Biblioteca Digital
F C E N - U B A

BIBLIOTECA CENTRAL LUIS F LELOIR
BIBLIOTECA CENTRAL LELOIR
FACULTAD DE CIENCIAS EXACTAS Y NATURALES UBA

Tesis Doctoral

Especificación formal y verificación de propiedades temporales de procesos de negocios

Regis, Germán Enrique

2014

Este documento forma parte de la colección de tesis doctorales y de maestría de la Biblioteca Central Dr. Luis Federico Leloir, disponible en digital.bl.fcen.uba.ar. Su utilización debe ser acompañada por la cita bibliográfica con reconocimiento de la fuente.

This document is part of the doctoral theses collection of the Central Library Dr. Luis Federico Leloir, available in digital.bl.fcen.uba.ar. It should be used accompanied by the corresponding citation acknowledging the source.

Cita tipo APA:
Regis, Germán Enrique. (2014). Especificación formal y verificación de propiedades temporales de procesos de negocios. Facultad de Ciencias Exactas y Naturales. Universidad de Buenos Aires.

Cita tipo Chicago:
Regis, Germán Enrique. "Especificación formal y verificación de propiedades temporales de procesos de negocios". Facultad de Ciencias Exactas y Naturales. Universidad de Buenos Aires. 2014.

EXACTAS UBA
Facultad de Ciencias Exactas y Naturales

UBA
Universidad de Buenos Aires

Dirección: Biblioteca Central Dr. Luis F. Leloir, Facultad de Ciencias Exactas y Naturales, Universidad de Buenos Aires. Intendente Güiraldes 2160 - C1428EGA - Tel. (++54 +11) 4789-9293

Contacto: digital@bl.fcen.uba.ar

Universidad de Buenos Aires
Facultad de Ciencias Exactas y Naturales
Departamento de Computación

Especificación formal y verificación de propiedades temporales de procesos de negocios

Tesis presentada para optar al título de Doctor de la Universidad de Buenos Aires en el área Ciencias de la Computación

Germán Enrique Regis

Directores:

Dr. Nazareno Matías Aguirre
Dr. Carlos Gustavo López Pombo

Lugar de Trabajo:

Departamento de Computación

Facultad de Ciencias Exactas Físico-Químicas y Naturales
Universidad Nacional de Río Cuarto

Río Cuarto, 2014

Especificación formal y verificación de propiedades temporales de procesos de negocios

Resumen: El uso de diferentes métodos y herramientas que asistan en la organización y el control de procesos, ha tomado relevancia debido a la necesidad constante de eficiencia por parte de la industria y a la adecuación de sus prácticas a las emergentes y novedosas técnicas de desarrollo y producción. Como producto de estas necesidades, diversos lenguajes para la descripción de procesos de negocios capturaron el interés tanto del sector productivo, para su aplicación en sus actividades, como así también del sector académico, con el fin de aportar conocimiento y generar nuevas herramientas para tal fin.

Si bien la mayor parte de los lenguajes para la especificación de procesos de negocios son informales, existen algunos fundados en formalismos matemáticos. Esto brinda a estos lenguajes formales para procesos de negocios la eliminación de ambigüedades en la interpretación de especificaciones, y la posibilidad de realizar diferentes análisis formales, como chequeos de sanidad. Sin embargo, las alternativas existentes poseen en general limitaciones, o bien en poder expresivo, o, para las alternativas más expresivas, la imposibilidad de realizar algunas tareas de análisis automáticamente. En este trabajo presentamos un enfoque formal para la especificación y el análisis de procesos de negocios, que permite dar semántica a construcciones complejas de procesos de negocios, a la vez que admite el análisis de chequeos de sanidad usuales de manera automática mediante model checking. Más aún, nuestra propuesta abarca la posibilidad de que el usuario especifique y analice propiedades de interés acerca de las especificaciones, mediante lógicas temporales, un mecanismo con frecuencia ausente en los lenguajes existentes.

Además del desarrollo de una semántica formal para un lenguaje de proceso de negocios sumamente expresivo, nuestro trabajo toma como base para la especificación de propiedades de procesos de negocios el uso de lógicas temporales. Identificamos algunos elementos que hacen particularmente complicada la expresión de algunos tipos de propiedades, en particular cuando las mismas demandan expresar repeticiones de eventos o tareas, y proponemos una lógica temporal que permite describir tales propiedades con mayor facilidad. Con el fin de no perder capacidad de análisis, desarrollamos traducciones que nos permiten capturar los elementos novedosos de esta lógica en términos de otros elementos, soportados por lógicas

temporales tradicionales, con soporte de herramientas. Por otra parte, observamos la tendencia de los lenguajes de descripción de procesos de negocios a no tratar formalmente la descripción de los productos que manipulan. Tomamos un formalismo excepcional en este sentido, PPML, y definimos a partir del mismo un lenguaje de especificaciones, una semántica simplificada para el mismo, y una técnica de análisis de especificaciones basada en una caracterización de sus modelos en UPPAAL.

Nuestro enfoque está claramente orientado a la producción de herramientas de soporte al proceso de especificación. Para los lenguajes tratados brindamos herramientas de especificación y análisis de procesos de negocios, como así también la posibilidad de construir descripciones de procesos de negocios a partir de las propiedades que los mismos deben cumplir. Estas tareas son evaluadas en este trabajo a través del desarrollo de casos de estudio tomados de la literatura del área, en lo que respecta a descripción de procesos y chequeos de sanidad de los mismos, y en base a variantes de estos casos en las cuales se enfatiza la necesidad de contar con propiedades descriptas por el usuario, y de dar más relevancia a la descripción de los productos manipulados por los procesos de negocios.

Palabras clave: Procesos de Negocios, Workflows, Lógicas Temporales, Fluentes, Análisis automático, Model checking.

Formal specification and temporal properties verification of business processes

Abstract: The use of methods and tools to assist in process organization and control is receiving increasing attention. This is due to the constant need for efficiency in industry, and the pressure on industry to adopt emergent development and production techniques, and adapt current practices to these emergent technologies. As a consequence, various languages for describing business processes have attracted both industry, which applies them in its activities, and academia, which can contribute knowledge and help provide novel tools for these languages.

Even though most languages for business process specification are informal, some of these languages are founded on mathematical formalisms. Formal business process specification languages profit from the elimination of ambiguities in the interpretation of their specifications, and the possibility of performing different kinds of formal analyses, such as soundess checks. However, the available formal alternatives have limitations, either in expressive power, or, in the case of the more expressive formalisms, the impossibility of performing some analysis tasks fully automatically. In this work we present a formal approach for the specification and analysis of business processes, that allows us to provide suitable semantics to complex constructions in business process specifications, while at the same time allowing us to perform typical soundness checks fully automatically via model checking. Moreover, our approach includes the possibility of enabling users to specify and analyze properties of their interest about specifications, using temporal logics, a mechanism often missing from existing languages.

Besides the development of a formal semantics for a very expressive business process language, our work is based on the use of temporal logics for specifying properties of business processes. We identify some elements that make it difficult to express some kinds of properties, in particular those demanding referring to repetitions of events or tasks, and propose a temporal logic that allows us to specify these properties more conveniently. In order to maintain the possibility of automatically analyzing specifications, we develop translations that enable us to capture the novel elements in our logic in terms of other elements, supported by traditional temporal logics with tool support. Also, we observe a tendency in business process languages to disregard, in their formalizations, the description of the products that processes

manipulate. We take a formalism that is exceptional in this respect, PPML, and define a formal specification language in terms of it, a simplified semantics for it, and an analysis technique for its specifications, based on a characterization of its models in UPPAAL.

Our approach is clearly oriented towards producing tool support for business process specification. For the languages that we deal with in this work, we provide tools for business process specification, property specification, and automated analysis. We also enable the user to build business process descriptions from the properties these processes are required to fullfil. These tasks are assessed in our work via case studies taken from the literature, for the case of business process description and soundness checks, and variants of these case studies in which the need of referring to event repetitions, and the role of products in process descriptions, are emphasized.

Keywords: Business Process, Workflows, Temporal Logics, Fluents, Automatic Analysis, Model Checking.

Agradecimientos

Que difícil es para mi poder expresar un agradecimiento de manera breve a todos aquellos que de alguna u otra manera los siento parte de este “infinitesimal aporte a la ciencia e infinto logro para mí”. Sinceras disculpas a aquellos que, por falta de memoria, en este preciso instante estoy omitiendo explícitamente, estén seguros que jamás olvidaré los momentos compartidos.

Este trabajo tiene un promotor... gracias Naza por haberme dado la oportunidad de construir este sueño en mi vida, no sólo por tu paciencia y dedicación en la dirección de este trabajo, sino también por tu amistad, que sin duda fue la fuente de apoyo, inspiración y coraje para enfretar este gran desfío.

Gracias Gabriel, Profe y Chelo por su confianza depositada en mí al apostar a esta descabellada idea de realizar un doctorado, espero poder estar a la altura de semejante compromiso durante el resto de mi vida. Mi más sincero agradecimiento por los momentos compartidos y el ejemplo de esfuerzo, pasión y entrega a las ciencias que despertaron en mí.

Many thanks Tom for letting me work with you, sharing your ideas and making real one of the greatest experiences of my life.

Gracias Charly por aceptar semejante desfío y por haberme dado el gran empujón final.

Gracias Pancho por tu amistad y compañerísmo, ayudandome a empujar juntos en los momentos difíciles y compartiendo con una sonriza los buenos momentos de este camino.

Gracia a mis amigos del DC: Marcelo, Marta, Pablo, Vale, Zurdo, Pancho, Sonia, Gaston, Nico, Ceci, Ivana, Chino, Damian, Ramiro, Valen... por su incondicional apoyo y por compartir juntos este fascinante tramo de nuestras vidas.

Gracias a Mariano, Nico, Dippi, Diego por hacerme sentir “como en casa” cuando estoy a más de 600km. de ella.

A Victor, Pedro y Gustavo, quienes fueron jurados de este trabajo, es un gran honor para mi que hayan aceptado la revisión del mismo. Gracias por la lectura minusiosa y sus observaciones, que además de aportar a la calidad de este trabajo, fueron de gran enseñaza para mi carrera.

A la Universidad de Buenos Aires y las personas involucradas en el desarrollo de mi carrare que allí trabajan, por aceptarme como alumno y por su cordial y en ocaciones, excepcional trato conmigo.

A la Universidad Nacional de Río Cuarto, por haberme formado en mi carrera de grado y por brindarme el lugar de donde puedo hacer lo que me gusta.

Sin duda, este trabajo no hubiera sido factible sin los recursos que me otorgaron el Consejo Nacional de Investigaciones Científicas y Técnicas (CONICET), la Agencia Nacional de Promoción Científica y Tecnológica (ANPCyT) y el Ministerio de Ciencia y Tecnología de Cordoba (MinCyT). Mi más sincero agradecimiento a estas instituciones.

Para finalizar quiero dedicar este trabajo a Mamá y Papá, por haberme dado el mágico don de la vida y por haberme criado resaltando los valores de honestidad, solidaridad, estudio y esfuerzo. A mi querida hermana por ser el gran pilar en la familia en el que puedo descansar. A Fino, por haber llenado mis silencios de trabajo, acompañando a mis hijos y cebando los mas exquisitos mates.

A los seres que más amo, Eli, Valen y Tomy, por regalarme en cada despertar las ganas de seguir viviendo, la fuerza para seguir luchando y la esperanza de seguir soñando con un mundo mejor para todos.

Índice general

1 Introducción **2**
1.1 Estado del Arte 3
1.2 Objetivos 6
1.3 Organización 8

2 Preliminares **11**
2.1 Introducción 11
2.2 Modelos de Comportamientos para Sistema Reactivos 12
2.2.1 Sistemas de Transición de Estados 13
2.2.2 Redes de Petri 17
2.2.3 Autómatas Temporizados 21
2.3 Lógicas 24
2.3.1 Lógica Proposicional 24
2.3.2 Lógicas Temporales 25
2.4 Técnicas de Análisis Automático 31
2.4.1 Model Checking 31

3 Lenguajes para el modelado de Procesos de Negocio y Workflows **41**
3.1 Introducción 41
3.2 Procesos de Negocios y Flujos de Trabajo 42
3.3 Descripción y especificación de workflows 43
3.3.1 Elementos de especificación 43
3.4 Análisis de Workflows 49

4 Especificación y Verificación de Propiedades de Workflows **55**
4.1 Introducción 55
4.2 Verificación de Propiedades de Workflows 57
4.3 Caracterización de YAWL mediante FSP 59
4.3.1 Relación de flujo de trabajo 63

4.3.2 Codificación de Compuertas 65
4.3.3 Regiones de Cancelación 74
4.3.4 Tareas Compuestas 75
4.3.5 Tareas Múltiples 75
4.3.6 Reinicio de compuertas 77
4.3.7 Corrección de la traducción 77
4.4 Especificación de Propiedades 80
4.4.1 Fluentes 81
4.5 Automatización de la codificación y asistente para el análisis de *Workflows*. 83
4.6 Caso de Estudio 84
4.6.1 Propiedades analizadas 87
4.7 Otras Aplicaciones de Especificaciones de Propiedades Temporales para la Construcción de Workflows 91
4.7.1 Esquema del proceso 91
4.8 Counting Fluent Linear Time Temporal Logic 94
4.8.1 Representación como Autómatas de Büchi 98
4.8.2 Implementación y Análisis 102
4.9 Sumario 105

5 Productos como referentes explícitos en el modelado de Workflows 108
5.1 Introducción 108
5.2 Product Process Modelling Language 110
5.2.1 Productos 110
5.2.2 Procesos 116
5.2.3 Compuertas 120
5.2.4 Composición de procesos 122
5.2.5 Procesos estructurados 125
5.3 Sintaxis y Representación gráfica de PPML 126
5.3.1 Representación Gráfica 127
5.3.2 Simple caso de estudio 128
5.4 Sumario 130

6 Verificación de Propiedades de Productos 132
6.1 Introducción 132
6.2 Caracterización de PPML mediante Uppaal 133
6.2.1 Traducción de Productos 134
6.2.2 Traducción de Procesos 141
6.3 Especificación de Propiedades 144

6.4 Formalización y Análisis de Procesos de Negocios Mediante π-Cálculo 147
6.4.1 π-Cálculo 147
6.4.2 Descripción general de la semántica de π-PML 148
6.4.3 Análisis de modelos 150
6.5 Sumario 150

7 Conclusiones y Trabajos Futuros 152
7.1 Trabajos Futuros 154

A Lenguajes para el Modelado de Procesos de Negocios 165
A.1 The Business Process Modeling Notation (BPMN) 165
A.1.1 Objetos de Flujo 165
A.1.2 Canales 167
A.1.3 Artefactos 167
A.1.4 Conectores 168
A.2 Yet Another Workflow Language (YAWL) 168
A.2.1 Patrones de Flujos de Trabajo 168
A.2.2 Lenguaje YAWL 170
A.2.3 Semántica formal de YAWL 171
A.3 Event-Driven Process Chain (EPC) 172
A.3.1 Objetos de Control 173
A.3.2 Objetos Complementarios 174
A.3.3 Conectores 175
A.4 Unified Modeling Language (UML) 176
A.4.1 Diagrama de Actividades 176

Índice de figuras

2.1 Especificación de una máquina expendedora de té o café, a través de un sistema de transición de estados. 15
2.2 Especificación FSP de un *Buffer* limitado. 16
2.3 Especificación de un sistema compuesto por un *buffer* acotado, un productor y un consumidor. 17
2.4 LTS correspondiente la especificación FSP de la Figura 2.4 con `Size=3` 18
2.5 Representación gráfica de los componentes de un Red de Petri 18
2.6 Semántica Operacional de una Red de Petri 20
2.7 Semántica Operacional de una Red de Petri con arcos de reinicio . . . 21
2.8 Modelo de Autómata Temporizado para *clics* de un *mouse*. 23
2.9 Representación gráfica de los operadores temporales de LTL 27
2.10 Representación de un sistema que satisface:(a)$\mathbf{EF}\phi$, (b)$\mathbf{EG}\phi$, (c)$\mathbf{AF}\phi$, (d)$\mathbf{AG}\phi$ 31
2.11 Especificación mediante LTS del comportamiento de arrojar una moneda. 32
2.12 Modelo en Uppaal que representa un usuario con dos ratones 39

3.1 Bosquejo de un proceso 44
3.2 Composición Vertical y Horizontal de Actividades 46

4.1 Esquema general de la verificación de Workflows 58
4.2 Red YAWL simple, con compuertas XOR-split y XOR-join 60
4.3 "TWO_TSK_COND": Modelo YAWL con dos tareas atómicas consecutivas y una condición intermedia 64
4.4 Configuración abstracta de una tarea con compuertas de bifurcación y unión. 65
4.5 LTS correspondiente a la especificación FSP de una compuerta de bifurcación `AND` con dos puertos de salida. 67
4.6 LTS (simplificado) correspondiente a la compuerta de bifurcación `OR` con dos salidas. 69

4.7 LTS correspondiente a la compuerta de unión `AND` con dos entradas. . 70
4.8 LTS correspondiente a la compuerta de unión `XOR` con dos entradas. . 71
4.9 LTS correspondiente a la compuerta de unión `OR` con dos entradas. . . 72
4.10 Modelo FSP correspondiente al workflow de la Figura 4.2 73
4.11 LTS correspondiente al modelo YAWL de la Figura 4.2. 73
4.12 Captura de pantalla de la herramienta FLYAnalyser. 83
4.13 Red principal del Caso de Estudio "*Order Fulfillment*" 84
4.14 Modelo YAWL de la tarea compuesta "*Carrier Appointment*" 86
4.15 Esquema general del proceso de construcción de Workflows 92
4.16 Autómata de Büchi que representa la relación $cFl_1 \geq 1$ 99
4.17 Autómata de Büchi que representa la relación entre dos fluentes $cFl_1 \leq cFl_2$ 102
4.18 Sincronizador y una descripción abstracta del mismo. 104

5.1 Representación gráfica de los componentes de un modelo PPML . . . 128
5.2 Diagrama PPML del caso de estudio. 129

6.1 Autómata temporizado correspondiente al esquema del producto *MotherBoard* 139
6.2 Autómata temporizado correspondiente al esquema del proceso *Assem1* 143

A.1 Notación gráfica de BPMN 169
A.2 Notación gráfica de YAWL 171
A.3 Semántica de construcciones YAWL en términos de Redes de Petri con reinicio 172
A.4 Notación gráfica de EPC 173
A.5 Notación gráfica de Diagrama de Actividades UML 178

Capítulo 1

Introducción

El constante esfuerzo de distintas organizaciones por el perfeccionamiento de sus procesos en busca de eficiencia y control, ha impulsado el actual auge de diversos métodos y lenguajes para el modelado de procesos de negocios. Estos métodos y lenguajes proponen como uno de sus principales objetivos, la descripción de procesos de negocios a través de la especificación de los elementos que los componen y la relación entre estos elementos. Estas descripciones brindan al usuario un modelo abstracto con una visión estructurada del proceso de negocios en cuestión, identificando en él los actores intervinientes, sus respectivas responsabilidades, y los recursos necesarios para el desarrollo del mismo.

Toda descripción admite diferentes niveles de abstracción. En las especificaciones de procesos (y en particular de procesos de negocios), a medida que el nivel de abstracción en la descripción de procesos decrece, surge la necesidad de dar descripciones operacionales que permitan capturar de forma más precisa las actividades que componen los procesos, pero sin necesariamente tener que alcanzar un nivel de detalle propio de una implementación. Este tipo de descripciones se basan generalmente en el uso de *modelos de procesos de negocios*. Cuando a las actividades (y relaciones entre las mismas) que componen un modelo de proceso de negocio se las puede describir de manera precisa, se denomina al modelo un *flujo de trabajo* (*workflow*) [46, 91].

En la última década han surgido una gran cantidad de herramientas de software para asistir en la descripción de procesos de negocios. Éstas forman parte de lo que se denomina *Business Process Management*, el campo de conocimiento que agrupa conceptos, metodologías, notaciones, herramientas, etc., que apuntan a mejorar la eficiencia y efectividad de procesos de negocios. Es por esto que, en su gran mayoría, estas herramientas tuvieron su génesis influenciadas fuertemente por la industria y sus objetivos orientados hacia la gestión de procesos de negocios. Estas herramientas

proveen en general lenguajes gráficos para la descripción de los modelos y, en general, carecen de una semántica formal, es decir, el significado de los modelos suele quedar abierto en algunos aspectos, y por lo tanto sujetos a la interpretación o intención de los usuarios. Un aspecto importante en el modelado de procesos de negocios es el *análisis* que podemos realizar sobre los mismos. En este sentido, el nivel de formalidad del lenguaje es fundamental para poder automatizar dicho análisis. Además del grado de formalidad, la *expresividad* juega también un papel importante en análisis (lenguajes más expresivos suelen dar lugar a especificaciones que son más difíciles o costosas de analizar).

La necesidad de comprender modelos de procesos de negocios y analizar sus propiedades ha fomentado la investigación en análisis sistemático de procesos de negocios. Una condición deseable del análisis es su automatización, es decir, poder desarrollar procedimientos automáticos que reciban como entrada modelos de procesos de negocios, y propiedades o condiciones a analizar en los mismos, y que produzcan como resultado respuestas relativas a si los modelos cumplen o no las propiedades en cuestión. Este proceso suele denominarse verificación [92].

Una de las técnicas de análisis automático que ha tomado mayor relevancia en Ingeniería de Software es el *Model Checking*. Esta técnica consiste en, dado un modelo (acotado, en general finito) de un sistema, comprobar de forma exhaustiva y automática si este modelo cumple con una propiedad determinada. Generalmente, los modelos se expresan como algún tipo de máquina de estados, y las propiedades en alguna lógica temporal. Una característica importante de Model Checking es que, en caso de que el modelo no cumpla con la propiedad, esta técnica permite obtener un *contraejemplo*, una traza de ejecución de dicho modelo que lleva a una violación de la propiedad [25]. Pese a algunas limitaciones, como su aplicación a modelos acotados y el inherente problema de explosión de estados, el Model Checking ha tenido gran auge en los último tiempos, no sólo en ámbitos académicos sino también en su aplicación en la industria. En este trabajo, como explicaremos más adelante, desarrollaremos bases y herramientas para la aplicación de análisis basado en model checking a especificaciones de procesos de negocios.

