_A.task[5].start,C_A.task[5].end >`, en esta definición `C_A` hace referencia a la subred *Carrier Appointment* y `task[5]` representa la tarea *CarrierTimeout*. El fluente correspondiente a la tarea *Produce Shipment Notice* se define de manera similar.
2) *Las tareas que corresponden a distintos modos de envío no pueden ocurrir de manera simultánea.* `!( (FTL && LTTL) || (FTL && SP) || (SP && LTTL))`. Donde los fluentes `FTL`, `LTTL` y `SP` caracterizan cada una de las formas de transporte. Por ejemplo, `FTL=<C_A.ftl.i_cond,C_A.ftl.o_cond>`, aquí `ftl` es el identificador del proceso FSP de la sub-red.

Los recursos necesarios para la verificación de las propiedades fueron: (1) 154 ms [11.8MB] y (2) 152 ms [11.9MB]. La traducción y verificación se llevó a cabo en un sistema provisto de un procesador Intel Core 2 Duo 2.2 Ghz, 2 GB 667 Mhz DDR2 SDRAM de memoria y un sistema operativo basado en Unix.

6. Conclusiones y Trabajos Futuros

La especificación formal y verificación de procesos de negocios ha ganado relevancia en la última decada, no sólo academicamente, sino también en el ámbito industrial, en dónde la optimización de los procesos de negocios es crucial. Como consecuencia, se han propuesto una gran cantidad de lenguajes para la descripción de los mismos. Si bien, la mayoría de los lenguajes presentan una sémantica informal, con el objetivo de realizar análisis de sus modelos, en los últimos años han cobrado interés diversas caracterizaciones formales de workflows [4,15]. En [13] se puede encontrar una reseña general de los diferentes lenguajes de modelado de procesos de negocios.

En este trabajo, nos concentramos en el análisis de propiedades sobre workflows, presentamos una traducción automática de workflows YAWL a especificaciones FSP, con el fin de verificar propiedades FLTL sobre los mismos utilizando LTSA. Hemos elegido YAWL como la base de nuestro trabajo debido a su semántica formal, y su soporte para una amplia gama de patrones de workflow. Como mencionamos, YAWL permite la verificación de algunas propiedades especificas sobre workflows, tales como *soundness* o ausencia de deadlock [3],

pero no provee un marco lo suficientemente flexible para expresar otro tipo de propiedades de comportamiento.

Nuestra codificación traduce YAWL a FSP preservando su semántica, posibilitando así el uso de *fluentes* para capturar propiedades sobre la ejecución del sistema. Los workflows, en particular aquellos basados en patrones de control de flujo, están inherentemente basados en eventos, lo cual dificulta su análisis con lógicas temporales tradicionales como LTL, que están basadas en la noción de estado. Por otro lado FLTL, permite describir de modo natural, escenarios de ejecución en workflows mediante la ocurrencia de eventos de activación y desactivación, a diferencia de los enfoques presentados en [10,6]. Otro trabajo relevante es [9], donde se propone un método basado en autómatas para la formalización de workflows pero con limitaciones de poder expresivo respecto de YAWL.

Este trabajo es parte de un proyecto más ambicioso en donde estamos desarrollando un ambiente para el análisis automatizado de workflows. Entre nuestros objetivos se encuentran un asistente para la descripción gráfica de fluentes y propiedades en workflows, permitiendo así al usuario estandar prescindir de conocimientos avanzados para realizar análisis en los mismos.

Referencias

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. A. H. M. ter Hofstede, W. M. P. van der Aalst, M. Adams, N. Russell, *Modern Bussiness Process Automation*, Springer, 2010.
9. C. T. Karamanolis, D. Giannakopoulou, J. Magee and S. M. Wheater, *Model Checking of Workflow Schemas*, EDOC, pp.170-181, 2000.
10. N. Leyla, A. S. Mashiyat, H. Wang, W. MacCaull, *Towards workflow verification*, CASCON, pp. 253-267, 2010.
11. J. Magee, J. Kramer, *Concurrency: State Models and Java Programs*, 1999.
12. Z. Manna, A. Pnueli, *Temporal Verification of Reactive Systems (Safety)*, 1995.
13. S. Morimoto, *A Survey of Formal Verification for Business Process Modeling*, (ICCS 2008), pp.514-522, 2008.
14. M. Pesic, H. Schonenberg, W. M. P. van der Aalst *Declarative Workflow*, Modern Business Process Automation, pp. 175-201, 2010.
15. G. Regis, N. Aguirre, T. S. E. Maibaum, *Specifying and Verifying Business Processes Using PPML*, ICFEM, pp. 737-756, 2009.
16. P. Y. H. Wong, J. Gibbons, *Property specifications for workflow modelling*, Sci. Comput. Program, vol. 76, nro. 10, pp. 942-967, 2011.