En la próxima Sección 1.1 analizamos diferentes trabajos en el contexto de esta tesis que fundamentan los objetivos de la misma que se detallan en la Sección 1.2.

1.1 Estado del Arte

La especificación y verificación de procesos de negocios ha ganado relevancia en la última década, no sólo a nivel académico, sino también en la industria, donde diferentes lenguajes y métodos fueron desarrollados. Si bien estos lenguajes y sus

respectivas herramientas surgieron fuertemente influenciados por su aplicación en la industria, en los últimos años diversos enfoques formales basados en estos lenguajes fueron abordados. El principal objetivo de las caracterizaciones formales de procesos de negocios fue la de soportar el análisis automático o semi-automático de los mismos.

En [73] podemos encontrar una revisión exhaustiva de diferentes enfoques para la formalización de procesos de negocios, clasificados según los formalismos en los cuales se basan. Particularmente, se analizan los enfoques según tres tipos de semánticas formales subyacentes: las basadas en *autómatas* [48], en *Redes de Petri* [38] y en *Álgebras de Procesos* [9, 14].

Entre los lenguajes para el modelado de procesos de negocios como mayor aceptación por parte de la industria, podemos mencionar a la Business Process Modeling Notation (BPMN). El mismo fue adoptado por muchas herramientas como lenguaje para la descripción de sus modelos. En [112, 113] se propone una semántica formal para BPMN basada en álgebras de procesos, específicamente en CSP [45, 86]. Esta semántica permite el análisis de compatibilidad en la interacción entre modelos a nivel de diseño (denominadas *colaboraciones*), incluso teniendo en consideración la noción de tiempo real.

BPEL (Business Process Execution Language) surgió a mediados de la década pasada como una combinación de dos lenguajes existentes. Este lenguaje se enfoca en brindar soporte de alto nivel para la descripción de las interacciones entre diferentes módulos, componentes o sistemas. En este sentido, y a pesar de contar con una notación sintáctica que se asemeja a la de un lenguaje de programación, BPEL apunta a descripciones de sistemas "in the large" [26]. En [94] se presenta una revisión exhaustiva de diferentes técnicas y herramientas utilizadas para el análisis de BPEL. Entre las herramientas evaluadas, se mencionan SPIN [47] y LOTOS [16], entre otras. Para el mismo lenguaje, en [80] se propone la utilización de Model Checking para el análisis de alcanzabilidad (y en particular ausencia de deadlock). En dicho trabajo se presenta una traducción de modelos caracterizados en BPEL a autómatas temporizados. Por otra parte, además el proceso de análisis desarrollado se integró a ActiveBPEL, un ambiente para la especificación de modelos BPEL de notable relevancia por su utilización. Otro trabajo relacionado es el correspondiente al análisis de *compensaciones* [20] sobre una extensión BPEL para modelar servicios web, que es propuesto en [81].

Otras revisiones exhaustivas de enfoques en materia de modelado y análisis de procesos de negocio se presentan en [35, 36]. En ellas se estudia la composición de servicios especificados en UML (Unified Modeling Language) mediante la utilización de Sistemas de Transición de Estados como formalismo subyacente para llevar a cabo el análisis.

Continuando con trabajos sobre lenguajes específicos, en [67] se propone una traducción semi-automática de Diagramas de Procesos de Negocios (BPD) a modelos especificados en TLA+ [59]. La traducción utiliza como lenguaje intermedio Redes de Petri y la misma permite aplicar la herramienta TLC (Temporal Logic Model Checker) al modelo traducido con el objetivo de analizar propiedades expresadas mediante fórmulas en TLA.

Con el propósito de fundar una base conceptual para la descripción y el análisis de procesos de negocios, en [98] se han identificado una serie de patrones para el modelado de procesos de negocios. En dicho trabajo se puede encontrar un estudio exhaustivo de diferentes perspectivas que forman parte en la especificación de *workflows*, y en cada un de ellas se enumeran diversos patrones que recrean situaciones presentes en dichas especificaciones. Como formalismo subyacente a la descripción de estos patrones se utiliza una extensión de Redes de Petri. Además, entre los objetivos de dicho proyecto, podemos mencionar el desarrollado el lenguaje YAWL (Yet Another Workflow Language) [91] para la especificación de *workflows*. Este lenguaje propone la especificación de modelos guiada por patrones y brinda el análisis de propiedades como ausencia de *deadlock*, y *soundness*. Es un lenguaje sumamente rico en poder expresivo, y existen diferentes trabajos que presentan traducciones de otros lenguajes a YAWL (en particular para aprovechar las herramientas de análisis para este último).

Como herramienta para la automatización del análisis de *workflows* podemos citar el trabajo [102], donde se presenta una técnica para el análisis de corrección en la definición de *workflows*. Como herramienta de soporte se provee una herramienta basada en la traducción de modelos de workflows a Redes de Petri. Además, en [101], y con el objetivo de reducir la complejidad de los modelos traducidos, se presentan diferentes reglas de reducción de modelos de Redes de Petri con extensiones de arcos de reinicio e inhibidores.

En relación a patrones de *workflows*, en [77] se propone una formalización de los mismos utilizando π-cálculo. Esta aplicación del formalismo permite el estudio de propiedades como *soundness* y sus variantes, mediante la caracterización de invariantes en términos de equivalencias por bisimulación de modelos.

Centrados en el análisis de *workflows*, en [51] se propone un método para la formalización de esquemas de *workflows* basado en la teoría de autómatas. Dicha formalización permite la definición de propiedades mediante las cuales se pueden evaluar y analizar la composición de estos esquemas.

Otro trabajo que explota el uso de herramientas para el análisis de *workflows* se presenta en [40]. Allí se propone la formalización de los Diagramas de Actividades de UML mediante PROMELA (Process Meta Language) para emplear SPIN [12] como herramienta para la verificación de propiedades.

Un enfoque alternativo para el análisis de *workflows* puede encontrarse en [99], donde se utiliza *Process mining*[1] para realizar análisis de *conformidad* (*compliance*) de procesos.

La importancia de descripciones *declarativas* de workflows se destaca en [75, 89], donde se propone DECLARE como herramienta para la especificación de workflows mediante restricciones entre los procesos o tareas que los conforman. Estas restricciones se expresan como fórmulas y tienen asociadas una semántica basada en alguna lógica temporal. Con esta caracterización de modelos de *workflows*, pueden realizarse diferentes análisis como por ejemplo, la verificación de consistencia.

1.2 Objetivos

Como describimos anteriormente, existen numerosos antecedentes de enfoques formales para la especificación de procesos de negocios, incluso muchos de ellos con soporte para el análisis automático de propiedades de procesos. Sin embargo, los enfoques existentes poseen limitaciones. En muchos casos, las limitaciones son de poder expresivo. EPC es uno de estos lenguajes. Estos lenguajes ofrecen mecanismos completamente automt́icos para análisis de sanidad de procesos, pero a costa de proveer un lenguaje menos expresivo, en los que muchos elementos útiles para la descripción de procesos de negocios, tales como regiones de cancelación (a describirse más adelante en este trabajo), no son soportados. En otros casos, notablemente en YAWL, se cuenta con un enorme poder expresivo, pero a costa de sufrir grandes dificultades para dar semántica a algunos elementos del lenguaje, y de perder automatización, o automatización abarcando la totalidad del lenguaje, para análisis de sanidad de procesos.

Una característica que suele ser en general dejada de lado en la especificación y análisis de procesos de negocios es la de brindar al usuario la posibilidad de describir cuáles son las propiedades de los procesos en las cuales está interesado. En efecto, las propiedades de sanidad de procesos suelen ser *generales*, en el sentido de que las mismas se aplican y exigen a todo proceso de negocios, pero si un usuario está interesado en asegurar que su proceso de negocios brinda un comportamiento particular, los lenguajes no cuentan con mecanismos para especificar y analizar tales comportamientos.

Finalmente, observamos que los lenguajes para la descripción de procesos de negocios suelen poner énfasis en la descripción de los *procesos*, no así en los *productos*

[1] *Process mining* es una técnica que se basa en el análisis de grandes volúmenes de datos, en particular de registros de eventos de procesos, para obtener información sobre el comportamiento de sistemas [95].

que se manipulan. Si bien la descripción de productos o datos es relevante, como se evidencia con la posibilidad de describir datos participantes en los procesos en las herramientas para procesos de negocios y workflows, estos productos y sus descripciones no tienen participación en los mecanismos de análisis, ni en las propiedades de sanidad vinculadas a procesos de negocios.

Basados en estas observaciones, los objetivos de este trabajo se resumen en los siguientes puntos:

- Desarrollar una semántica formal para lenguajes de especificación de procesos de negocios, que admita los niveles de expresividad de los lenguajes de especificaciones de procesos de negocios y workflows más expresivos, permitiendo capturar de manera menos compleja que las alternativas existentes algunos de los elementos de especificación más complejos desde el punto de vista semántico. Este objetivo se busca poniendo especial atención a la posibilidad de dar soporte de análisis automático, en cuanto a los chequeos de sanidad, y de los objetivos vinculados a la expresión de propiedades de interés de los usuarios, fuera de las de sanidad, que se describen más abajo.

- Complementar la semántica formal propuesta con la provisión a los usuarios de la posibilidad de describir propiedades de interés, fuera de aquellas vinculadas a sanidad. Los mecanismos de especificación de propiedades estarán centrados en el uso de lógicas temporales, complementadas con mecanismos de especificación que permitan simplificar la descripción de propiedades cuya expresión es compleja.

- Complementar la especificación de procesos de negocios con la posibilidad de capturar *productos*. Como mencionamos anteriormente, creemos que una característica importante de los procesos de negocios reside en la especificación de los elementos que éstos manipulan. En general, en los lenguajes para la especificación de procesos de negocios existentes, esta información es relegada a la especificación de variables que toman como parámetros los procesos del modelo. Por ello, estamos interesados en contar con un lenguaje en el cual la especificación de estos elementos, que llamamos *productos*, juegue un rol tan importante como el de los procesos que los manipulan. De esta manera, brindaríamos la posibilidad de complementar lenguajes existentes para la descripción de procesos con la capacidad de especificar modelos de sistemas donde la presencia de productos sea una condición necesaria. Además, extrapolando el objetivo del punto anterior, nos interesa contribuir al análisis de estos modelos mediante métodos basados en la especificación de propiedades.

Particularmente, brindar la posibilidad de verificar propiedades que incluyan especificaciones sobre productos como objetos del modelo.

- Produccir herramientas de soporte al proceso de especificacón de procesos de negocios. Basados en el avance de poder de cómputo que provee la constante evolución de la tecnología informática, un punto crucial que consideramos necesario para la utilización de los análisis propuestos, es su automatización. En este sentido, creemos que contar con métodos formales como herramientas subyacentes a nuestra propuesta es una condición necesaria. Más aún, como parte del trabajo proponemos el desarrollo de herramientas que permitan reducir la brecha entre la sencillez en la descripción de modelos y la formalidad de su análisis, motivando así una potencial aplicación en la industria. Para los lenguajes tratados brindaremos herramientas de especificación y análisis de procesos de negocios, como así también la posibilidad de construir descripciones de procesos de negocios a partir de las propiedades que los mismos deben cumplir.

Con el propósito de evitar cumplir con los objetivos arriba descriptos para lenguajes específicos, y perder la oportunidad de producir resultados más generales, realizamos un análisis de los lenguajes existentes, tratando de identificar elementos comunes a los mismos, y trabajar sobre éstos desde una perspectiva general (aunque usemos alguna sintaxis concreta específica). Estas tareas son evaluadas en este trabajo a través del desarrollo de casos de estudio tomados de la literatura del área, en lo que respecta a descripción de procesos y chequeos de sanidad de los mismos, y en base a variantes de estos casos en las cuales se enfatiza la necesidad de contar con propiedades descriptas por el usuario, y de dar más relevancia a la descripción de los productos manipulados por los procesos de negocios.

1.3 Organización

Este trabajo se presenta de la siguiente manera:

En el siguiente capítulo definimos varios conceptos que serán necesarios para abordar el estudio de la presente tesis. Inicialmente describimos modelos de comportamiento para Sistemas Reactivos. A partir de allí, y con la finalidad de expresar propiedades de interés sobre estos modelos, se describen diferentes lógicas modales [15]. Finalmente presentamos el *Model Checking*, donde los lenguajes para la descripción de Sistemas Reactivos y las lógicas para la especificación de propiedades de estos modelos confluyen [10]. Como instanciación de esta técnica examinamos

dos herramientas automáticas para la verificación de modelos, las que utilizaremos en este trabajo.

En el segundo capítulo describimos el contexto de este trabajo tomando como marco los Procesos de Negocios y centrando la atención en la descripción de los mismos mediante *Workflows*. Damos una perspectiva propia de los lenguajes para el modelado de los mismos, con la intención de brindar al lector una mirada general de los mismos. Sobre el final, y como principal motivación de este trabajo, mostramos la importancia del análisis de workflows con énfasis en la automatización, basado en *métodos formales*.

Utilizamos el tercer capítulo para trabajar sobre uno de los objetivos de la tesis: el análisis de propiedades de *workflows* capturados a través de sistemas de transición de estados. En dicho capítulo tomamos como partida un lenguaje para modelar *workflows*, y luego de detallar la traducción de sus modelos a sistemas de transición de estados, mostramos las ventajas en la utilización de una lógica temporal para describir propiedades de modelos. Además se muestran herramientas que automatizan este proceso y que asisten al usuario en la especificación de propiedades. Estas propiedades pueden ser verificadas por medio de *model checking*. Además, en este capítulo, explotando la caracterización abstracta de *workflows* que nos brinda la traducción detallada, proponemos un proceso para construir workflows a partir de refinamientos basados en una clase de propiedades temporales. Al final de este capítulo y con el objetivo de abordar la especificación de ciertas propiedades temporales, proponemos una extensión de la lógica temporal utilizada junto con un método para verificar propiedades expresadas con la misma.

En los Capítulos 5 y 6 nos focalizamos en una característica que consideramos importante en el modelado de procesos de negocios y que en gran medida los lenguajes actuales relegan o no tienen en cuenta. La característica que abordamos como objetivo en estos capítulos es es la posibilidad de contar en el lenguaje con un elemento que permita especificar *productos* (una referencia a un elemento empírico con propiedades) como un referente explícito del modelo. En el Capítulo 5 retomamos un lenguaje formal para la descripción de procesos de negocios que contempla este aspecto, detallando y aportando una sintaxis que permita su utilización. Luego en el Capítulo 6 proponemos el análisis de modelos en este lenguaje utilizando un método similar al del Capítulo 4 pero en este caso considerando tiempo real. Así las propiedades a verificar no sólo pueden considerar, a diferencia de las del Capítulo 4, situaciones que involucren especificaciones de productos, sino también pueden tener referencias a restricciones de tiempo real sobre los modelos. Para finalizar el Capítulo 6, describimos una semántica alternativa para PPML basada en π-Cálculo con el objetivo de analizar en los modelos del lenguaje el aspecto de movilidad que dicho formalismo provee.

Finalmente, en el Capítulo 7 resumimos los objetivos alcanzados que consideramos contribuciones y detallamos líneas de investigación futuras relacionadas con este trabajo.

La descripción detallada de los lenguajes para el modelado de procesos de negocios mencionados a lo largo del trabajo, pueden encontrarse en el Apéndice A.

Capítulo 2

Preliminares

2.1 Introducción

Este capítulo define varios conceptos que serán necesarios para abordar el estudio de la presente tesis, y establece la notación para los mismos. No pretende ser una presentación completa de cada tema abordado en la tesis. En cambio, presenta las nociones más importantes utilizadas en el trabajo, generalmente a través de definiciones formales, y recurriendo también a explicaciones informales para una mejor comprensión. Cada tema estará acompañado de citas a trabajos relevantes al respecto, para que el lector pueda acceder a una presentación completa, en caso de requerirlo.

Inicialmente se describen modelos de comportamiento para Sistemas Reactivos, comenzando con Sistemas de Transición de Estados y Redes de Petri, para luego abarcar modelos más complejos que involucran la noción de tiempo, como es el caso de Autómatas Temporizados. A partir de allí, y con la finalidad de expresar propiedades de interés sobre estos modelos (y más adelante analizar la validez de las mismas), se describen diferentes lógicas modales [15] para la especificación de propiedades, como la Lógica Temporal de Tiempo Lineal, en la cual las ejecuciones del sistema de capturan con secuencias de estados y se cuenta con operadores para predicar sobre estados futuros a un momento determinado en la ejecución. Se presenta además una variante de esta lógica, apropiada para sistemas basados en eventos, que incorpora una suerte de estados abstractos denominados *fluentes*. A diferencia de estas lógicas donde la ejecución de un sistema interpreta linealmente, como una secuencia de estados, en la *Computational Tree Logic* (CTL) la ejecución de un sistema se interpreta como una ramificación de estados futuros posibles, esto es, a cada estado pueden seguirle diferentes estados sucesores.

Finalmente nos enfocamos en el problema de analizar la validez de una propiedad

en un modelo, en particular en el análisis automático de propiedades de modelos. Presentamos la técnica de *verificación de modelos* (*Model Checking*), donde los lenguajes para la descripción de Sistemas Reactivos y las lógicas para la especificación de propiedades de estos modelos confluyen [10]. Como instanciación de esta técnica examinamos dos herramientas automáticas para la verificación de modelos, las que utilizaremos en este trabajo. Una de ellas es LTSA (*Labelled Transition System Analyser*), que utiliza sistemas de transición de estados (más específicamente, sistemas de transiciones etiquetadas de estados) para describir los modelos a verificar, y la Lógica Temporal de Tiempo Lineal para la especificación de propiedades. La segunda herramienta que analizaremos es *Uppaal*, que incorpora a su análisis la noción de tiempo real. Para ello, utiliza Autómatas Temporizados para la descripción de sus modelos y brinda como lenguaje de descripción de propiedades a un subconjunto de *Timed Computational Tree Logic* (una extensión de CTL que incorpora tiempo real).

2.2 Modelos de Comportamientos para Sistema Reactivos

En los comienzos de las Ciencias de la Computación, la idea de sistema, como proceso automatizado, estuvo asociada a la noción de función o cálculo, esto es, dados un conjunto de datos entrada (*input*) que alimenta el proceso, éste luego de procesarlos brinda un conjunto de datos como resultado (*output*). Bajo este enfoque, la terminación es una propiedad deseable de todos los sistemas, y la correctitud puede capturarse mediante la relación deseada entre entradas y salidas (diferentes trabajos importantes, como [27, 44], profundizan sobre estos aspectos).

A partir de la década del 80 surgió la necesidad de caracterizar otro tipo de sistemas, aquellos en los cuales el comportamiento no depende exclusivamente de una serie de datos iniciales, sino que tienen un comportamiento dinámico asociado a la ocurrencia de eventos y la reacción a los mismos, y la interacción con el ambiente en los que se encuentran. El término de *Sistemas Reactivos*, como se denomina a este tipo de sistemas, fue introducido por David Harel y Amir Pnueli [41]. En un sentido general, en [109] se define un sistema reactivo de la siguiente manera.

Definición 2.1. ***Sistema Reactivo***

Un Sistema Reativo es un sistema que, cuando se encuentra activo, es capaz de modificar el entorno a través de la habilitación, reacción o restricción de eventos del mismo.

A diferencia de los sistemas funcionales o transformacionales, los sistemas reacti-

vos tienen características particulares entre las que podemos mencionar las siguientes:

- Son procesos continuos (no se espera su terminación) con alta interacción con el entorno.
- La interacción con el entorno se basa en la ocurrencia de eventos, los cuales el sistema debe estar preparado para procesar.
- El comportamiento de un sistema reactivo, en general, está caracterizado por la interacción de un conjunto de procesos paralelos o concurrentes que lo componen.
- Algunos sistemas pueden tener restricciones temporales. A este tipo particular de sistemas reactivos se los denomina *Sistemas de Tiempo Real*.

Algunos ejemplos sistemas reactivos son los diferentes sistemas de servicios en ambientes distribuidos (web servers, servidores de correo, servidores FTP); también los *embedded systems*, sistemas incrustados, con gran auge en estos tiempos ya que se encuentran en un gran número de dispositivos electrónicos, son sistemas reactivos.

Para la especificación de este tipo de sistemas se necesitan lenguajes cuyo poder expresivo se enfoque fuertemente en capturar los elementos o procesos que componen a los mismos, su combinación e interacción. A continuación introducimos tres lenguajes para modelar sistemas reactivos, los cuales serán utilizados en este trabajo. Inicialmente presentamos los *Sistemas de Transición de Estados*, que representan un modelo muy rico para la descripción de sistemas reactivos a través de la especificación de componentes que pueden comunicarse mediante la sincronización de eventos comunes. Luego introducimos las *Redes de Petri*, modelo fuertemente utilizado para describir sistemas concurrentes. Finalmente abordaremos los *Autómatas Temporizados* que, como complemento, permiten incorporar en sus modelos restricciones de tiempo real.

Como referencias para una lectura detallada de los temas introducidos, proponemos al lector recurrir a [1, 45, 62, 65, 70, 71, 109].

2.2.1 Sistemas de Transición de Estados

Un *Sistema de Transiciones Etiquetadas de Estados* (LTS), o simplemente *Sistema de Transición de Estados*, es un tipo de máquina abstracta cuyos modelos describen sistemas como un conjunto de componentes caracterizados mediante estados y transiciones entre ellos. En las Ciencias de la Computación, entre otras aplicaciones, se utilizan para modelar el comportamiento de *sistemas reactivos* [62]. En este

contexto, las transiciones representan eventos del sistema y están etiquetadas con nombres (etiquetas o acciones) de un conjunto dado. Además la sincronización entre distintos componentes se puede especificar por medio de *eventos compartidos*, en este caso, identificados con el mismo nombre. De esta manera, el comportamiento del sistema completo puede obtenerse mediante la *composición paralela* de sus componentes. Cabe destacar que por composición paralela nos referimos al *interleaving* de las acciones de las componentes, es decir, a todas las ejecuciones posibles del sistema producto de intercalar (combinar) los comportamientos de sus componentes.

Definición 2.2. ***Sistema de transiciones etiquetadas de estados***

Un Sistema de transiciones etiquetadas de estados (LTS) $\mathcal{S}$ es una cuatro-upla $\langle Q, A, \delta, q_0 \rangle$, donde:

- *Q es un conjunto finito de estados,*
- *A es el* alfabeto *de $\mathcal{S}$,*
- *$\delta \subseteq Q \times A \cup \{\tau\} \times Q$ es la* relación de transición *entre los estados de $\mathcal{S}$,*
- *q_0 es el estado inicial.*

De manera informal, la semántica de un LTS $\mathcal{S}$ es su conjunto de *ejecuciones*, es decir, el conjunto de secuencias de eventos (trazas de ejecución) que $\mathcal{S}$ puede realizar partiendo de su estado inicial y respetando δ.

Los sistemas de transición de estados pueden describirse gráficamente como grafos, teniendo a los estados como nodos y las transiciones como aristas. Como ejemplo, en la Figura 2.1 se muestra el LTS correspondiente a una versión simple de una máquina expendedora de té o café con las siguientes características: la máquina sólo acepta monedas de 1 y 2 pesos, el té cuesta 1 peso y el café cuesta 2 pesos.

Finite States Processes

El modelado de sistemas concurrentes utilizando máquinas de estados, como los LTS, se torna inmanejable rápidamente debido al número de estados y transiciones necesarios para describir sistemas complejos. Esto ha motivado el desarrollo de lenguajes de especificación alternativos, como por ejemplo *Finite States Processes* (FSP) [62]. El objetivo de este lenguaje es brindar una sintaxis clara, concisa y flexible para la especificación de sistemas reactivos. FSP es un álgebra de procesos, similar a las desarrolladas por Milner (*Calculus of Communicating Systems (CCS)* [70]) y Hoare (*Communicating Sequential Processes (CSP)* [45]), para la especificación y el razonamiento sobre sistemas concurrentes.

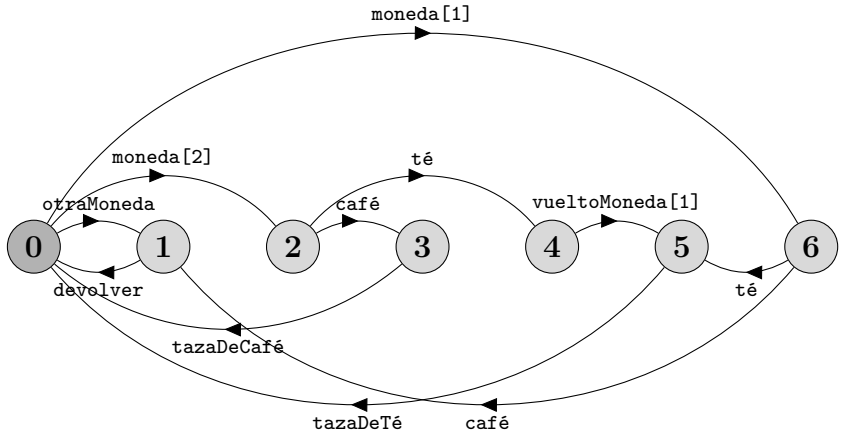


Figura 2.1: Especificación de una máquina expendedora de té o café, a través de un sistema de transición de estados.

Una especificación FSP está compuesta por dos conjuntos de definiciones de procesos: *primitivos* y *compuestos*. A continuación se enumeran todas las construcciones sintácticas básicas de FSP para la descripción de cada uno de ellos:

- *Procesos primitivos*: Los procesos primitivos son el bloque de construcción básico del lenguaje. Éstos están formados por uno o mas *procesos locales* delimitados por comas y finalizan con un punto. Los identificadores de procesos deben comenzar con mayúscula. Existen tres procesos primitivos locales especiales: `END`, `STOP` y `ERROR`.

- *Prefijado de acciones*: El prefijado de acciones, anotado como "`->`", es la forma canónica para describir el comportamiento de procesos. Describe los eventos que caracterizan o con los cuales sincroniza un proceso primitivo. Los nombres de acciones se anteponen al del proceso y deben ser identificadores que comiencen con minúscula.

- *Elección*: El operador de elección "`|`" permite especificar un punto de bifurcación del comportamiento de un proceso, es decir, un punto en el cual el comportamiento de un proceso varía según la ocurrencia de ciertos eventos. Consideremos `a` y `b`, dos acciones y `P` y `Q`, dos procesos, entonces `(a->P | b->Q)` es un proceso que ante la ocurrencia de `a` (resp. `b`) se comporta como `P` (resp. `Q`).

- *Elecciones condicionales*: Para condicionar el comportamiento de procesos podemos utilizar las cláusulas `when` o `if`. Por ejemplo, en `(when (B) a ->P | b ->Q)`, `a` puede ser elegida sólo si `B` (expresión lógica) es verdadera.

- *Recursión*: De manera similar a los lenguajes de programación, la recursión en FSP permite la definición de un proceso en términos de si mismo. En el ejemplo de la Figura 2.2 podemos observar la especificación de un proceso mediante recursión. En el mismo también están presentes los conceptos de *parámetro* (línea 1) e *indexación* (líneas 2 y 3). Este último recurso puede ser aplicado tanto a procesos como a eventos.

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

Figura 2.2: Especificación FSP de un *Buffer* limitado.

- *Procesos compuestos*: A diferencia de los procesos primitivos, un proceso compuesto se define por la composición paralela de dos o más procesos. Este tipo de composición permite la descripción de sistemas concurrentes, los cuales se denotan anteponiendo `||` al nombre del proceso.

- *Composición paralela*: La composición paralela de dos procesos define un nuevo proceso que contiene todos los posibles *interleavings* de acciones de cada uno de sus constituyentes. Cabe destacar que si dos procesos comparten una misma acción (llamada con el mismo nombre de forma directa o mediante renombramiento), entonces los procesos deben "colaborar" para ejecutarla de forma simultánea (se sincronizan). La composición paralela se denota con el operador"`||`".

- *Instanciación*: Para generar diferentes instancias de un proceso, FSP brinda diferentes mecanismos:

  Etiquetado (*labeling*): tiene el efecto de prefijar con un identificador a todos los nombres en el alfabeto de un proceso. La expresión `a:P` prefija todos los nombres del alfabeto de `P` con `a`.

  Replicación: es equivalente a múltiples composiciones paralelas. Ésta se simboliza mediante la palabra reservada `forall`. Su sintaxis es la siguiente: `forall [i:1..N] P(i)`.

En pos de entender la notación, notar que las siguientes declaraciones definen exactamente el mismo proceso:

```
1 ||P1 = (a[0]:X || a[1]:X || a[2]:X || a[3]:X).
2 ||P2 = a[0..3]:X.
3 ||P3 = forall[i:0..3] a[i]:X.
```

- *Re-etiquetado* (*re-labeling*): Esta operación permite renombrar las acciones del alfabeto de un proceso. Se denota mediante "/" y su sintaxis es la siguiente: `/` { `new_label`$_1$`/old_label`$_1$`,... , new_label`$_n$`/old_label`$_n$}.
- *Ocultación* (*hiding*): Esta operación, denotada como "\", permite ocultar acciones del alfabeto de un proceso al resto del sistema. Dado la especificación \{`a1,... , ax`} en la definición de un proceso, las acciones `a1,... , ax` se reemplazan por τ (acción silenciosa).
- *Interfaz* (*interface*): Esta operación es complementaria respecto a la anterior. Se simboliza mediante "`@`" y oculta todas las acciones que no figuren como parte de la *interfaz*.

Como ejemplo, en la Figura 2.3 se especifica un sistema compuesto por un *buffer* (ver Figura 2.2) y dos procesos, productor y consumidor. Cabe destacar que `Size` es una constante predefinida, que indica la capacidad deseada del *buffer*. El sistema de transición de estados correpondiente a la especificación anterior se muestra en la Figura 2.4.

```
1 PROD = (produce ->PROD).     CONS = (consume ->CONS).
2 || SYSTEM = (PROD || BUFFER(Size) || CONS)
3 /{ put[0..Size-1]/produce, get[1..Size]/consume }.
```

Figura 2.3: Especificación de un sistema compuesto por un *buffer* acotado, un productor y un consumidor.

2.2.2 Redes de Petri

La teoría de Redes de Petri data de la década del 60, y nace como propuesta de un formalismo para caracterizar el comportamiento dinámico de sistemas concurrentes

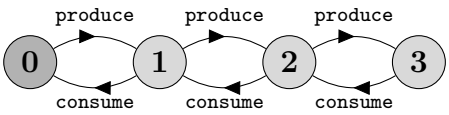


Figura 2.4: LTS correspondiente la especificación FSP de la Figura 2.4 con `Size=3`

discretos. Además de poseer las Redes de Petri una semántica operacional, también brindan una notación gráfica, ampliamente conocida, que permite describir fácilmente tanto la estructura (estática) de un sistema concurrente (en términos de sus estados y cómo éstos están organizados), el estado dinámico de un sistema, y su evolución en tiempo de ejecución.

Definición 2.3. ***Red de Petri***

Una Red de Petri es una tripla (P, T, F) *donde* P *es un conjunto finito de sitios (places),* T *es un conjunto finito de transiciones (transitions) con* $T \cap P = \varnothing$*, y* $F \subseteq (P \times T) \cup (T \times P)$ *es la relación de flujo entre los sitios y transiciones.*

Los sitios modelan variables del sistema y la transiciones representan eventos cuya ocurrencia provoca la modificación del estado del sistema. Los elementos que forman parte de una Red de Petri poseen, como mencionamos, una notación gráfica, que se muestra en la Figura 2.5.

Figura 2.5: Representación gráfica de los componentes de un Red de Petri

Consideremos t una transición y P un conjunto de sitios. Los sitios $p_i \in P$ conectados hacia t ($p_i \rightarrow t$) son considerados como las entradas (*input*) de t, mientras que los sitios $p_j \in P : t \rightarrow p_j$ son sus salidas (*output*). La semántica operacional de una Red de Petri es descripta en término de *tokens* que representan cómo el flujo de control transcurre en un proceso. Una Red de Petri puede contener tantos *tokens* como sean necesarios para caracterizar los subprocesos (*threads*) que coexisten en la misma. Una distribución M de *tokens* en una Red de Petri es una función $M : P \rightarrow \mathbb{N}$, denominada *marking*, y representa un estado de la misma. Podemos definir un orden $\geq$ sobre el conjunto de todos los posibles markings de una red de Petri, de la siguiente manera: $M_1 \geq M_2 \Leftrightarrow \forall p \in P : M_1(p) \geq M_2(p)$.

La semántica operacional de una Red de Petri está definida mediante la noción de *ejecución o disparo de una transición* (*firing*): una transición podrá ejecutarse siempre que exista *como mínimo* un *token* por cada sitio de entrada de la misma. La ejecución de una transición t es un evento atómico que instantáneamente produce un cambio de estado de la Red, consumiendo un *token* por cada sitio de entrada y produciendo uno en cada sitio de salida conectado a t.

Gráficamente podemos observar la semántica de redes de Petri en la Figura 2.6, donde se ilustran diferentes configuraciones de una Red de Petri con una transición t y, en cada caso, el correspondiente estado posterior a la ejecución de t. Por ejemplo, para la configuración (d) se especifica que partiendo de un estado (*marking*) $M_1 = [2, 1, 1, 0]$ ($p_1 = 2$, $p_2 = 1$, $p_3 = 1$ y $p_4 = 0$) y mediante la ejecución de t, se alcanza el estado $M_2 = [1, 0, 2, 1]$. El conjunto de todos los posibles estados de una red N lo denotaremos como $\mathcal{M}(N)$ y los cambios de estados mediante la siguiente notación:

- $M_1 \xrightarrow{t} M_2$: desde un estado M_1 alcanzamos M_2 como consecuencia de la ejecución de t.
- $M_1 \rightarrow M_2$: Existe alguna transición que, mediante su ejecución, partiendo de un estado M_1 alcanzamos M_2.
- $M_1 \xrightarrow{\sigma} M_n$: Mediante la secuencia de ejecuciones de transiciones $\sigma = t_1, t_2, ..., t_{n-1}$, alcanzamos desde el estado M_1 al estado M_2, es decir: $M_1 \xrightarrow{t_1} M_2 \xrightarrow{t_2} M_3 \ldots \xrightarrow{t_{n-1}} M_n$.
- $M_1 \xrightarrow{*} M_n$: M_n es *alcanzable* desde M_1, es decir, existe una secuencia de transiciones σ tal que $M_1 \xrightarrow{\sigma} M_n$.

La utilización de este formalismo para la descripción de sistemas es alentada por diferentes análisis abstractos de sus modelos. Si bien existe una gran cantidad de trabajos sobre análisis de Redes de Petri (cf. [38]), sólo enunciaremos algunas de las principales propiedades de las mismas.

- Una Red de Petri está *viva* (*live*) si y sólo si, para todo estado alcanzable M y toda transición t, existe M' tal que $M \xrightarrow{*} M'$ y t está habilitada en M'.
- Una Red de Petri está *acotada* (*bounded*) si y sólo si, existe n tal que para todo estado alcanzable M, $M(p) \leq n$ para todo sitio p.
- Una estado M de una Red de Petri es *cubrible* (*coverable*) si existe un estado alcanzable M' tal que $M' \geq M$.

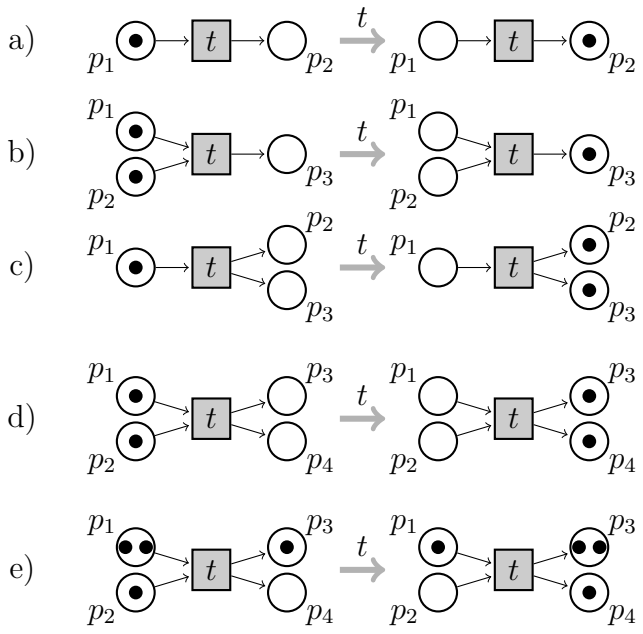


Figura 2.6: Semántica Operacional de una Red de Petri

- Una Red de Petri es *fuertemente conexa* (*strongly connected*) si y sólo si, para todo par de elementos (sitios o transiciones) x e y, existe un *camino dirigido* de x a y.

Debido a diferentes limitaciones de las Redes de Petri para la representación de ciertos aspectos de sistemas concurrentes, como por ejemplo la representación de información, han surgido variadas extensiones de las mismas, tales como las Redes de Petri Coloreadas [50] y las Redes de Petri Temporizadas [104]. Una de las extensiones que mencionaremos en este trabajo son la Redes de Petri con Reinicio [6].

Definición 2.4. *Reset Net*

Una Red de Petri con Reinicio (Reset Net) es una cuádrupla (P, T, F, R) donde (P, T, F) es una Red de Petri y $R : T \to 2^P$, es una función que especifica los arcos de reinicio para cada transición $t \in T$.

La semántica operacional no cambia respecto de la Red de Petri subyacente, esto es, las reglas para la ejecución de las transiciones se mantiene. La diferencia reside en que cuando un transición es ejecutada, *todos* los sitios asociados a la misma por medio de un arco de reinicio son "vaciados", es decir, sus *tokens* son eliminados. La

notación gráfica para los arcos de reinicio es: $\twoheadrightarrow$. Como ejemplo, en la Figura 2.7 se analizan dos configuraciones, donde la primera (a) ilustra la neutralidad de los arcos de reinicio con respecto a la habilitación para la ejecución de una transición y en (b), se puede observar cómo el sitio p_3 asociado a t mediante un *arco de reinicio* es vaciado luego de la ejecución de t.

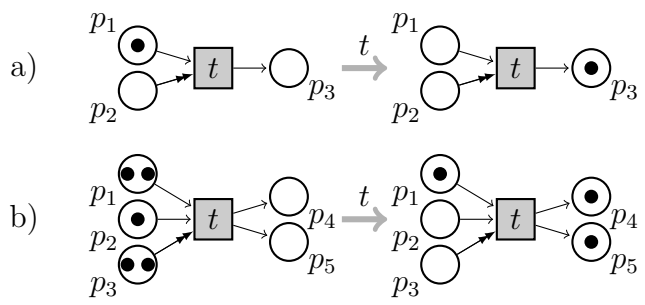


Figura 2.7: Semántica Operacional de una Red de Petri con arcos de reinicio

2.2.3 Autómatas Temporizados

Los *autómatas termporizados* fueron introducidos a principios de la década del 90 [4, 5] como un formalismo que permite capturar en sus modelos, restricciones temporales. En términos conceptuales y enfocados en su aplicación a la representación de sistemas reactivos, un autómata temporizado es un autómata extendido con un conjunto de variables que representan el paso del tiempo, denominadas *relojes*. Estas variables pueden ser consultadas, por ejemplo para analizar lapsos de tiempo entre dos eventos del sistema o para establecer restricciones de comportamiento del mismo; para ello, pueden asociarse condiciones temporales (que involucran valores de los relojes) a las transiciones (eventos) del sistema. Si bien el paso del tiempo es uniforme para el conjunto de relojes, éstos pueden ser reiniciados (puestos en 0) en cualquier momento. De esta manera, el valor de una variable reloj representa el tiempo transcurrido desde su última inicialización. La noción de tiempo puede comprenderse como contínua o discreta. Para los fines de este trabajo, será suficiente considerar el tiempo de manera discreta. La siguiente definición, sin embargo, abarca tanto automátas temporizados con tiempo contínuo como de tiempo discreto.

Definición 2.5. ***Autómata Temporizado***

Un Autómata Temporizado es una tupla $\langle L, l_0, C, A, E, I\rangle$, donde L es un conjunto de estados, $l_0 \in L$ es el estado inicial, C es un conjunto de clocks (relojes), A es

un conjunto de acciones (incluida la acción interna τ), $E \subseteq L \times A \times B(C) \times 2^C \times L$ es un conjunto de arcos entre estados y finalmente $I : L \to B(C)$ es una función que asocia a los estados con una fórmula (Invariante) que deben preservar.

Los arcos están etiquetados con:

- *una acción.*
- *una condición o guarda.*
- *un conjunto de clocks a ser reiniciados.*

La semántica formal de un autómata temporizado $\langle L, l_0, C, A, E, I\rangle$ está definida como un LTS $\langle S, s_0, \to\rangle$, donde $S \subseteq L \times \mathbb{R}^C$ es el conjunto de estados, $S_0 = (l_0, u_0)$ es el estado inicial, y $\to \subseteq S \times (\mathbb{R}_{\geq 0} \cup A) \times S$ es la relación de transición, que satisface lo siguiente:

- $(l, u) \xrightarrow{d} (l, u + d)$, si $\forall d' : 0 \leq d' \leq d \implies u + d' \in I(l)$
- $(l, u) \xrightarrow{a} (l', u')$, si existe $e = (l, a, g, r, l') \in E$ tal que $u \in g$, $u' = [r \mapsto 0]u$, y $u' \in I(l')$

donde para $d \in \mathbb{R} \geq 0$, $u+d$ asigna a cada *clock* x perteneciente a C el valor $u(x)+d$, y $[r \mapsto 0]\, u$ denota la valuación donde a cada *clock* en r se le asigna el valor 0 y los demás $C \backslash r$ coinciden con u.

La semántica establece que los cambios de estados se suceden o bien, por el paso del tiempo, o bien, porque tiene lugar una acción (transición en el autómata temporizado). En ambas situaciones los invariantes de estado deben mantenerse y además, en caso de la ocurrencia de una acción, deben reflejarse los efectos de la misma.

Como lo mencionamos, los sistemas reactivos pueden estar conformados por un conjunto de componentes que interactúan mediante alguna forma de comunicación. Esta necesidad de composición de elementos es capturada por la noción de *Red de Autómatas Temporizados*. Estas redes permiten modelar un sistema compuesto a partir de un conjunto de autómatas que coexisten. Además, estos autómatas cuentan con la posibilidad de interactuar mediante algún tipo de comunicación entre los mismos.

Definición 2.6. ***Red de Autómatas Temporizados***

Dados un conjunto de relojes y de acciones en común, una Red de Autómatas Temporizados sobre ellos es un conjunto de n autómatas temporizados $\mathcal{A}_i = \langle L_i, l_i^0, C, A, E_i, I_i\rangle, 1 \leq i \leq n$. Sus estados están formados por vectores de estados $\vec{l} = \langle l_1, \ldots, l_n\rangle$ y la funciones de invariante están compuestas en una que opera sobre vectores de la siguiente manera $I(\vec{l}) = \bigwedge_i I_i(l_i)$.

La semántica de una Red de autómatas temporizados $\mathcal{A}_i = \langle L_i, l_i^0, C, A, E_i, I_i \rangle$ de n autómatas con vector (estado) inicial $\vec{l_0} = (l_1^0, \ldots, l_n^0)$ está definida como un LTS $\langle S, s_0, \rightarrow \rangle$, donde $S \subseteq (L_1 \times \ldots \times L_n) \times \mathbb{R}^C$ es el conjunto de estados, $S_0 = (\vec{l_0}, u_0)$ es el estado inicial, y $\rightarrow \subseteq S \times S$ es la relación de transición definida por:

- $(\vec{l}, u) \xrightarrow{d} (\vec{l}, u + d)$, si $\forall d' : 0 \leq d' \leq d \implies u + d' \in I(\vec{l})$
- $(\vec{l}, u) \xrightarrow{a} (\vec{l}[l_i'/l_i], u')$, si existe $l_i \xrightarrow{\tau, g, r} l_i' \in E$ tal que $u \in g$, $u' = [r \mapsto 0]u$, y $u' \in I(\vec{l}[l_i'/l_i)$
- $(\vec{l}, u) \xrightarrow{a} (\vec{l}[l_j'/l_j, l_i'/l_i], u')$, si existe $l_i \xrightarrow{c?, g_i, r_i} l_i'$ y $l_j \xrightarrow{c!, g_j, r_j} l_j' \in E$ tal que $u \in (g_i \wedge g_j)$, $u' = [r_i \cup r_j \mapsto 0]u$, y $u' \in I(\vec{l}[l_j'/l_j, l_i'/l_i])$

donde $\vec{l}[l_i'/l_i]$ denota el reemplazo de l_i por l_i' dentro del vector $\vec{l}$.

Además de la generalización con respecto a la semántica de un autómata temporizado, el tercer punto especifica el funcionamiento de la comunicación. Particularmente en este caso, la misma se realiza a través de canales comunes de sincronización con las primitivas de lectura ? y escritura ! sobre los mismos.

A modo de ilustración, a continuación, en la Figura 2.8, mostramos como ejemplo un modelo simple de un *mouse* donde se pretende capturar la funcionalidad de *doble clic*. Asumimos que el intervalo de tiempo para que dos *clics* sean considerados como un clic doble, es de 3 unidades de tiempo.

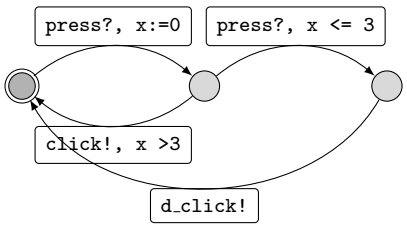


Figura 2.8: Modelo de Autómata Temporizado para *clics* de un *mouse*.

Como podemos observar, el autómata consta de 3 estados. Una vez posicionados en el estado inicial, cuando el usuario presiona el botón del *mouse*, pasamos al siguiente estado inicializando el reloj `x` en 0. Este reloj mide el tiempo hasta la próxima vez que presiona el botón del mouse. Si esto sucediera en un perído inferior al las 3 unidades de tiempo (último estado), se publica el evento `d_click`. Si, en cambio, transcurrieran más unidades de tiempo, se publica un simple `click`.

2.3 Lógicas

Con el objetivo de razonar sobre propiedades de sistemas, necesitamos un lenguaje lo suficientemente expresivo que permita describirlas formalmente para analizar su verdad o falsedad. Comenzamos introduciendo brevemente la Lógica Proposicional y luego presentamos extensiones modales de la misma que nos brindan la expresividad necesaria para caracterizar una gran cantidad de propiedades que nos interesan.

2.3.1 Lógica Proposicional

La Lógica Proposicional (LP) nos brinda una forma abstracta y formal de caracterizar condiciones o circunstancias del mundo que necesitamos analizar a través de *proposiciones* y *relaciones* entre ellas. Las *fórmulas* se construyen a partir de proposiciones *atómicas* y operadores lógicos como por ejemplo $\wedge$ (*conjunción*), $\vee$ (*disyunción*), $\neg$ (*negación*) y $\rightarrow$ (*implicación*). La proposiciones atómicas no tienen estructura interna y las mismas pueden ser *verdaderas* o *falsas*.

Definición 2.7. ***Sintaxis de la Lógica Proposicional***
El conjunto $\mathcal{F}_{\mathcal{LP}}$ de fórmulas de la Lógica Proposicional se define como el mínimo conjunto que satisface lo siguiente:

- *Todo elemento del conjunto $\mathcal{P}$ de variables proposicionales es una fórmula.*
- *Si ϕ es una fórmula de $\mathcal{F}_{\mathcal{LP}}$, entonces $\neg\phi$ también lo es.*
- *Si ϕ y ψ son fórmulas de $\mathcal{F}_{\mathcal{LP}}$, entonces $\phi \wedge \psi$, $\phi \vee \psi$, y $\phi \rightarrow \psi$ también lo son.*

Una vez establecida la definición de la sintaxis de las fórmulas, cuya gramática podemos expresar de manera sinóptica en *Backus Naur form* (BNF) [55] como

$$\phi ::= p \mid (\neg\phi) \mid (\phi \wedge \phi) \mid (\phi \vee \phi) \mid (\phi \rightarrow \phi),$$

podemos definir la semántica de esta lógica, mediante la función de interpretación.

Definición 2.8. ***Interpretación***
Sea $\phi \in \mathcal{F}_{\mathcal{LP}}$ una fórmula de la Lógica Proposicional y consideremos P_ϕ el conjunto de variables proposicionales de ϕ. Una interpretación de ϕ es una función total $I_\phi : P_\phi \rightarrow \{V, F\}$ que asigna un posible valor verdadero *(V) o* falso *(F) a cada variable proposicional de ϕ.*

Dada una interpretación I_ϕ para $\phi \in \mathcal{F}_{\mathcal{LP}}$, la semántica v_I de la fórmula ϕ, es decir, su valor de verdad se define inductivamente de la siguiente manera:

$$
\begin{array}{ll}
v_I(\phi) = I_\phi(\phi) & \text{si } \phi \in \mathcal{P} \text{ (es una variable proposicional)} \\
v_I(\neg\phi) = V & \text{si } v_I(\phi) = F \\
v_I(\neg\phi) = F & \text{si } v_I(\phi) = V \\
v_I(\phi \wedge \psi) = V & \text{si } v_I(\phi) = V \text{ y } v_I(\psi) = V \\
v_I(\phi \wedge \psi) = F & \text{si } v_I(\phi) = F \text{ o } v_I(\psi) = F \\
v_I(\phi \vee \psi) = F & \text{si } v_I(\phi) = F \text{ y } v_I(\psi) = F \\
v_I(\phi \vee \psi) = V & \text{si } v_I(\phi) = V \text{ o } v_I(\psi) = V \\
v_I(\phi \rightarrow \psi) = F & \text{si } v_I(\phi) = V \text{ y } v_I(\psi) = F \\
v_I(\phi \rightarrow \psi) = V & \text{si } v_I(\phi) = F \text{ o } v_I(\psi) = V.
\end{array}
$$

2.3.2 Lógicas Temporales

Las Lógicas Temporales son variantes de las *Lógicas Modales*. Estas últimas son extensiones de la Lógica Proposicional que, a diferencia de ella, expresan la distinción entre lo que es *necesariamente* verdadero y lo que es *posiblemente* cierto. Para ello hacen uso de operadores que expresan *modalidades*.

En este sentido las lógicas temporales brindan el soporte formal para describir y razonar sobre sistemas en los cuales la veracidad de las propiedades a analizar varía con el *tiempo*. Así las modalidades de las lógicas temporales permiten expresar que una propiedad *siempre* (durante todo el tiempo) se cumple, o que una propiedad *en algún momento* se cumple.

Si bien existen diferentes lógicas temporales y extensiones de sobre ellas, presentamos únicamente aquellas que utilizamos en el desarrollo del presente trabajo. Particularmente analizamos dos Lógicas Temporales que difieren en la observación del tiempo. Una de ellas es la *Linear-time Temporal Logic* (LTL). En ella, para cada momento sólo hay un estado futuro siguiente posible. Por otro lado en la *Computational Tree Logic* (CTL), el tiempo se comprende como una ramificación donde el tiempo puede dividirse en cursos alternativos que representan diferentes futuros posibles. Las modalidades que proveen cada una de ellas difieren en cuanto a la semántica, debido a estructura del tiempo subyacente. Así, por ejemplo, en LTL podemos expresar la idea de próximo estado ya que es único, mientras que CTL brinda cuantificadores para predicar sobre algunos o todos los futuros posibles.

Linear-time Temporal Logic

La *Linear-time Temporal Logic* (LTL) [64, 65] es una lógica temporal que permite expresar predicados con alusión al tiempo futuro. En ella, el tiempo es modelado como una secuencia infinita de estados en la cual, para cada instante de tiempo sólo existe un único sucesor (lineal). Gracias a su poder expresivo, este formalismo es muy utilizado para analizar y verificar sistemas mediante la especificación (y posterior

análisis) de propiedades. Estas propiedades son descriptas mediante fórmulas que caracterizan un conjunto de secuencias, aquellas en las cuales dichas propiedades son verdaderas.

Las fórmulas se construyen a partir de las proposiciones atómicas, los operadores de la Lógica Proposicional ($\vee$, $\wedge$, etc.) y los operadores temporales. Los operadores temporales básicos son **F** *eventually* (en algún momento futuro), **G** *always* (durante todo momento futuro), **X** *next* (en el próximo instante de tiempo) y **U** *until* (sucede hasta).

Definición 2.9. ***Sintaxis de la Linear-time Temporal Logic***

Dado un conjunto de Proposiciones Atómicas $\mathcal{P}$, el conjunto $\mathcal{F}_{\mathcal{LTL}}$ de fórmulas de la Linear-time Temporal Logic está definido como el mínimo conjunto que satisface lo siguiente:

- *Todo elemento del conjunto $\mathcal{P}$ es una fórmula.*
- *Si ϕ y ψ son fórmulas de $\mathcal{F}_{\mathcal{LTL}}$, entonces $\neg\phi$ y $\phi \wedge \psi$ también lo son.*
- *Si ϕ y ψ son fórmulas de $\mathcal{F}_{\mathcal{LTL}}$, entonces $\mathbf{X}\phi$, $\mathbf{F}\phi$, $\mathbf{G}\phi$, $\phi\mathbf{W}\psi$ y $\phi\mathbf{U}\psi$ también lo son.*

Para describir formalmente la semántica, primeramente definimos la noción de estructura temporal subyacente:

Definición 2.10. ***Estructura Temporal Lineal***

Dado un conjunto de Proposiciones Atómicas $\mathcal{P}$, una estructura temporal lineal $\mathcal{T}$ es una triupla (S, t, v) donde:

- *S es un conjunto de estados.*
- *$t : \mathbb{N} \to S$ define una secuencia infinita de estados.*
- *$v : S \to 2^{\mathcal{P}}$ define para cada estado, el conjunto de proposiciones que se satisfacen en él.*

Una notación alternativa para representar una estructura temporal lineal es $w : s_0, s_1, \ldots,$ como una secuencia de estados.

Dada una estructura temporal lineal $w = s_0, s_1, s_2 \ldots$ sobre un conjunto $\mathcal{P}$ de Proposiciones Atómicas, una fórmula $\phi \in F_{LTL}$ es verdadera en w (denotado $w \models \phi$), si se cumplen las siguientes condiciones:

- $w \models p$ *sii* $p \in s_0$
- $w \models \phi \vee \psi$ *sii* $(w \models \phi)$ o $(w \models \psi)$

- $w \models \neg\phi$ *sii* no se cumple $w \models \phi$
- $w \models \phi \wedge \psi$ *sii* $(w \models \phi)$ y $(w \models \psi)$
- $w \models \mathbf{X}\phi$ *sii* $w_1 \models \phi$
- $w \models \phi\mathbf{U}\psi$ *sii* *existe* $i \geq 0 : w_i \models \psi$ y *para todo* $0 \leq j \leq i, w_j \models \phi$

La semántica de los demás operadores temporales **F** (eventually), **G** (always) y **W** (weak until) pueden definirse en términos de los anteriores de la siguiente manera:

- $\mathbf{F}\phi \equiv \text{true}\mathbf{U}\phi$
- $\mathbf{G}\phi \equiv \neg\mathbf{F}\neg\phi$
- $\phi\mathbf{W}\psi \equiv ((\phi\mathbf{U}\psi) \vee \mathbf{G}\phi)$

donde true $\equiv \phi \vee \neg\phi$, es decir, es la constante proposicional verdadera.

En la Figura 2.9 podemos observar gráficamente el significado de los operadores temporales de la *Linear-time Temporal Logic*.

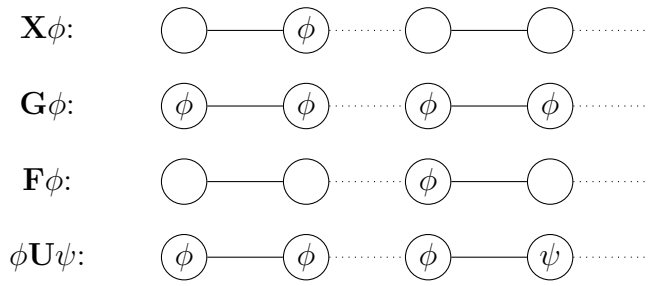


Figura 2.9: Representación gráfica de los operadores temporales de LTL

Nuestro principal interés en este tipo de lógicas radica en su utilización para la especificación y verificación de sistemas, especialmente sistemas reactivos, ya sea basado en estados o basados en eventos. A diferencia de otro tipo de sistemas donde, por ejemplo, una propiedad interesante a comprobar puede ser *terminación*, en los sistemas basados en eventos, las propiedades que podemos caracterizar pueden clasificarse en dos grandes clases: *safety* y *liveness*. Más aún, en [2] se demuestra que cualquier propiedad puede especificarse como una intersección entre éstas.

De manera general, las propiedades de *safety* especifican que el sistema no puede alcanzar un determinado estado (indeseable). Usualmente toman la siguiente forma "Nunca ocurre ϕ", donde ϕ caracteriza el hecho de que el sistema se encuentre en

el estado indeseable. Por otro lado, las propiedades de *liveness* aseguran que cierta propiedad (deseable) siempre será verdad en algún momento en el futuro "En algún momento futuro ocurre ψ", donde ψ caracteriza una cierta propiedad deseable.

Fluent Linear-time Temporal Logic

En la especificación de sistemas basados en eventos, los estados de la estructura temporal subyacente están relacionados con los eventos del sistema. En este tipo de especificaciones la caracterización y análisis de estados o situaciones del sistema, en general se torna compleja. La *Fluent Linear-lime Temporal Logic* (FLTL) es una variante de LTL que facilita la especificación de propiedades de sistemas basados en eventos [37]. Específicamente, FLTL extiende LTL brindando la posibilidad de definir estados abstractos llamados *fluents* (fluentes). Como se define en [69], los fluentes son variables binarias cuyo valor depende de la ocurrencia de eventos. Los fluentes se caracterizan por dos conjuntos de eventos asociados a ellos, cuya ocurrencia modifica su valor.

Formalmente, un *fluente* fl se define como:

$$fl = \langle \{s_1, ..., s_n\}, \{e_1, ..., e_n\} \rangle initially\ B$$

donde $B \in \{true, false\}$, es el valor inicial del mismo en el estado inicial, y $\{s_1, ..., s_n\}$ y $\{e_1, ..., e_n\}$ son conjuntos disjuntos de eventos. El fluente es *verdadero* luego de la ocurrencia de cualquier $s_i \in \{s_1, ..., s_n\}$ (conjunto de activación) y se torna falso luego de que algún evento $e_i \in \{e_1, ..., e_n\}$ tenga lugar. De manera predeterminada, si se omite *initially B* se asume inicialmente *falso*.

Una vez definidos, estos fluentes pueden ser utilizados para la especificación de propiedades, es decir, los fluentes pueden ser utilizados como variables proposicionales en la construcción de fórmulas.

Computational Tree Logic

A diferencia de LTL, en la *Computational Tree Logic* (CTL) [28], el tiempo se comprende como una ramificación de futuros posibles, esto es, a cada instante pueden seguirle diferentes estados sucesores. La estructura temporal subyacente es caracterizada como un árbol.

En CTL además de los operadores de LTL (llamados aquí operadores temporales) se agregan dos cuantificadores (de *caminos*) para especificar futuros posibles: **A** (en todo futuro posible) y **E** (por lo menos en un futuro posible).

Definición 2.11. ***Sintaxis de la Computational Tree Logic***

Dado un conjunto de Proposiciones Atómicas $\mathcal{P}$, el conjunto $\mathcal{F}_{\mathcal{CTL}}$ de fórmulas de la Computational Tree Logic es el mínimo conjunto que satisface lo siguiente:

- *Todo elemento del conjunto $\mathcal{P}$ es una fórmula.*
- *Si ϕ y ψ son fórmulas de $\mathcal{F}_{\mathcal{CTL}}$, entonces $\neg\phi$ y $\phi \vee \psi$ también lo son.*
- *Si ϕ y ψ son fórmulas de $\mathcal{F}_{\mathcal{CTL}}$, entonces $\mathbf{EX}\phi$, $\mathbf{EG}\phi$ y $\mathbf{E}\phi\mathbf{U}\psi$ también lo son.*

Su gramática completa, como BNF:

$$\begin{aligned}\phi ::= \quad & p \mid (\neg\phi) \mid (\phi \wedge \phi) \mid (\phi \vee \phi) \mid (\phi \rightarrow \phi) \mid \mathbf{AX}\phi \mid \mathbf{EX}\phi \mid \\ & \mathbf{AF}\phi \mid \mathbf{EF}\phi \mid \mathbf{AG}\phi \mid \mathbf{EG}\phi \mid \mathbf{A}\phi\mathbf{U}\phi \mid \mathbf{E}\phi\mathbf{U}\phi,\end{aligned}$$

Cabe destacar que los operadores en CTL son pares (cuantificador de camino, operador temporal).

Definición 2.12. ***Estructura Temporal Ramificada***
Dado un conjunto de Proposiciones Atómicas $\mathcal{P}$, una estructura temporal lineal $\mathcal{T}$ es una terna (S, R, v) donde:

- *S es un conjunto de estados.*
- *$R \subseteq S \times S$ es una relación total binaria entre estados.*
- *$v : S \rightarrow 2^{\mathcal{P}}$ define para cada estado, el conjunto de proposiciones que se satisfacen en él.*

Cabe destacar que si consideramos a R como un grafo dirigido, éste es acíclico, cada nodo tiene un único predecesor excepto el nodo *raíz* desde el cual, todos los demás nodos son alcanzables.

Dada una estructura temporal ramificada $\mathcal{T} = (S, R, v)$ sobre un conjunto $\mathcal{P}$ de Proposiciones Atómicas, $s \in S$ y una fórmula $\phi \in F_{LTL}$, el hecho de que ϕ sea verdadera en $\mathcal{T}, s$ (denotado $\mathcal{T}, s \models \phi$) se define recursivamente de la siguiente manera:

- $\mathcal{T}, s \models p$ *sii* $p \in v(s)$
- $\mathcal{T}, s \models \phi \vee \psi$ *sii* $(\mathcal{T}, s \models \phi)$ o $(\mathcal{T}, s \models \psi)$
- $\mathcal{T}, s \models \neg\phi$ *sii* no se cumple $\mathcal{T}, s \models \phi$
- $\mathcal{T}, s \models \mathbf{EX}\phi$ *sii existe* $s_1 : sRs_1$ *y* $\mathcal{T}, s_1 \models \phi$
- $\mathcal{T}, s \models \mathbf{EG}\phi$ *sii existen* $s_1, s_2, \ldots : s = s_1 \wedge s_i R s_{i+1}$ *y para todo* $s_i : \mathcal{T}, s_i \models \phi$

- $\mathcal{T}, s \models \mathbf{E}\phi\mathbf{U}\psi$ *sii existen* $s_1, s_2, \ldots : s = s_1 \wedge s_i R s_{i+1}$ *y existe* $i \geq 1 : s_i \models \psi$ *y para todo* $1 \leq j < i, s_j \models \phi$

La semántica de los demás operadores queda definida de acuerdo a:

- $\mathbf{EF}\phi \equiv \mathbf{E}$ true $\mathbf{U}\phi$
- $\mathbf{AX}\phi \equiv \neg\mathbf{EX}(\neg\phi)$
- $\mathbf{AG}\phi \equiv \neg\mathbf{EF}(\neg\phi) \equiv \neg\mathbf{E}($ true $\mathbf{U}(\neg\phi)]$
- $\mathbf{AF}\phi \equiv \mathbf{A}V\mathbf{U}\phi \equiv \neg\mathbf{EG}(\neg\phi)$
- $\mathbf{A}\phi\mathbf{U}\psi \equiv \neg(\mathbf{E}(\neg\psi\mathbf{U}\neg(\phi \vee \psi)) \vee (\mathbf{EG}\neg\psi))$

En la Figura 2.10 podemos observar una representación gráfica del significado de algunas fórmulas temporales de la *Computational Tree Logic*.

El poder expresivo de LTL y CTL no es comparable, es decir, existen propiedades que pueden expresarse en LTL y no en CTL y viceversa. En [29] se define una lógica temporal que incluye a las anteriores, llamada *Computational Tree Logic** (CTL*). En CTL* existen operadores temporales ($\mathbf{X}, \mathbf{U}, \mathbf{F}, \mathbf{G}$) y cuantificadores de caminos ($\mathbf{A}, \mathbf{E}$), pero a diferencia de CTL, los operadores temporales pueden utilizarse sin tener necesariamente asociado un cuantificador de camino. En [49] puede encontrarse un análisis detallado de las diferentes lógicas mencionadas.

Timed Computational Temporal Logic

Con el propósito de utilizar lógicas temporales para capturar y analizar propiedades de sistemas con restricciones de tiempo real, como aquellos modelados con Autómatas Temporizados (ver Subsección 2.2.3), en [3] se propone una extensión de CTL denominada *Timed Computational Tree Logic*. Esta extensión permite dotar a las fórmulas de restricciones de tiempo real [58].

Dado un conjunto de relojes C y $x, y \in C$, el conjunto de restricciones de tiempo real está definido por:

$$\alpha ::= x \prec c \mid x - y \prec c \mid \neg\alpha \mid (\alpha \wedge \alpha)$$

donde $c \in \mathbb{N}$ y $\prec \in \{<, \leq\}$.

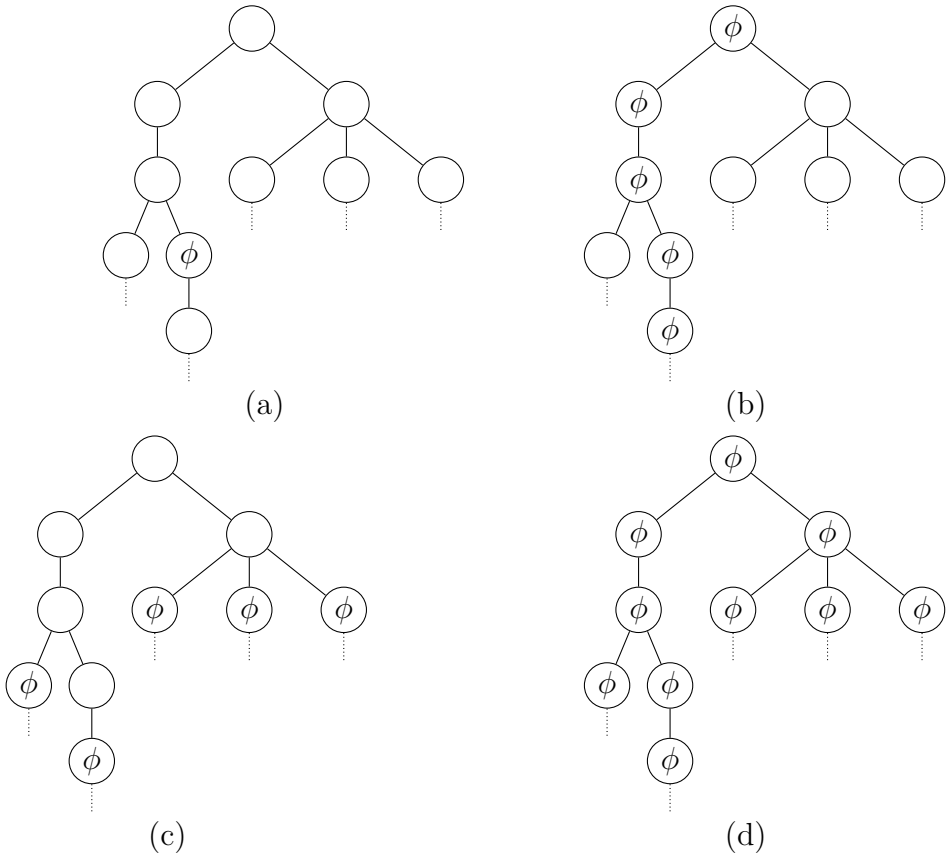


Figura 2.10: Representación de un sistema que satisface:(a)**EF**ϕ, (b)**EG**ϕ, (c)**AF**ϕ, (d)**AG**ϕ

2.4 Técnicas de Análisis Automático

2.4.1 Model Checking

En las dos últimas décadas, el desarrollo de métodos algorítmicos para la verificación de software y hardware ha generado poderosas técnicas de análisis de los mismos. Uno de ellos es *model checking* [24]. *Model checking* brinda un método automatizado para la verificación de *Sistemas de Estados Finitos*, esto es, dado un modelo acotado de un sistema, comprobar de forma exhaustiva y automática si este modelo cumple con una especificación (propiedad) determinada. Las propiedades se especifican mediante fórmulas generalmente en alguna lógica temporal. Cabe destacar en caso de que el modelo no cumpla con la propiedad, esta técnica permite obtener una eje-

cución de dicho modelo que lleva a la violación de la propiedad. Esta información, entre otras aplicaciones, puede utilizarse en composición con otras técnicas, para guiar la mejora/depuración de la especificación de sistema mediante el refinamiento de la misma.

En general, esta técnica trabaja sobre *modelos*, en muchos casos abstractos, de sistemas, por lo cual no garantiza la ausencia de errores en un programa. Sin embargo, es significativamente más efectiva que el tradicional testing para garantizar la calidad de sistemas, dado que realiza una exploración exhaustiva de los modelos de sistemas. Esta técnica ha tenido gran auge en los último tiempos, no sólo en ámbitos académicos sino también en su aplicación a la industria.

Las propiedades utilizadas para verificar sistemas concurrentes pueden clasificarse en grupos. Como ya mencionamos, una clasificación interesante de las mismas las distingue en dos grupos: *safety* y de *liveness*. Las propiedades de *safety* pueden interpretarse como que aseguran que el sistema no puede alcanzar un determinado estado (indeseable). Usualmente toman la siguiente forma "Nunca ocurre ϕ", o su equivalente "Siempre ocurre que $\neg\phi$", donde ϕ caracteriza el hecho de que el sistema se encuentre en el estado indeseable. Un ejemplo de estas propiedades es la ausencia de *deadlock*.

Por su parte, las propiedades de *liveness* predican sobre la recurrencia de ciertas situaciones (deseables) en el futuro. Usualmente toman la siguiente forma: "en algún momento sucederá ϕ", donde ϕ especifica una situación deseable del sistema. Un ejemplo de este tipo de propiedades son aquellas que garantizan que el sistema siempre producirá una respuesta ante un evento dado.

Las propiedades de *liveness*, en general dependen de ciertas características funcionales del comportamiento del sistema. Una de estas características importantes en presencia de no-determinismo es la de *fairness* (justicia). Para ejemplificar el concepto, supongamos que modelamos el comportamiento de `arrojar` una moneda y revisar si sale `cara` o `cruz` como lo muestra la Figura 2.11.

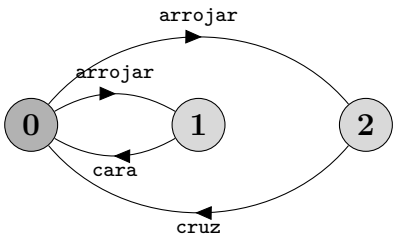


Figura 2.11: Especificación mediante LTS del comportamiento de arrojar una moneda.

En este modelo una propiedad de *liveness* deseable sería "**GF**`cara`" que especifica que en toda ejecución (traza infinita), `cara` "sale" un número infinito de veces (de igual manera podríamos especificar la propiedad para `cruz`). Sin una política de comportamiento *fair* esta propiedad no es válida para el modelo, dado que una ejecución posible podría ignorar indefinidamente el evento "cara". Fairness, como hipótesis en este caso, elimina tales situaciones "injustas", en las cuales un evento es ignorado un número infinito de veces. Formalmente el concepto de *fairness* (técnicamente, *strong fairness*) se define como:

Definición 2.13. ***Fairness***
Si una opción sobre un conjunto de transiciones es alcanzada infinitas veces, entonces cada transición en el conjunto también se ejecutará infinitas veces.

Actualmente existen diversas alternativas para el uso de model checking. Las diferentes alternativas corresponden a distintas formas de representación del sistema (representaciones explícitas del espacio de estados vs. representaciones simbólicas), la lógica utilizada para especificar propiedades (LTL, CTL, etc.), la descripción de los modelos (código en algún lenguaje de programación, lenguaje abstracto o de algo nivel, etc.). A continuación analizamos dos herramientas de *model checking* que se utilizan en este trabajo. Ellas son *Labelled Transition System Analyser* y *Uppaal*. En cada caso, introducimos brevemente el formalismo subyacente, luego una descripción del lenguaje para la especificación de sus modelos y de las propiedades que permiten analizar.

Labelled Transition System Analyser

Primero abordamos la alternativa propuesta en [62] que utiliza como formalismo subyacente los *Labelled Transition Systems* (LTS) (Sistemas de transición de estados etiquetados) y la FLTL como lógica para la descripción de propiedades a verificar. Como lenguaje para la descripción de modelos se utiliza *Finite States Process* (FSP), que es un lenguaje de alto nivel para la especificación de LTS. La herramienta soporte que utilizaremos para la verificación de propiedades es *Labelled Transition System Analyser* (LTSA) [1].

LTSA es una herramienta para la verificación de sistemas concurrentes que permite verificar automáticamente si un sistema satisface un propiedad dada. Estas propiedades deben estar expresadas mediante fórmulas en *Lógica Temporal Lineal* siguiendo la sintaxis que provee FSP para la definición de la mismas. LTSA además, posibilita la definición y uso de *Fluentes* como parte de las fórmulas a verificar.

[1] LTSA puede adquirirse de manera gratuita en: http://www.doc.ic.ac.uk/ltsa/

Los modelos de LTSA se especifican como un conjunto de procesos que se comunican mediante la sincronización de eventos de los mismos. La herramienta realiza un análisis exhaustivo en busca de la violación de la propiedad. Para ello, compone los sistemas de transición de estados correspondientes a los procesos que conforman el modelo. Este sistema contiene todos los estados y transiciones (eventos) que el sistema puede alcanzar. Nuevamente, al componer esta especificación del sistema junto con el sistema de transición que caracteriza la propiedad *negada*, el problema se limita a buscar estados alcanzables desde el estado inicial. Si existieran, dichos caminos conforman los contraejemplos (trazas de ejecución del sistema) que no cumplen con la propiedad especificada.

LTSA soporta de manera transparente el análisis de propiedades de *safety* mediante las expresiones `property` o `assert`. Las primeras permiten especificar una propiedad mediante la definición de un proceso. Un proceso que describe una propiedad de *safety* describe las trazas de la ejecución del modelo que se aceptan como seguras. Las expresiones `assert` permiten especificar propiedades de *safety* de manera declarativa mediante fórmulas FLTL. La sintaxis para los operadores lógicos es la siguiente: `[]` (siempre sucede), `<>` (finalmente sucede), `X` (siguiente), `U` (sucede hasta), `&&` (conjunción), `||` (disyunción), `!` (negación), `->` (implicación). Además, LTSA brinda soporte directo para la especificación de fluentes, permitiendo que los mismos formen parte de las fórmulas que especifican las propiedades a analizar.

En caso de propiedades de *liveness*, LTSA ofrece el análisis de un tipo particular de estas propiedades que predican acerca de la ocurrencia futura de una determinada acción. Este tipo de propiedades se las denomina de *progress* (progreso) y se especifican de manera simple mediante la expresión `progress` en la cual se enumeran las acciones que se desean analizar. El resultado depende de que en toda ejecución del sistema, al menos uno de los eventos especificados ocurra un número infinito de veces.

Como ejemplo simple que permite apreciar la especificación de modelos en LTSA, retomamos el ejemplo de un *buffer* acotado con un proceso que produce y uno que consume. En él describimos por medio de fluentes los estados *lleno* (linea 9) y *vacío* (linea 10) y luego expresamos una propiedad de *safety* (linea 12) que especifica la mutua exclusividad de dichos estados y una propiedad de *liveness* que establece que en todo momento se puede alcanzar un estado futuro en el cual el *buffer* esté lleno.

```
1  const Size = 3;
2
3  BUFFER(N=Size) = STATE[0],
4  STATE[i:0..N] = ( when (i<N) put[i] ->STATE[i+1]
5       | when (i>0) get[i] ->STATE[i-1]).
6
7  PROD = (produce ->PROD).     CONS = (consume ->CONS).
8  || SYSTEM = (PROD || BUFFER(Size) || CONS)
9  /{ put[0..Size-1]/produce, get[1..Size]/consume }.
10
11 fluent Full = < put[Size-1], get[1..Size]>
12 fluent Empty = < get[1], put[0..Size-1]> initially True
13
14 assert CORRECT_BUFFER = [](!(Full && Empty))
15 progress ALWAYS_CAN_BE_FULL = {put[Size-1]}
```

Uppaal

Uppaal es una herramienta[2] para la verificación de sistemas de tiempo real [11, 13]. La herramienta está basada en la teoría de autómatas temporizados para la especificación de sus modelos y provee un subconjunto de TCTL (Timed Computational Tree Logic) como lenguaje para la declaración de propiedades a verificar.

En Uppaal los modelos están compuestos por un conjunto de instancias de esquemas, los cuales se comunican por medio de distintos tipos de canales. La especificación de un modelo consta de tres partes: *Declaraciones Globales*, *Esquemas* (*Automata Templates*) con sus *Declaraciones Locales* y la *Especificación del Sistema*. A continuación describiremos brevemente cada una de estas partes:

- **Declaraciones**: Las declaraciones globales, o locales a un esquema, contienen definiciones de variables, arreglos, registros y tipos (siguiendo el estilo del lenguaje de programación C). Existen cuatro tipos predefinidos: int (enteros), bool (lógicos), clock (relojes), y chan (canales). A su vez, los canales pueden declararse como *urgent* (no consume tiempo) o *broadcast* (de difusión). También pueden definirse constantes anteponiendo el prefijo *const*. Uppaal provee un lenguaje rico para la declaración de funciones que pueden ser invocadas dentro de los esquemas. Estas funciones pueden tener asociadas parámetros, sentencias condicionales, sentencias iterativas como *While* y *For*.

[2]Uppaal puede adquirirse de manera gratuita en: http://www.Uppaal.org/

- **Esquemas**: Los esquemas son definidos en forma de *Autómatas Temporizados Extendidos*. En ellos se pueden realizar acciones como por ejemplo leer y modificar relojes, variables, invocar a funciones definidas, etc. El autómata de un esquema esta formado por *estados* y *aristas* entre ellos. Como herramienta para la modularización de los modelos, los esquemas pueden contener declaraciones locales (privadas) y *parámetros*. Estos últimos pueden especificarse por *valor* o por *referencia* [105].

  Los *estados* pueden tener asociados una *etiqueta* (nombres que los referencian) y un *invariante*. Los invariantes son una condición que siempre debe cumplirse a lo largo de la ejecución de un sistema y se especifican mediante expresiones lógicas. Las expresiones permitidas como invariantes son restringidas: sólo pueden ser una conjunción de condiciones simples sobre relojes, o expresiones lógicas que no contengan relojes. No se permiten condiciones que involucren cotas inferiores sobre relojes.

  Existen tres modificadores para los estados de esquemas:

  - *Inicial* (*initial*): El modificador *initial* determina cuál es el estado donde comienza la ejecución. Cada esquema debe tener exactamente un estado inicial.
  - *Urgente* (*urgent*): Si el estado está declarado como *urgent*, detiene el paso del tiempo mientras un proceso se encuentre en uno de ellos.
  - *Comprometido* (*committed*): Los estados comprometidos al igual que los urgentes detienen el paso del tiempo, y además obligan a que la próxima transición elegida a ser ejecutada deba ser alguna transición habilitada saliente de (con origen en) un estado comprometido (establece una prioridad en la elección de las transiciones).

  Los estados están conectados por medio de **aristas** que establecen la relación de transición entre los mismos. Las aristas pueden tener asociados: *selectores*, *guardas*, *sincronizaciones* y *modificaciones*.

  - *Selectores*: Los selectores retornan, de manera no determinista, un valor en un rango especificado.
  - *Guardas*: Las guardas establecen las condiciones necesarias para que una arista esté habilitada. La especificación de una guarda se realiza por medio de expresiones lógicas con la mismas restricciones que las especificaciones de los invariantes.

  - *Sincronizaciones*: Los procesos pueden sincronizarse a través de canales compartidos. Las acciones permitidas sobre los canales son: escribir (`!`) y leer (`?`). Cada arista sólo puede hacer referencia a un único canal.
  - *Modificaciones*: Cuando las aristas son ejecutadas, las expresiones establecidas en esta sección son evaluadas y su efecto cambia el estado del sistema. El lenguaje para especificar las transformaciones de estados es muy rico ya que, entre otros aspectos, permite invocar a funciones la cuales puede estar "programadas" con un estilo imperativo.

  Como convención en los diagramas de autómatas, para diferenciar estos campos, para el código correspondiente a las guardas utilizamos texto más claro y para el cógido de sincronizaciones *texto en cursiva*.

  Cuando dos procesos son sincronizados, ambas aristas de los procesos son ejecutadas. Las modificaciones de ambos se realizan en orden, primero evaluando las del proceso que escribe y luego las del que lee. Los canales *broadcast* permiten especificar sincronizaciones de uno-a-muchos; esto es, cuando un proceso escribe en un canal de este tipo, todos aquellos procesos con aristas habilitadas en escucha (lectura) de dicho canal, serán invocados.

  La especificación de los esquemas, en particular los autómatas que los definen, se modela de manera gráfica.

- **Especificación del Sistema**: La Especificación del sistema consiste de uno o mas procesos concurrentes (instancias de esquemas), variables y canales de comunicación. Las variables, canales y funciones definidas aquí no son visibles para los esquemas.

Como ejemplo de especificación en Uppaal, a continuación retomamos el ejemplo presentado en la Subsección 2.2.3, y lo extendemos para mostrar algunas de las características sobresalientes de lenguaje. El ejemplo consiste en modelar un usuario con dos ratones y contabilizar la cantidad de *clics* y *doble-clics* en cada uno de ellos. El *mouse* funciona de la siguiente manera: una vez que el usuario presiona su botón, si vuelve a presionarlo en menos de 3 unidades de tiempo, se considera *doble-clic*; sino, simplemente se asume un simple *clic*.

En la Figura 2.12 se muestra un modelo en Uppaal que describe el ejemplo planteado. Inicialmente se muestran de manera gráfica (como se modela en la herramienta) los esquemas que describen el comportamiento del *mouse* y del usuario. El primero contiene un estado *idle* a partir del cual, si recibe la notificación mediante el canal de comunicación *press*, pasa al estado siguiente iniciando el conteo. A partir

de allí, si vuelve a recibir una nueva notificación en menos del tiempo máximo estipulado (*DClickDelay*) para considerar *doble clic*, entonces vuelve al estado inicial contabilizando, mediante la función *DClickCount*, el suceso en el correspondiente *mouse*. En caso de no recibir dicha notificación, se contabiliza de manera similar el suceso simple clic. La identificación de los mismos se realiza mediante el parámetro *mouse_id*.

Siguiendo con la explicación del modelo, en el mismo se muestra el esquema correspondiente a la descripción del comportamiento del usuario. Dicho esquema presenta un único estado con una posible transición que modela la acción del usuario de presionar. La recurrencia de esta transición se encuentra condicionada a un cierto intervalo de tiempo *HumanDelay*. Cuando la transición tiene lugar, se notifica la acción por medio del canal *press*, se inicializa el reloj para capturar el intervalo mencionado y se contabiliza, mediante *PressCount*, la cantidad de veces que el usuario ha presionado la tecla de algún mouse.

Finalmente, se muestran el código correspondiente a las diferentes partes de un modelo Uppaal: declaraciones, esquemas y sistema.

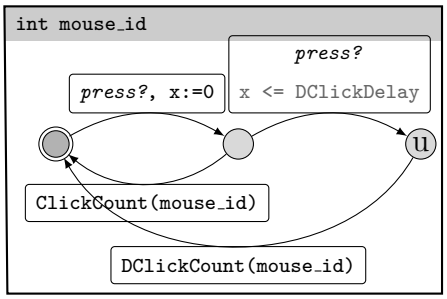
En las *declaraciones* (líneas 1-16) inicialmente se definen los elementos (tipos, constantes, variables, funciones, etc.) utilizados por el modelo. En particular, para el modelo del ejemplo, se definen una constante que indica la cantidad de *mouses*, el canal de comunicación que comunica la acción de presionar del usuario, dos arreglos (uno por cada mouse), una variable para contabilizar los eventos a analizar y un reloj para analizar el paso del tiempo a nivel de sistema. Desde la línea 9 a 19, y con el propósito de mostrar la flexibilidad que brinda Uppaal, se definen tres funciones que ofician de interfaz para la actualización de los datos.

En las declaraciones locales a cada esquema, líneas 17,18 para `mouse` y 19,20 para `user`, simplemente se declaran relojes locales utilizados para condicionar la ocurrencia de eventos.

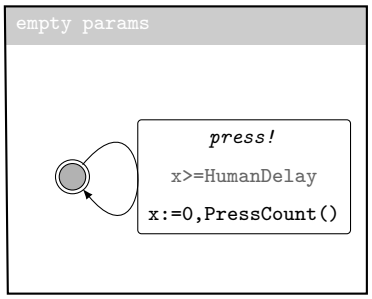
Finalmente, en la declaración del sistema (líneas 21-27), se definen las constantes correspondientes a la demora (unidades de tiempo) del usuario para presionar la tecla y la del *mouse* para considerar un doble clic. Luego se declaran las instancias de los esquemas: dos instancias `M0,M1`, para el esquema `Mouse` representando los dos ratones del sistema y una instancia para el esquema `User`. El *sistema* se especifica mediante el comportamiento de estas tres instancias.

Uppaal brinda como lenguaje de especificación de propiedades, la lógica CTL con ciertas restricciones. Las restricciones exigen que las fórmulas sólo puedan contener un cuantificador de caminos. Así las propiedades que podemos especificar con estas restricciones tiene la forma:

- `E<>`ϕ: se evalúa como verdadera si y sólo si existe un camino donde en algún estado se satisface ϕ.



Esquema: *Mouse*



Esquema: *Usuario*

```
--------------Declarations--------------
const int MouseNumber = 2;
chan press; //user-mouse comunication channel
int Clicks[MouseNumber],DClicks[MouseNumber];//clicks counting
int presses; //press counting
clock time; //global time

//function for one click updating count
void ClickCount(int m_id ){
  cantClick[m_id]++;}
//function for double click updating count
void DClickCount(int m_id){
  cantDClick[m_id]++;}
//function for press updating count
void PressCount(){
  press++;}
-------Mouse Template Declarations-------
clock x; //local clock
--------User Template Declarations-------
clock x; //local clock
-----------System Declarations-----------
const int HumanDelay = 1;
const int DClickDelay = 3;
M0 = Mouse(0);
M1 = Mouse(1);
U = User();
system M0,M1,U;
```

Figura 2.12: Modelo en Uppaal que representa un usuario con dos ratones

- `A[]`ϕ: es verdadera si y sólo si todo estado alcanzable (en cualquier camino) satisface ϕ.
- `E[]`ϕ: es verdadera si y sólo si existe un camino cuyos estados (todos) satisfacen ϕ.
- `A<>`ϕ: es verdadera si y sólo si para cualquier camino es posible alcanza un estado que satisface ϕ.

En las fórmulas anteriores, ϕ es una expresión lógica bien formada en la cual podemos hacer referencia a variables o estados de los procesos (instancias de esquemas). En la Figura 2.10 se puede encontrar una descripción gráfica de estas expresiones. Cabe aclarar que la sintaxis que utiliza Uppaal para los operadores temporales *siempre sucede* y *finalmente ocurrirá* siguen el estilo de FSP: `[]` y `<>`, respectivamente.

Como ejemplo, podemos analizar las siguientes propiedades sobre el modelo de la Figura 2.12:

- La relación entre la cantidad de veces que el usuario ha apretado algún botón debe ser consistente con la cantidad de eventos clic y doble-clic reportados.

```
A[] (M0.idle && M1.idle) imply (Presses ==
 (Clicks[0] + Clicks[1] + (DClicks[0]+DClicks[1]) *2))
```

- Es posible hacer doble-clic en algún *mouse* en menos de tres unidades de tiempo.

```
E<> (time < 3) && ((DClicks[0] + DClicks[1])>=1)
```

Para una referencia detallada sobre Uppaal y los formalismos relacionados a esta herramienta, sugerimos al lector recurrir a [11].

Capítulo 3

Lenguajes para el modelado de Procesos de Negocio y Workflows

3.1 Introducción

Este capítulo presenta el marco teórico general del presente trabajo. El mismo se focaliza en la descripción, especificación y análisis de Procesos de Negocios. La utilización de descripciones de procesos de negocios ha tomado notable importancia debido a su creciente aplicación en la industria. Como consecuencia, han surgido tanto a nivel académico como comercial, diferentes lenguajes y herramientas de software para especificar y analizar procesos de negocios.

Existe una muy amplia variedad de lenguajes de descripción de procesos de negocios, con diferentes objetivos y enfoques. En este trabajo, acotamos el marco teórico a aquellas descripciones que abordan aspectos de *comportamiento* de los procesos de negocios. Estas descripciones, en general, apuntan a la especificación de diferentes elementos que conforman procesos, y la interacción entre los mismos.

En este capítulo tomamos como punto de partida la noción de Proceso de Negocio en general. A partir de esta noción, centramos la atención en un caso particular de procesos de negocios, denominados *Workflows* (Flujos de Trabajo). Esbozaremos una síntesis de los elementos necesarios para poder describirlos. Revisaremos cuatro lenguajes particulares para la descripción de workflows, cuyos detalles se muestran en el Apéndice A. Esta revisión nos permitirá identificar elementos comunes de las descripciones de procesos de negocios, y dará pie para motivar la importancia del análisis de estas descripciones, en nuestro caso mediante *métodos formales*. Finalizamos el capítulo con una breve motivación que introduce los capítulos siguientes.

3.2 Procesos de Negocios y Flujos de Trabajo

Podemos situar el surgimiento del concepto de *proceso de negocio* a comienzos de la década del '90, impulsado por la necesidad de las empresas de optimizar su producción. Con el paso del tiempo diferentes definiciones de procesos de negocios, que hacen énfasis en distintos aspectos de los mismos, fueron caracterizando dicho concepto. Recientemente, en [106], y como conclusión de un análisis de caracterizaciones previas, se define un proceso de negocio como se describe a continuación.

Definición 3.1. ***Proceso de Negocio (business process)***
Un proceso de negocio *consiste en un conjunto de actividades, con un objetivo común, que se realizan de manera coordinada en un entorno organizado. Cada proceso de negocio es ejecutado por una organización, pero puede interactuar con otros procesos que lleven a cabo otras organizaciones.*

De esta definición surgen dos elementos relevantes. Por un lado, la definición hace referencia a una organización y sus capacidades, por lo cual resulta importante contar con descripciones organizacionales, que capturen justamente estas capacidades, y otros aspectos relevantes tales como actores intervinientes en las actividades, roles, etc. Por otro lado, la definición menciona la *coordinación* de las actividades, lo que demanda un lenguaje para describir, operacionalmente, cómo se organizan o componen las actividades que conforman el proceso.

Las descripciones organizacionales brindan una visión estratégica sobre el entorno en el cual se desarrolla el proceso, identificando los actores intervinientes, sus respectivas responsabilidades y los recursos necesarios para el desarrollo del proceso. La vinculación entre diferentes procesos (que pueden llevar adelante otras organizaciones) se denomina comúnmente *coreografía*, y suele describirse de manera informal, utilizando algún tipo de notación gráfica.

Además, y como suele ocurrir en diferentes ambientes de especificación, las descripciones pueden estar dadas en variados niveles de abstracción. A medida que el nivel de abstracción en la descripción de un proceso decrece, resulta necesario brindar una descripción operacional que permita capturar de forma precisa las actividades que componen el proceso, pero sin llegar a un nivel de detalle propio de una implementación. Esta descripción tiene el propósito de ordenar y coordinar las tareas controladas por el proceso, cuya realización genera el producto u objetivo esperado. A este tipo de descripciones se las denomina *orquestaciones*, y su realización se basa en el uso de modelos de procesos de negocios. Cuando las actividades (y las relaciones entre las mismas) que componen un modelo de proceso de negocio se pueden describir de manera precisa, se denomina a la descripción un *flujo de trabajo* (*workflow*) [46, 91].

Definición 3.2. ***Flujos de Trabajo (workflows)***
Un flujo de trabajo define la asistencia o automatización computarizada, total o parcial, de un proceso de negocio.

En la última década, impulsados por variadas innovaciones tecnológicas, han surgido una gran cantidad de técnicas, lenguajes y herramientas para asistir en la descripción y el análisis de procesos de negocios. Estos lenguajes y sus respectivas herramientas han sido fuertemente influenciados por su aplicación en la industria, y la necesidad de automatizar el análisis de procesos de negocios, entre otras razones, ha fomentado la investigación en el área. Este trabajo se focaliza, en gran parte, en la especificación de propiedades de procesos de negocios y su posterior análisis automático.

3.3 Descripción y especificación de workflows

Como mencionamos anteriormente, la descripción de un proceso de negocio es en general una descripción en la cual pueden tenerse en cuenta varias perspectivas. Nos concentraremos en las descripciones de *orquestación*, particularmente en la especificación de workflows.

Como lo muestra el bosquejo de la Figura 3.1, las descripciones gráficas son muy adecuadas para la representación de workflows, ya que facilitan la comprensión del sentido del flujo de control, ayudan a visualizar las dependencias entre actividades, etc., especialmente cuando el ámbito de utilización de tales representaciones está vinculado a la industria. Por esta razón, los lenguajes para la especificación de workflows se basan, en gran medida, en descripciones gráficas, aunque existen lenguajes para la especificación de workflows cuyas descripciones no son gráficas, como BPEL[1] [31].

3.3.1 Elementos de especificación

A continuación exploramos los principales componentes que integran la descripción de un *workflow*. En cada situación analizaremos su implementación en diferentes lenguajes para la especificación de *workflows*. A diferencia de análisis como los expuestos en [18, 57, 91], entre otros, en este caso no evaluamos los lenguajes en sí, sino que intentamos identificar elementos comunes a ellos, que consideraremos fundamentales en el modelado de workflows. Si bien los lenguajes elegidos para la evaluación

[1] Business Process Execution Language es un lenguaje para la especificación de workflows, particularmente enfocado hacia la composición de servicios, cuya sintaxis se basa en XML.

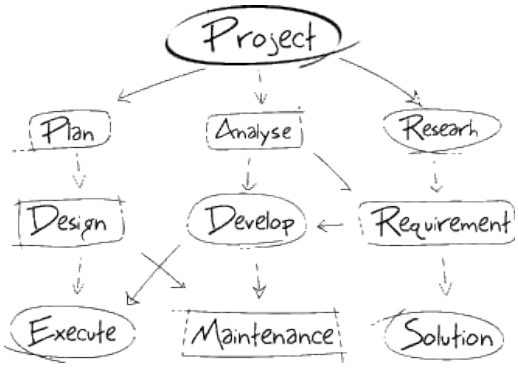


Figura 3.1: Bosquejo de un proceso

(cuya elección estuvo basada en su madurez y nivel de uso en la práctica) tienen diferentes particularidades, veremos que todos ellos coinciden en un núcleo de conceptos comunes, los que consideramos fundamentales en las descripciones de workflows. La descripción para cada uno de los lenguajes, como mencionamos anteriormente, se detalla en el Apéndice A.

Los lenguajes evaluados son los siguientes:

- *The Business Process Modeling Notation (BPMN).* BPMN [108] surgió en 2004 como un estándar para la descripción de procesos de negocios. Uno de sus objetivos de su definición fue la unificación de diferentes lenguajes y herramientas existentes al momento. Debido a su aprobación por la comunidad académica y su utilización en la industria, en 2008 el *Object Management Group* (OMG)[2] publicó la versión 1.1 [39] y en 2011 la versión 2.0.

- *Yet Another Workflow Language (YAWL).* YAWL [91, 96] esá basado en Redes de Petri (PN) [38], y uno de sus principales propósitos es el de brindar análisis formal de especificaciones. Entre una de sus principales características podemos destacar que sus modelos son producto de la composición de patrones de flujos de trabajo (*workflow patterns*). Estos patrones, construcciones genéricas y recurrentes en las descripciones de procesos de negocios, fueron identificados por la *Workflow Pattern Initiative*[3] con el objetivo de brindar bases conceptuales para la especificación de procesos de negocios [98].

[2]OMG Es un consorcio formado por diversas compañías y organizaciones sin fines de lucro dedicado al cuidado y el establecimiento de diversos estándares de tecnologías orientadas a objetos.

[3]Puede encontrarse más información relativa a la Workflow Pattern Initiative en http://www.workflowpatterns.com.

- *Event-Driven Process Chain (EPC).* EPC [52] es un lenguaje gráfico desarrollado a principios de los años '90 como parte del *framework Architecture of Integrated Information System (ARIS)* para la descripción de procesos de negocios. ARIS ofrece diferentes enfoques para el análisis y desarrollo de un sistema de información. Cada uno de estos enfoques, Organizacional, Funcional, Datos y Control, provee diferentes diagramas específicos para describir, por ejemplo, diagramas de Árbol de Funciones para el enfoque Funcional, diagrama de Entidades y Relaciones para el enfoque de Datos, un Organigrama para describir las responsabilidades en el enfoque Organizacional, etc. Dentro del enfoque de Control y con un rol integrador entre los distintos enfoques, se utiliza el diagrama EPC que describe la ejecución (flujo de control) del proceso.

- *Unified Modeling Language (UML).* UML es una familia de notaciones gráficas que fue desarrollada por Booch, Rumbaugh y Jacobson [17] para la organización, especificación, desarrollo y documentación de diferentes aspectos de un sistema. Gracias a su difusión y aceptación fue adoptado como un *estándar* para el modelado orientado a objetos por la *Objet Management Group* (OMG). Pese a sus escasos fundamentos formales, la versatilidad del lenguaje puede apreciarse a través de sus numerosos diagramas que permiten abordar diferentes tipos y etapas en el desarrollo de sistemas. Estos diagramas pueden agruparse como *estructurales* o de *comportamiento*. Los *diagramas de actividades*, en particular, describen el flujo de control de las diferentes actividades que conforman un sistema.

Actividades

Son el concepto básico y principal componente de un Proceso de Negocio. Las actividades representas los elementos del sistema que, mediante su ejecución, contribuyen a la realización del mismo. Dependiendo de la riqueza del lenguaje, pueden recibir, modificar y retornar información.

Como se diagrama en la Figura 3.2, las actividades pueden estar organizadas de manera jerárquica a través del concepto de *composición vertical*, esto es, una actividad puede estar especificada por un conjunto de actividades que la componen. La *composición horizontal* determina la interrelación entre las actividades en cuya cooperación contribuyen a la realización del objetivo parcial o final del proceso de negocio. Esta relación entre actividades es especificada por medio del *flujo de control* entre las mismas.

Bajo distintas denominaciones, las actividades forman parte de los diagramas de los lenguajes analizados. En YAWL se denomina *tareas* y como característica par-

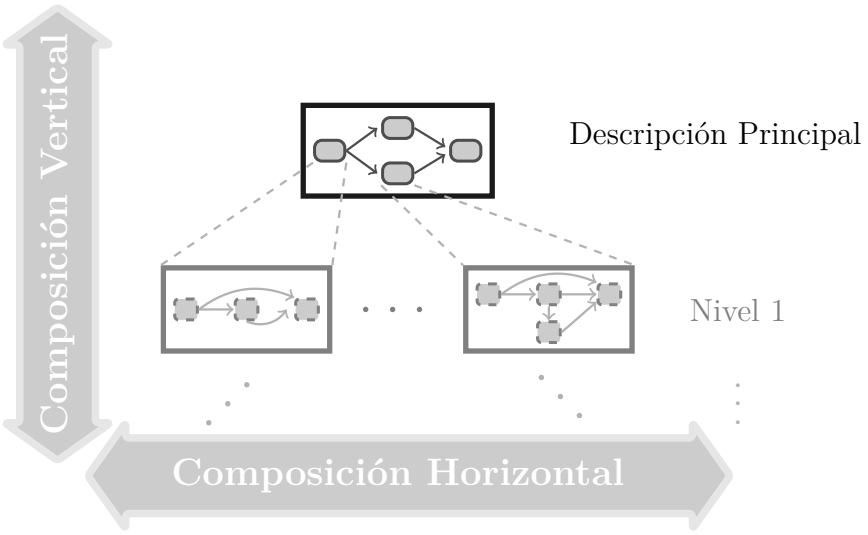


Figura 3.2: Composición Vertical y Horizontal de Actividades

ticular, pueden especificarse *instancias* de las mismas. El concepto de composición vertical es común a todos los lenguajes analizados, y su implementación en los mismos es recursiva, es decir, una actividad puede ser detallada por otra instancia de modelo del lenguaje.

Estados, eventos y condiciones

En los modelos descriptivos de Procesos de Negocios es necesario explicitar estados que capturan situaciones relevantes para el modelo. Como ejemplo de estas situaciones podemos mencionar eventos que intervienen en la ejecución del modelo, estados condicionales para la continuación del flujo, estados parciales dentro del proceso, entre otros. Particularmente los eventos o estados comunes a los diferentes lenguajes, son los que capturan el *comienzo* y *fin* de procesos.

La caracterización de estos componentes como parte de los lenguajes depende del enfoque del mismo. En caso de BPMN, en el cual prima la riqueza descriptiva por sobre la formalidad, estos componentes, adoptados como *eventos*, están agrupados en categorías, las cuales proveen diferentes tipos (simple, mensaje, señal, condicional, etc.), cada uno con su correspondiente interpretación. En YAWL, donde los fundamentos formales fueron predominantes en el diseño del lenguaje, se provee el concepto de *condición* para reflejar estas situaciones. La semántica formal de las condiciones de un modelo YAWL son capturadas de manera transparente por los

sitios de la Red de Petri que lo especifica. En otros lenguajes, como por ejemplo en los Diagramas de Actividades de UML, este tipo de situaciones está asociado directamente al control del flujo de ejecución.

Control del flujo

El control de flujo es uno de los aspectos más relevantes de los lenguajes para el modelado de procesos de negocios, en relación a nuestro interés sobre el análisis de los mismos. El control de flujo representa la interrelación que existe entre las diferentes entidades que componen un modelo para lograr el objetivo del mismo. El control de flujo está presente en todos los lenguajes para el modelado de procesos de negocios, generalmente a través de operadores de control. Podemos clasificar los operadores, independientemente de los lenguajes, en dos grandes categorías: los de *distribución* y los de *sincronización*. En [98], y como parte de la *Workflow Patterns Initiative*, se han identificado una serie de patrones para el modelado de procesos de negocios, entre ellos los correspondientes a la perspectiva de flujo de control. Más allá de esta clasificación detallada, en general los operadores presentes en los lenguajes de modelado de procesos de negocios son: *Todos*, *Al menos uno*, y *Exactamente uno*. Estos operadores se aplican como distribuidores o sincronizadores de flujo de control. Su descripción, para cada una de estas categorías, es la siguiente:

- Como distribución del control:
  - *Todos*: Implementa el concepto de paralelismo, esto es, distribuye el control a todos los objetos asociados a él.
  - *Exactamente uno*: Se utiliza como punto de decisión exclusiva, pasando el control a uno de los objetos asociados.
  - *Al menos uno*: Distribuye el control de manera paralela a uno o mas objetos asociados.
- Para sincronización del control:
  - *Todos*: Sincroniza, esperando por la finalización de todos los objetos asociados a él.
  - *Exactamente uno*: Implementa la noción de filtro, esto es, propaga el flujo de control de uno y sólo uno de los objetos asociados.
  - *Al menos uno*: La semántica de este operador es un tema de discusión. En particular existen diferentes trabajos que lo abordan, por ejemplo [54, 97, 114]. Dependiendo de la formalidad de sus fundamentos, la

discusión sobre su semántica diverge con respecto a cuándo el operador debe trasladar el flujo al componente siguiente en el modelo.

Además de éstos, cada lenguaje tiene otros operadores particulares, que en general son una notación abreviada de la composición de operadores básicos mencionados. Dependiendo de su enfoque, ciertos lenguajes brindan la posibilidad de abordar situaciones especiales de control de flujo. Por ejemplo, YAWL permite capturar situaciones de interferencia dentro de la ejecución de un modelo por medio de *regiones de cancelación*. A través de éstas, se especifica cómo la ejecución de una actividad impide o aborta la ejecución de un conjunto de actividades asociadas a su región de cancelación.

Dada la naturaleza gráfica de los lenguajes, la relación de flujo o composición secuencial (es decir, la conexión entre los diferentes elementos que componen un modelo) se especifica por medio de líneas que denotan dicha conexión. Estas conexiones deben respetar reglas de formación que impone cada lenguaje para la especificación de sus modelos. Por ejemplo, en caso de EPC, las actividades y eventos deben estar intercalados. Además, en este lenguaje los eventos tienen como máximo un flujo entrante y a lo sumo uno saliente, pero al menos uno de ellos debe estar presente. Si bien los operadores lógicos pueden ser combinados con actividades y funciones, en EPC el único operador de bifurcación que puede seguir a un evento es *todos*.

Elementos Complementarios

Además de los objetos pertinentes al modelado de workflows mencionados, en general los lenguajes proveen elementos complementarios para modelar objetos que conciernen a los procesos de negocios. Si bien estos objetos varían en su descripción y detalle según la perspectiva del lenguaje, podemos clasificarlos en dos grupos:

- Representación estructural y de recursos: Son aquellos elementos que permiten describir en el modelo aspectos estructurales del proceso de negocio. En general, los lenguajes que brindan soporte para la representación organizacional permiten describir estructuras o recursos, y asignar a los mismos las actividades que tienen a su cargo, o eventos que pueden suceder dentro de su ámbito. Algunos ejemplos de éstos son las *unidades* en EPC y los *canales* en BPMN. YAWL provee una descripción detallada de los recursos y asignación de actividades a los mismos (*Resource Management*), siguiendo conceptos introducidos en [98] donde se describen *patrones de recursos*.

- Representación de Información: En todo proceso de negocio existe información que es manipulada por el mismo. Esta información en algunos casos puede estar asociada a ciertas actividades o eventos. Por ejemplo, una actividad pude

recibir, procesar y retornar información. También, la ocurrencia de un evento puede depender de determinados valores de un objeto del ambiente. En general los lenguajes brindan soporte para representar la información existente. En el caso de YAWL, se provee un formato muy rico (XML [4]) para describir la información, y la misma puede estar asociada directamente con las actividades. Otros lenguajes que forman parte de un proceso de especificación integral, como por ejemplo EPC, brindan como parte de los mismos diagramas específicos para representar la información (diagramas de Entidades y Relaciones, en el caso de EPC). Luego, dentro del modelo de *workflow*, los objetos que modelan información son representados gráficamente para asociarlos a las actividades que los manipulan. De manera similar que para los recursos, en [98] se describen y analizan diferentes patrones (*Data patterns*) sobre la manipulación de información como parte de un proceso de negocio.

3.4 Análisis de Workflows

A medida que las descripciones crecen, la necesidad de asistir la especificación y seguimiento de workflow se torna indispensable. Es allí donde aparecen los sistemas para la gestión de workflows.

Definición 3.3. ***Sistema de Gestión de Workflows (Workflows Management Systems)***

Es un sistema que define, crea y gestiona la ejecución de workflows a través del uso de software. Este sistema puede tener asociado uno o más motores de workflows capaces de interpretar la definición del proceso, interactuar con los participantes y, de ser necesario, invocar el uso de herramientas o aplicaciones. [61]

Actualmente existe una gran variedad de herramientas para la gestión de workflows utilizados frecuentemente por la industria. Algunos ejemplos son: Bizagi, BonitaSoft, jBPM, CuteFlow, Biztalk, entre otros. Entre sus características más importantes, estas herramientas no sólo asisten en la especificación de workflows basados en uno o mas lenguajes, sino que permiten la integración (composición) de los mismos con otras aplicaciones o servicios (*coreografía*).

Uno de los aspectos importantes de un lenguaje es su semántica, esto es, de qué manera son son interpretados sus modelos. El grado de formalidad de la semántica de un lenguaje tiene un impacto directo en el grado de automatización del análisis

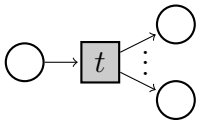
[4] de las siglas en inglés "eXtensible Markup Language", es un lenguaje de marcas desarrollado por el *World Wide Web Consortium* (W3C) que permite definir gramáticas para estructurar documentos grandes. El mismo puede ser utilizados como soporte a bases de datos, lo cual facilita, entre otras aplicaciones, el intercambio de información entre procesos.

de sus modelos. Así, los lenguajes con semántica semi-formal o informal, como sucede en general con los lenguajes para especificaciones de workflows, suelen tener un poder expresivo muy rico, pero como contraparte de esto el análisis de sus modelos es en general muy limitado. No obstante se pueden encontrar en el área de los métodos formales numerosos trabajos que aportan técnicas y herramientas con el fin de formalizar dichos lenguajes.

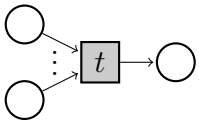
En general, estas técnicas adoptan algún lenguaje para modelar procesos de negocios o parte de ellos, a través de alguna notación basada en diagramas, y les brindan una semántica formal con soporte en algún formalismo. Luego, haciendo uso de las herramientas automáticas o semi-automáticas para el análisis de dichos formalismos, permiten analizar propiedades sobre los modelos originales. En [73] podemos encontrar un exhaustivo análisis de éstos trabajos clasificados según los formalismos en los cuales se basan.

Más allá de éstas técnicas basadas en lenguajes informales o semi-formales existentes, en los últimos años y con la intención de fundar formalmente lenguajes para la especificación de procesos de negocios, han surgido diferentes propuestas que utilizan formalismos para la representación de procesos de negocios. Debido a su semántica simple, su potencialidad para describir sistemas concurrentes y su notación gráfica, las Redes de Petri (ver Subsección 2.2.2) es un formalismo muy utilizado como fundamento formal para la especificación de Procesos de Negocios. En particular y como complemento de su utilización como base teórica en la especificación de los patrones de workflows analizados en [98], una extensión de las Redes de Petri brinda la semántica formal del control de flujo del lenguaje YAWL.

Por ejemplo, para el capturar la semántica de algunos operadores de flujo de control se especifican las siguientes Redes de Petri:



Distribución a *todos*



Sincronización desde *todos*

En este ejemplo puede observarse cómo el concepto de *token* oficia de instancia de control. Así, aprovechando las nociones de concurrencia y no determinismo que ofrecen las Redes de Petri, pueden capturarse todos los patrones de flujo de control. En la Sección A.2 puede encontrarse de manera detallada la semántica formal de YAWL.

Otro formalismo utilizado como lenguaje para especificar workflows son las Álgebras de Procesos. En [78,79], por ejemplo, se utiliza el π-*Cálculo* [71,87] para capturar

las semántica de diferentes patrones de workflows con un aporte sobre la comunicación y dinamismo entre las actividades que conforman un proceso de negocio. Siguiendo los ejemplos anteriores, los procesos que capturan los operadores mencionados se muestran a continuación:

$A = \pi_a.(\bar{b}.\mathbf{0} \mid \bar{c}.\mathbf{0})$	$B = \pi_b.\bar{d_1}.\mathbf{0}$
$B = b.\pi_b.B'$	$C = \pi_c.\bar{d_2}.\mathbf{0}$
$C = c.\pi_c.C'$	$D = d_1.d_2.\pi_d.D'$
Distribución a *todos*	Sincronización desde *todos*

Los enfoques basados en álgebras de proceso, particularmente los basados en π-cálculo, utilizan fuertemente canales de comunicación, a través de sus operaciones de escritura ($\bar{c}$) y lectura(c), para capturar la noción de flujo de control. Además, estos enfoques permiten abarcar formalmente y de manera transparente el *dinamismo* en cuanto a los componentes que conforman un modelo, ya que brindan la noción de instancia de procesos, con sus operaciones de creación (!) y finalización($\mathbf{0}$).

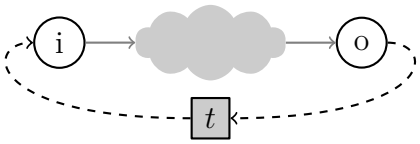
El surgimiento de estos lenguajes, cuyas semánticas están basadas en alguna teoría formal, tiene como principal objetivo el poder utilizar la información de los modelos para realizar diferentes tipos de análisis formales sobre ellos. En este tipo de análisis una de las ventajas a considerar es la automatización de los mismos. Dada su complejidad, que en general requiere de conocimientos específicos sobre el formalismo en cuestión, la utilización de estos lenguajes con semántica formal depende fuertemente de la interfaz (*front-end*) que brinden al usuario. Por ejemplo, YAWL provee un ambiente gráfico para la definición de sus modelos, que abstrae al usuario de la necesidad de conocer sobre Redes de Petri.

Los diferentes enfoques, asociados a sus formalismos subyacentes, condicionan en parte el tipo de propiedades que pueden analizarse. Por ejemplo, las propiedades abordadas por las formalizaciones con Redes de Petri están fuertemente relacionadas a propiedades estructurales de sus modelos. En caso de YAWL, cuya semántica está definida en términos de *workflow nets* (extensión de Redes de Petri), una de la propiedades destacadas que se puede analizar es *soundness* (sanidad, o corrección) [97], con diferentes tipos de restricciones.

Definición 3.4. ***(Workflow Net)*** *Una Red de Petri $R = (P, T, F)$ es una Red de Flujo de Trabajo (WF-Net) si y sólo si*

- *Contiene dos sitios especiales: i (comienzo), el cual no tiene arcos de entrada; y o (salida); sin arcos de arcos de salida.*

- *Si agregamos una transición t a R conectando o con i (ilustrado en el siguiente diagrama), la Red resultante es* fuertemente conexa.



Definición 3.5. *Una Workflow Net W es* correcta *(sound) si y sólo si:*

- *Si un estado M es alcanzable desde i, entonces desde él puedo alcanzar o.*
- *El único estado alcanzable donde o tiene tokens es exactamente aquel donde o tiene uno y los demás sitios están vacíos.*
- *W no tiene transiciones bloqueadas (para toda transición de la red, existe un estado en la cual está habilitada).*

Este tipo de propiedades son abordadas también por otros formalismos, como es le caso de [77], donde luego de capturar la semántica de procesos de negocios mediante π-cálculo, se analizan propiedades estructurales como *deadlock* [8] y *soundness*. Esta última se analiza vinculándola al concepto de bisimulación que ofrece dicho cálculo para el análisis de sus modelos.

Este tipo de propiedades son *generales*, en el sentido de que las mismas son deseables en todo proceso de negocios o workflow. Constituyen, de alguna manera, chequeos de sanidad de un modelo dado. Además de estas importantes propiedades, es en muchos casos necesario estudiar otras más específicas de cada modelo, que si bien predican acerca de actividades, eventos u otros elementos puntuales de un modelo, tienen sentido o interés debido al dominio que se intenta modelar o capturar. Algunos ejemplos de este tipo de propiedades son las siguientes:

- ¿Puede una actividad depender de la ejecución de otra ?
- ¿Pueden dos actividades estar activas al mismo tiempo ?
- La ejecución de una actividad, ¿puede excluir la ocurrencia de otra?
- ¿Puede tener lugar una actividad entre la ocurrencia de dos eventos particulares ?

En los últimos años, diferentes trabajos (e.g., [72,113] entre otros) han abordado desde variadas perspectivas este enfoque de análisis de workflows. En ellos se revela

la necesidad de responder preguntas como las mencionadas, esto es, poder realizar análisis de propiedades específicas del modelo de interés, haciendo referencia a elementos del modelo.

Las especificaciones declarativas [32], aquellas que expresan relaciones o restricciones sobre los elementos del modelo sin recurrir a notaciones operacionales, son un común denominador para abordar este tipo de análisis. Los enfoques declarativos utilizan diferentes formalismos que permiten predicar acerca de propiedades de modelos, generalmente en alguna lógica modal. Por ejemplo, en [89] se provee un lenguaje de restricciones basado en lógica temporal lineal, que permite predicar acerca de trazas (ejecuciones) finitas, y cuyo análisis permite verificar tales propiedades respecto de los modelos analizados.

El presente trabajo también constituye una contribución en el ámbito de las especificaciones declarativas de propiedades de workflows. En el Capítulo 4 se detalla nuestra propuesta de utilización de una lógica temporal extendida para el análisis de modelos de workflows. Particularmente se adopta un lenguaje para la descripción de workflows y a partir de su traducción automática a sistemas de transición de estados, se provee un asistente, que permite utilizar la lógica temporal con *fluentes* para especificar y verificar propiedades sobre los modelos. Este trabajo permite el análisis de modelos procedurales de workflows mediante propiedades de estilo declarativo, explotando la ventaja de los *fluentes* para describir escenarios o situaciones de dichos modelos.

Usualmente, los enfoques para el modelado de procesos de negocios se concentran en la descripción de *procesos*. En este trabajo también abordaremos la posibilidad de contar en modelos de procesos de negocios con la posibilidad de especificar *productos*, como elementos explícitos de los modelos. La norma internacional ISO-9001 define un proceso como "una actividad que utiliza recursos, y que se gestiona con el fin de permitir que los elementos de entrada se transformen en resultados" [43]. Creemos que en muchos casos, por ejemplo, en el caso de procesos industriales o modelos en los cuales el efecto de los procesos en la información que manipulan es relevante, la necesidad de contar con la posiblidad de especificar productos en el modelo es tan fundamental como poder especificar los procesos que los manipulan. Además, la especificación de los productos ayuda en muchos casos a describir con mayor precisión o detalle algunos mecanismos complejos de composición de procesos. También nos brindan la posibilidad de analizar el comportamiento de procesos a través del estudio de cómo los mismos transforman los productos que manipulan. En este trabajo será de particular interés analizar las ventajas de contar con lenguajes que permitan la especificación de propiedades sobre los productos que forman parte de procesos de negocios, y su posterior análisis por medio de técnicas automáticas o semi-automáticas, como por ejemplo *model checking*.

En el Capítulo 5 mostramos un lenguaje que contempla, en la especificación de modelos, el concepto de *producto*. Este lenguaje también incorpora la noción de tiempo, brindando la posibilidad de imponer restricciones temporales a los procesos que forman parte de sus modelos. Luego, en el Capítulo 6 proponemos una técnica para analizar especificaciones de modelos de dicho lenguaje. Para tal fin, se utilizan descripciones de propiedades en una lógica temporal y la verificación de las mismas se basa en la utilización de una herramienta, que entre sus características relevantes cuenta con soporte para describir y operar con cotas de tiempo real.

Capítulo 4

Especificación y Verificación de Propiedades de Workflows

4.1 Introducción

La importancia del análisis de propiedades de *workflows* es reconocida por muchos autores, [72, 113] entre otros. Un aspecto que resulta particularmente importante en los lenguajes para la especificación de workflows, en lo relativo al análisis de sus modelos, es contar con semántica formal. Este aspecto juega un rol importante en cuanto a la automatización del análisis sobre los modelos del lenguaje en cuestión.

En este capítulo nos proponemos dos metas con un objetivo común:

- contar con un lenguaje lo suficientemente expresivo que permita caracterizar una amplia variedad de propiedades de modelos de procesos de negocios especificados como workflows, y
- desarrollar las herramientas necesarias para automatizar la verificación de propiedades expresadas en el lenguaje.

Con esta propuesta esperamos contribuir a reducir la brecha entre el poder expresivo de los lenguajes de especificación de workflows y el análisis automático de los mismos.

El lenguaje que proponemos para expresar propiedades sobre workflows es una lógica temporal, más precisamente *fluent linear time temporal logic* FLTL [37] (ver Subsección 2.3.2). Consideramos que esta lógica es adecuada para expresar formalmente propiedades sobre workflows, permitiendo caracterizar situaciones que complementan las tradicionales propiedades abordadas en otros trabajos, donde centran su análisis en propiedades generales de workflows, tales como *soundness*, ausencia de deadlock [96], etc., como se discutió en el capítulo anterior.

Para aplicar el lenguaje propuesto a modelos de workflows, proveemos una caracterización de los mismos como sistemas de transición de estados etiquetados (LTS). Si bien nuestro enfoque es básicamente independiente del lenguaje escogido, y en principio podría ser utilizado para cualquier lenguaje de modelado de workflows, elegimos a YAWL (Yet Another Workflow Language) [96] como lenguaje para el modelado de workflows. YAWL es un poderoso lenguaje con semántica formal basada en Redes de Petri, que utiliza patrones de flujo de trabajo (*workflow patterns*) [98]. Posee además un ambiente gráfico para la especificación de sus modelos.

Nuestra caracterización de modelos YAWL mediante LTS, además de brindar el soporte formal necesario para la especificación de propiedades con el lenguaje propuesto, nos permite utilizar técnicas y herramientas de análisis, como por ejemplo *model checking* [24] (ver Subsección 2.4.1). Particularmente, utilizaremos la herramienta LTSA, presentada en la Subsección 2.4.1, para verificar de manera automática la validez de una propiedad, o para generar un contraejemplo en caso de violación de la misma.

Como soporte a nuestra propuesta de análisis, en el marco de este trabajo se desarrollaron dos herramientas que realizan y promueven la aplicación de la técnica propuesta. Una de ellas, "YAWL2FSP" automatiza el proceso de traducción de modelos YAWL a LTS. La otra, "FLYAnalyser", es un asistente que facilita la especificación de propiedades en el lenguaje propuesto mediante el uso de plantillas de fórmulas y una descripción gráfica del modelo para la definición de fluentes. Además, FLYAnalyser integra el proceso general de verificación, visualizando los modelos YAWL para su análisis e interactuando con YAWL2FSP y LTSA para la verificación de las propiedades especificadas.

Este capítulo se presenta de la siguiente manera. Inicialmente describimos en detalle el proceso general de verificación junto con los fundamentos subyacentes de la caracterización de workflows (perspectiva de flujo de control) como sistemas de transición de estados. Luego se detalla dicha caracterización basada en la semántica formal de YAWL, generando como objetivo un proceso FSP (lenguaje para modelar Procesos de Estados Finitos introducido en la Subsección 2.4.1). En la Sección 4.4 mostramos la utilización de la lógica FLTL para especificar propiedades de workflows, explotando la flexibilidad y conveniencia de la caracterización propuesta para la utilización de *fluentes* en la especificación de propiedades de comportamiento de workflows. Las herramientas que asisten y automatizan el proceso de especificación y verificación propuesto se mencionan en la Sección 4.5. Validamos nuestro enfoque mediante un caso de estudio cuya complejidad nos permite ilustrar las ventajas del mismo.

Basados en la caracterización abstracta de *workflows* que provee nuestro enfoque, describimos un proceso para la generación de *workflows* aplicando restricciones

expresadas como una clase de propiedades temporales. Finalmente, con el objetivo de abordar la especificación de ciertas propiedades temporales, proponemos una extensión de FLTL que llamamos Counting Fluente Linear-time Temporal Logic. Esta extensión permite la definición de fluentes contadores que puede ser utilizados para describir propiedades por medio de expresiones aritméticas como literales de la fórmula. Además, detallamos un proceso que permite realizar *model checking* sobre este tipo de propiedades (bajo ciertas restricciones) por medio de LTSA.

4.2 Verificación de Propiedades de Workflows

La Figura 4.1 muestra el esquema general del proceso para verificar propiedades de workflows especificadas con el lenguaje propuesto. La misma ilustra que, partiendo de la descripción de un Proceso de Negocio y ante la necesidad de analizar propiedades del mismo, se espera como resultado del proceso garantizar la validez de las mismas o generar contraejemplos para aquellas que el proceso no cumple. Como contraejemplo, nos referimos a una sucesión de actividades del proceso de negocio que conducen a un estado del mismo en el cual la propiedad no se cumple.

Para lograr dicho objetivo se necesitan dos actividades de formalización. Por un lado describir el proceso de negocio (workflow en este caso particular). Por otro, especificar las propiedades a analizar. Si bien la existencia de numerosos lenguajes para el modelado de procesos de negocios contribuye a brindar soporte a la primera de las actividades mencionadas, la caracterización de propiedades y su posterior análisis generalmente no cuentan con un soporte automatizado. La dificultad de esta actividad radica en que lenguajes más expresivos son más difíciles de formalizar y automatizar por completo.

Para abordar esta dificultad, como dijimos anteriormente, proponemos la Lógica Temporal de Tiempo Lineal con Fluentes (FLTL) como lenguaje de especificación de propiedades sobre workflows. Además, para permitir su aplicación y automatizar la verificación de propiedades escritas en este formalismo, presentamos una caracterización workflows como sistemas de transición de estados etiquetados (LTS).

Con el objetivo de llevar adelante las tareas mencionadas, desarrollamos un asistente, denominado "FLYAnalyser", que brinda soporte para la especificación de las propiedades e integra el proceso general de verificación, interactuando con las herramientas "YAWL2FSP" [1] y LTSA. YAWL2FSP automatiza el proceso de traducción de una especificación de un workflow como LTS y su desarrollo forma parte de este trabajo. LTSA, introducida en la Subsección 2.4.1, es una herramienta de *model*

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

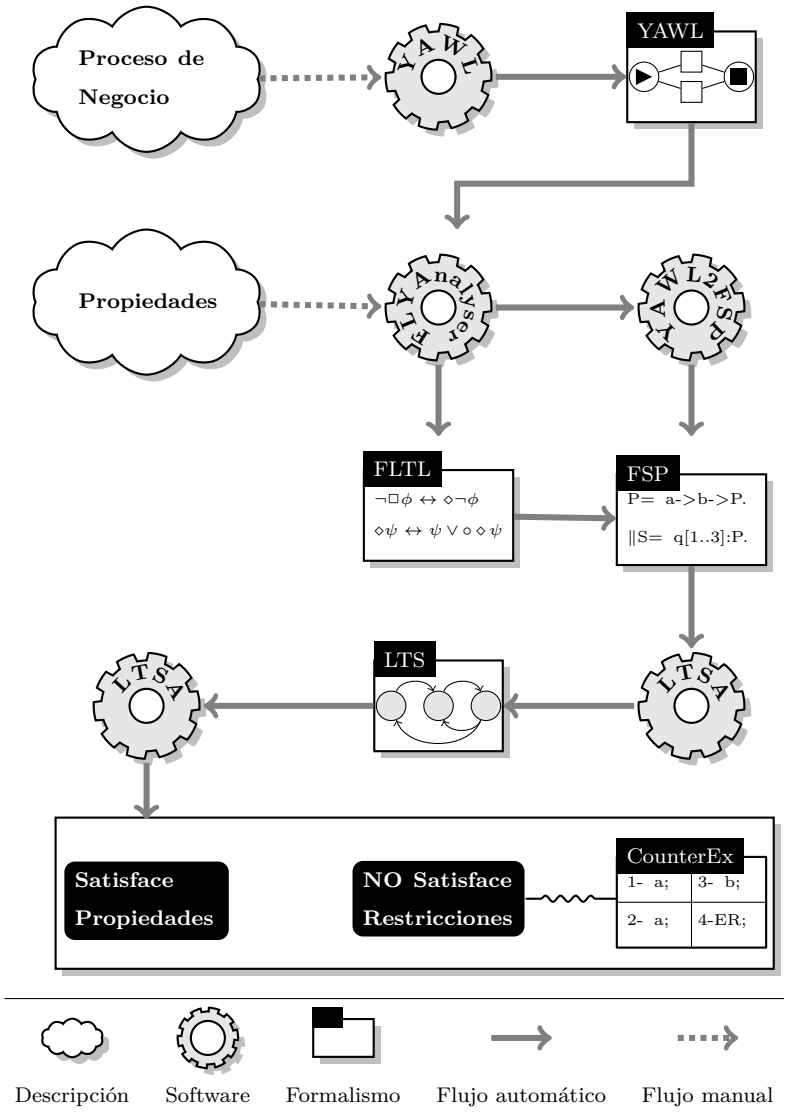


Figura 4.1: Esquema general de la verificación de Workflows

checking, que utilizamos para la verificación de propiedades descriptas con el lenguaje propuesto y lograr así el objetivo general.

El lenguaje para el modelado de workflows sobre el cual ensayamos nuestra propuesta de especificación y verificación de propiedades es YAWL (Yet Another Workflow Language) [96]. YAWL es un poderoso lenguaje orientado al uso de patrones de

flujo de trabajo (*workflow patterns*) [98]. La semántica de este lenguaje está fundada en la teoría de Redes de Petri y extensiones de la misma, tales como *Reset Nets* y *Workflow Nets*. Este lenguaje es considerado 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) [91]. 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 desde éstos a YAWL.

4.3 Caracterización de YAWL mediante FSP

El comportamiento de un sistema, especificado como un workflow, puede capturarse a través de la ocurrencia de sus tareas. De manera abstracta, podemos identificar a una tarea t como una entidad del sistema que colabora en el comportamiento del mismo mediante cierta actividad que ocurre entre el *comienzo* y *finalización* de la tarea. Estos dos eventos, que notaremos como $t.start$ y $t.end$, respectivamente, delimitan la actividad (ejecución) de una tarea. Así, una secuencia de estos eventos describen una posible ejecución del sistema y el conjunto de *todas* las posibles secuencias, es decir, todas sus posibles combinaciones de ejecuciones, el comportamiento del sistema. Cabe destacar que el conjunto de secuencias anterior, está restringido por los operadores que condicionan el flujo de control del sistema, es decir, no todas las posibles combinaciones de eventos representan necesariamente una secuencia realizable en el sistema.

Basados en esta abstracción, podemos caracterizar un workflow w como un sistema de transición de estados, donde los eventos (transiciones entre estados) simulan la actividad de los elementos que componen w. Debido al potencial tamaño de los sistemas, para la caracterización de los workflows como LTS, utilizamos como lenguaje intermedio FSP (*Finite State Processes*) [62]. Los sistemas en FSP se construyen a partir procesos definidos por sus eventos. De esta manera, para cada elemento de w se define un proceso FSP que modela su comportamiento. Para especificar el comportamiento del flujo de control, es decir las restricciones que imponen los operadores de flujo, introduciremos procesos que capturan la semántica de cada uno de ellos. Para obtener el modelo del sistema completo, se componen los procesos mencionados integrándose por medio de sincronización de sus eventos. Finalmente, las especificaciones en FSP son posteriormente traducidas, de manera automática, a sistemas de transición de estados mediante el uso de la herramienta *LTSA* (*Labelled Transition System Analyser*) [62].

Contando con esta traducción y aprovechando el poder expresivo de FLTL, po-

demos especificar y verificar situaciones del sistema de manera muy flexible. Por ejemplo, la actividad de una tarea se reduce a la especificación de un *fluente*, el cual es activado y desactivado con los eventos *start* y *end* de la tarea, respectivamente; luego, por medio de una fórmula temporal podemos analizar la ocurrencia de la misma. En dichas fórmulas no sólo pueden estar presentes los fluentes naturales que caracterizan las tareas del proceso, sino también fluentes definidos *ad-hoc* que capturen diferentes situaciones de interés que puedan ser especificadas por eventos. Por ejemplo, fluentes que caractericen la ejecución de una tarea compuesta, o una sub-secuencia de acciones, etc.

Como ejemplo simple para comprender la idea general de traducción, consideremos la especificación de la Figura 4.2. La misma ilustra una simple red YAWL con cuatro tareas. Inicialmente tiene lugar la tarea t_1, luego de manera excluyente (XOR), o bien se ejecuta t_2 o t_3, y finalmente lo hace la tarea t_4.

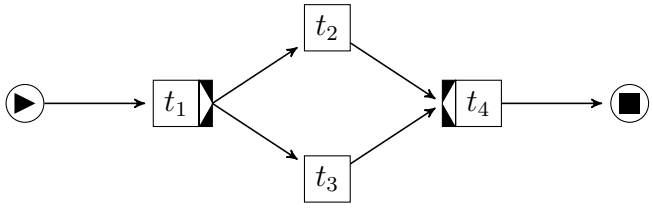


Figura 4.2: Red YAWL simple, con compuertas XOR-split y XOR-join

De acuerdo a la semántica de YAWL, el conjunto de posibles ejecuciones de las tareas es: $\{t_1t_2t_4, t_1t_3t_4\}$. En cada caso, existe la correspondiente secuencia de eventos. Por ejemplo, a $t_1t_2t_4$ le corresponde la secuencia $[t_1.start\ t_1.end\ t_2.start\ t_2.end\ t_4.start\ t_4.end]$. Como mencionamos anteriormente, la actividad de cada tarea la caracterizamos mediante un fluente. Por ejemplo, para la tarea t_2, definimos el fluente $\langle\{t_2.start\}, \{t_2.end\}\rangle$. Luego, mediante fórmulas temporales, podemos expresar propiedades del sistema. Siguiendo el ejemplo una propiedad básica a verificar sería que en la ejecución del sistema, las tareas t_2 y t_3 no pueden tener lugar simultáneamente. Como analizaremos en detalle en la Sección 4.4, esta propiedad puede ser especificada mediante la fórmula (FLTL):

$$\mathbf{G}\neg(T_2 \wedge T_3),$$

donde T_2 y T_3 son los fluentes que caracterizan las respectivas actividades de las tareas t_1 y t_2, definidos como se indicó anteriormente.

A continuación describimos la técnica de traducción de modelos YAWL a modelos FSP. La misma se lleva a cabo considerando la noción de *red*, introducida en la

Definición 4.1, y su semántica formal [91]. Teniendo en cuenta esta semántica, dada en términos de componentes, y la flexibilidad que brinda FSP para crear instancias de procesos, la traducción propuesta se basa en la definición de *esquemas* que caracterizan el comportamiento de los diferentes elementos que brinda YAWL para la especificación de sus modelos. Así, la traducción de los modelos se obtiene como la composición paralela de instancias de estos esquemas, respetando, por medio de sincronización, la relación de flujo.

Inicialmente explicamos cómo traducir las nociones de red, tarea y condición. Seguido, analizamos en detalle la traducción de los operadores de flujo de control que pueden tener asociados las tareas. Finalmente analizamos y proponemos las traducciones para construcciones más complejas, como *regiones de cancelación* y *tareas múltiples y compuestas*.

Cabe destacar que para la traducción de condiciones, debido a la restricción de los LTS a una cantidad finita de estados, limitamos el comportamiento de las condiciones a un número limitado de *tokens* que pueden residir en ellas. Si bien es una restricción importante, todo modelo YAWL puede ser codificado como un modelo FSP (de manera predeterminada el límite es la cantidad de entradas que tiene la condición). El efecto de nuestra limitación puede conducir a la presencia de *falsos positivos*[2], correspondientes a reportes de *deadlocks*. Pese a ello, nuestro análisis es *conservativo*, esto es, si no existen violaciones en nuestro modelo, entonces el modelo concreto garantiza la validez de la propiedad analizada.

Consideremos la definición de una red YAWL:

Definición 4.1. *Una red YAWL es una tupla* $(nid, C, i, o, T, T_A, T_C, M, F, Split, Join, Default, Rem, Nofi)$*, donde:*

- *nid es el identificador de la red YAWL.*
- C *es el conjunto de condiciones.*
- $i \in C$ *y* $o \in C$ *son condiciones especiales, inicial (input) y final (output) respectivamente.*
- T *es el conjunto de tareas compuesto por los siguientes conjuntos disjuntos:*
  - $T_A \subseteq T$ *es el conjunto de tareas atómicas,*
  - $T_C \subseteq T$ *es el conjunto de tareas compuestas,*
  - $M \subseteq T$ *es el conjunto de tareas múltiples (varias instancias de la misma).*

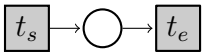
[2]Son violaciones a propiedades en el modelo abstracto que no representan ejecuciones válidas en el modelo concreto.

- $F \subseteq (C \backslash \{o\} \times T) \cup (T \times C \backslash \{i\}) \cup (T \times T)$ *es la relación de flujo; cada nodo en el grafo* $(C \cup T, F)$ *está en un camino directo desde* i *a* o*.*
- $Split : T \nrightarrow \{AND, XOR, OR\}$ *función parcial que asocia a una tarea el comportamiento de bifurcación del flujo de control.*
- $Join : T \nrightarrow \{AND, XOR, OR\}$ *función parcial que asocia a una tarea el comportamiento de unión del flujo de control.*
- $Default \subseteq F$ *define la salida predeterminada de una puerta OR-Split, asegurando que al menos una salida está habilitada.*
- $Rem : T \nrightarrow \mathbb{P}^{+}(T \cup C \backslash \{i, o\})$ *función parcial que asocia a cada tarea cancelante* t_c *las tareas o condiciones a cancelar (quitando sus tokens) como consecuencia de su ejecución.*
- $Nofi : M \rightarrow \mathbb{N} \times \mathbb{N} \times \mathbb{N} \times \{$*dinámico, estático*$\}$ *define la configuración para las tareas múltiples: cantidad mínima y máxima de instancias, umbral de continuación y su forma de creación.*

Sea N una red YAWL. El proceso `YNET` que codifica el *comienzo* y *fin* de una red es el siguiente:

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

La semántica de las tareas atómicas en YAWL está definida por la siguiente Red de Petri.



Para cada *tarea atómica* del modelo se genera una instancia del siguiente proceso `TASK` que captura su semántica:

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

El proceso simplemente identifica los eventos de comienzo y fin de la una tarea.

En YAWL, la semántica de una *condición* se define de manera directa como un *sitio* en el formalismo subyacente. La traducción de las condiciones, como lo mencionamos anteriormente, se restringe a un número limitado de *tokens* posibles que puedan residir en ellas. Con esta limitación, la caracterización de las condiciones es un proceso parametrizado con la cantidad de conexiones entrantes y salientes, como se muestra a continuación:

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

Las conexiones entrantes y salientes son codificadas como eventos. El proceso especifica que, comenzando en un estado sin *tokens*, en la medida que la capacidad lo permita, puede tener lugar cualquiera de los eventos `in` incrementando la cantidad de *tokens*, y en la medida que éstos existan, podrá tener lugar el evento `out` cuyo efecto es decrementar dicha cantidad. La capacidad predeterminada de *tokens* se establece en base a las conexiones entrantes.

4.3.1 Relación de flujo de trabajo

Para codificar la relación de flujo podemos analizar las siguientes situaciones:

- Cómo codificar el flujo entre tareas atómicas.
- Cómo codificar el flujo entre tareas y condiciones.
- Cómo codificar el flujo en presencia de bifurcaciones y uniones de flujo.

Para codificar el flujo entre tareas atómicas, consideremos $t_1, t_2 \in T_A$ tal que $(t_1 \notin Dom(Split) \wedge t_2 \notin Dom(Join))$ tareas atómicas sin compuertas de unión o bifurcación. La codificación del flujo desde t_1 hacia t_2 se realiza mediante la sincronización de los eventos `end` y `start` de t_1 y t_2, respectivamente.

El proceso FSP que codifica las condiciones contiene tantos eventos indexados `in` (`out`) como flujos de entrada (salida) tiene una condición YAWL. De manera similar que para las tareas, la codificación de modelos con presencia de condiciones se obtiene mediante la sincronización de los eventos `end / in` y `out / start` de las tareas y condiciones para los flujos entrantes y salientes, respectivamente.

Las sincronizaciones de los eventos entre los correspondientes procesos se efectúa mediante la técnica de *renombramiento*, como se muestra en la siguiente codificación de un modelo YAWL con dos tareas atómicas consecutivas intercaladas por una condición (Figura 4.3).

```
1 // ---------------PRELUDE---------------- //
2 YNET = (i_cond ->o_cond -> YNET).
3 /*** TASK templates ***/
4 TASK = (start ->end -> TASK).
5 /*** CONDITION templates ***/
6 CONDITION (IN=2,OUT=2) = STATE[0],
```

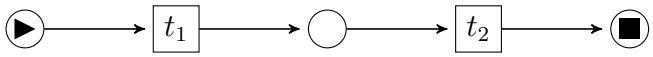


Figura 4.3: "TWO_TSK_COND": Modelo YAWL con dos tareas atómicas consecutivas y una condición intermedia

```
7  STATE[s:0..IN] = (when (s<IN) in[i:1..IN] ->STATE[s+1]
8                   |when (s>0) out[j:1..OUT] ->STATE[s-1] ).
9  // ----------------SYSTEM---------------- //
10 ||TWO_TSK_COND = (
11        /*** BASIC STUFF ***/
12               YNET
13        || start_cond : CONDITION(1,1)
14        || end_cond : CONDITION(1,1)
15        /*** TASKS ***/
16               // t1
17        || task[1] : TASK
18               // t2
19        || task[2] : TASK
20        /*** CONDITIONS ***/
21        || cond[1] : CONDITION(1,1)
22 )
23 / {
24        i_cond/start_cond.in[1],
25        task[1].start/start_cond.out[1],
26        task[1].end/cond[1].in[1],
27        task[2].start/cond[1].out[1],
28        task[2].end/end_cond.in[1],
29        o_cond/end_cond.out[1]
30 }.
```

El proceso FSP generado contiene inicialmente la especificación de esquemas para los tipos de elementos de YAWL (líneas 1-8). Luego, desde la línea 9 la definición del proceso correspondiente al modelo particular a codificar. Entre la línea 9 y la 22, la composición paralela de los elementos del modelo (instancias de esquemas). Finalmente, las líneas 23-29, describe la especificación mediante el renombramiento, de las conexiones entre los elementos.

A continuación describimos cómo se codifica el flujo de control en presencia de bifurcaciones o uniones. Para ello inicialmente describimos el proceso de manera abstracta para cualquier tipo de bifurcación y de unión, las cuales denominaremos

compuertas. Luego se explora en detalle la traducción para cada una de ellas.

Consideremos $t \in T_A$ tal que $(t \in Dom(Split) \vee t \in Dom(Join))$. Es decir, t es una tarea atómica decorada con algún tipo de compuerta (`AND, OR, XOR`). Para cada tipo de compuerta definimos un proceso esquema que especifica su comportamiento. Estos procesos están parametrizados con las correspondientes entradas (`in`) y salidas (`out`), a los que llamaremos *puertos*. Por ejemplo, una compuerta de unión puede tener dos o más puertos de entrada y sólo uno de salida, mientras que, simétricamente, una compuerta de bifurcación sólo puede tener un puerto de entrada y varios de salida.

Gráficamente, en la Figura 4.4 se ejemplifica una tarea atómica t decorada con dos compuertas abstractas de unión J y bifurcación S. Para especificar la conexión entre ellas, se sincroniza `J.out` con `t.start`, y `t.end` con `S.in`, respectivamente. Luego las tareas (condiciones) previas de t sincronizan sus eventos de finalización `end` (de salida `out`) con los puertos de entrada de J, y, simétricamente, los puertos de salida de S con los eventos de comienzo `start` (de entrada `in`) con las tareas (condiciones) posteriores de t.

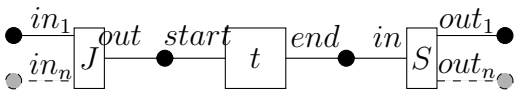


Figura 4.4: Configuración abstracta de una tarea con compuertas de bifurcación y unión.

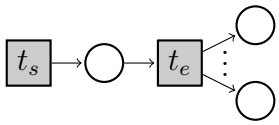
4.3.2 Codificación de Compuertas

Para cada tipo de compuerta se genera un proceso FSP que captura su comportamiento. Estos procesos tienen como parámetros la cantidad de puertos de entrada y salida. Al igual que para las tareas y condiciones, estos esquemas son instanciados en correspondencia con los elementos que conforman el modelo YAWL a codificar.

A continuación detallamos para cada uno de los tipos de compuertas el correspondiente proceso FSP, comenzando por las compuertas de bifurcación y luego por la de unión.

AND-Split

La semántica formal de las compuertas `AND-Split`, como se muestra a continuación, traslada el flujo control a cada uno de los elementos que se conectan a la misma.



Su codificación como un proceso FSP se especifica de manera compuesta, esto es, a través de un proceso que consta de varios procesos paralelos llamados *triggers*, tantos como flujos salientes existan en la compuerta:

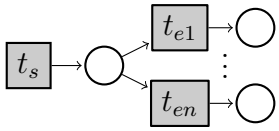
```
1  AND_SPLIT_COMPONENT(I=1) = (
2          in -> TRIGGERING
3        | reset -> AND_SPLIT_COMPONENT
4  ),
5  TRIGGERING = (
6          out[I] -> AND_SPLIT_COMPONENT
7        | reset -> AND_SPLIT_COMPONENT
8  ).
9  ||AND_SPLIT(N=2) = (
10       forall[i:1..N] AND_SPLIT_COMPONENT(i)
11 ).
```

El proceso `AND_SPLIT` lanza en forma paralela (línea 10) tantos componentes como el parámetro indica. Cada componente espera de manera sincronizada por la ocurrencia del evento `in` (línea 2) que estará asociado a la finalización de la tarea decorada. Cuando éste ocurre, se transfiere el control a todas las subsiguientes tareas asociadas con cada puerto de salida mediante el evento `out` (línea 6). Los eventos `reset` especificados en las líneas 3 y 7 corresponden al reinicio de las compuertas y serán discutidos en la Subsección 4.3.6.

En la Figura 4.5 se puede observar el LTS correspondiente a la especificación FSP anterior. El mismo consta de 4 estados. Una vez que ocurre el evento `in`, ocurren los eventos `out` secuencialmente, en cualquier orden. Además, en cualquier estado, la compuerta puede ser reiniciada.

XOR-Split

La semántica formal de estas compuertas traslada el flujo control a exactamente uno de los elementos que se conectan a la misma:



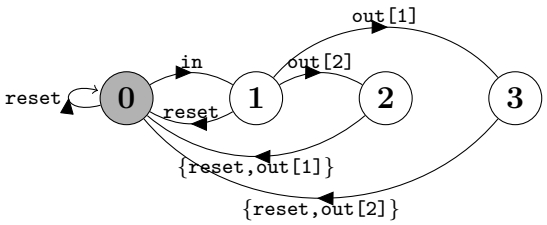


Figura 4.5: LTS correspondiente a la especificación FSP de una compuerta de bifurcación `AND` con dos puertos de salida.

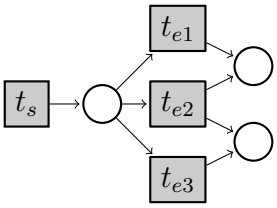
La compuerta de bifurcación `XOR` se codifica de la siguiente manera:

```
1 XOR_SPLIT(N=2) = (
2         in -> SPLITING
3       | reset -> XOR_SPLIT
4 ),
5 SPLITING = (
6         out[1..N] -> XOR_SPLIT
7       | reset -> XOR_SPLIT
8 ).
```

El proceso consta de un estado en el cual se aguarda la finalización de la tarea asociada mediante el evento `in` (línea 2). Una vez que éste ocurre, sólo uno de los posibles eventos `out` (línea 6), puertos asociados a las tareas siguientes, puede tener lugar, transfiriendo el control a la misma.

OR-Split

La siguiente Red de Petri modela el comportamiento de una compuerta de bifurcación `OR` de YAWL con dos puertos de salida. Además, como está detallado en la Definición 4.1, en YAWL se especifica para este tipo de compuerta cuál es la opción por defecto.



De manera similar a las compuertas de bifurcación `AND`, el proceso FSP que especifica el comportamiento de las compuertas `OR-Split` es un proceso compuesto.

El mismo consta de dos partes, un conjunto de *triggers* que pueden disparar el flujo saliente o abortar su ejecución, y un proceso que contabiliza cuántos *triggers* abortaron. En caso de que todos los *triggers* aborten, éste dispara el flujo de salida por defecto.

```
1  OR_SPLIT_TRIGGER(I=1) = (
2          in -> TRIGGERING
3        | reset -> OR_SPLIT_TRIGGER
4  ),
5  TRIGGERING = (
6          out[I] -> OR_SPLIT_TRIGGER
7        | abort[I] -> OR_SPLIT_TRIGGER
8        | reset -> OR_SPLIT_TRIGGER
9  ).
10
11 OR_SPLIT_DEFAULT_TRIGGER(N=2) = (
12         in -> OR_SPLIT_DEFAULT_TRIGGER_DEF[0][0]
13       | reset -> OR_SPLIT_DEFAULT_TRIGGER
14 ),
15 OR_SPLIT_DEFAULT_TRIGGER_DEF[i:0..N][j:0..N] = (
16         reset -> OR_SPLIT_DEFAULT_TRIGGER
17       | when(i+j<N) abort[k:1..N] ->OR_SPLIT_DEFAULT_TRIGGER_DEF[i+1][j]
18       | when(i+j<N) out[k:1..N] ->OR_SPLIT_DEFAULT_TRIGGER_DEF[i][j+1]
19       | when(i+j==N && j==0) default_out ->OR_SPLIT_DEFAULT_TRIGGER
20       | when(i+j==N && j!=0) invisible-> OR_SPLIT_DEFAULT_TRIGGER
21 ).
22
23 ||OR_SPLIT(N=2) = (
24          forall [i:1..N] OR_SPLIT_TRIGGER(i)
25       || /+OR_SPLIT_DEFAULT_TRIGGER(N)
26 ).
```

El proceso `OR_SPLIT` (línea 23-25) lanza en forma paralela tantos componentes `OR_SPLIT_TRIGGER` (líneas 1-9) como el parámetro indica. Además, junto con los anteriores, dispara el proceso `OR_SPLIT_DEFAULT_TRIGGER`. Cada instancia `OR_SPLIT_TRIGGER` espera de manera sincronizada por la ocurrencia del evento `in` (línea 2) que estará sincronizado con la finalización de tarea asociada. Una vez que este evento ocurre, las instancias pueden transferir el control a las tareas subsiguientes asociadas a los puerto de salida mediante el evento `out` (línea 6), o abortar dicha transferencia.

Dado que todas las instancias pueden abortar, el proceso `OR_SPLIT_DEFAULT_TRIGGER` (líneas 11-21), que sincroniza su comienzo con el evento `in` (línea 12), monitorea el comportamiento de los *triggers* sincronizando con los procesos `abort`

y `out` (línea 17-18) correspondientes. En caso de que *todos* aborten, éste transfiere el control a la tarea asociada al puerto predeterminado `default_out` (línea 19). Al igual que en las demás compuertas, los eventos `reset` corresponden al reinicio de las compuertas, y serán discutidos en la Subsección 4.3.6.

En la Figura 4.6 se puede observar el LTS generado a partir de la especificación anterior, con dos puertos de salida. El mismo se encuentra simplificado, sin los eventos `reset` (desde los estados 0 a 7 hacia el estado 0), para mejorar su visualización.

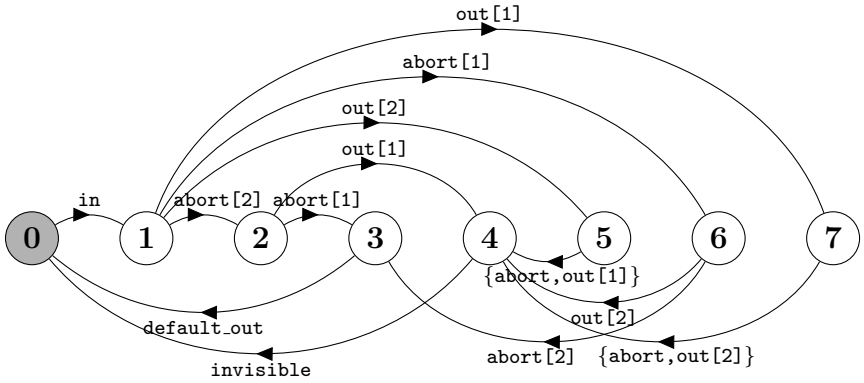
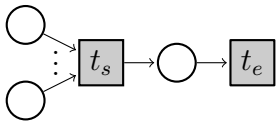


Figura 4.6: LTS (simplificado) correspondiente a la compuerta de bifurcación `OR` con dos salidas.

AND-Join

Como lo especifica la siguiente Red de Petri, el comportamiento de las compuertas de unión `AND` requiere la capacidad de sincronizar varios flujos de ejecución en un punto del sistema.



Su codificación en FSP se obtiene sincronizando todos los flujos entrantes mediante un mismo evento. Una vez que éste ocurre, la ejecución continua con la tarea decorada con la compuerta.

```
1 AND_JOIN_COMPONENT(I=1) = (
2         in[I] -> JOINING
```

```
3         | reset -> AND_JOIN_COMPONENT
4 ),
5 JOINING = (
6           out -> AND_JOIN_COMPONENT
7         | reset -> AND_JOIN_COMPONENT
8 ).
9
10 || AND_JOIN(N=2) = (
11        forall [i:1..N] AND_JOIN_COMPONENT(i)
12 ).
```

El proceso `AND-Join` (línea 10) que captura el comportamiento de este tipo de compuerta consta de varios procesos, un por cada flujo entrante. Estos procesos esperan la llegada del correspondiente flujo de control (línea 2). A partir de allí, pasan a un estado donde *todos* los componentes deben sincronizar por medio de la acción compartida `out` (línea 6). Como puede observarse en el LTS correspondiente a este proceso, Figura 4.7, sólo cuando todos los eventos `in` han ocurrido (en cualquier orden), el evento `out` puede acontecer.

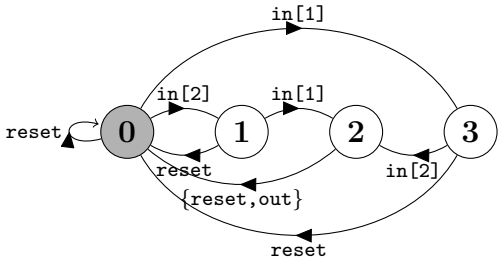
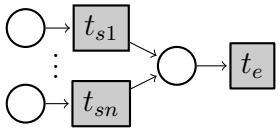


Figura 4.7: LTS correspondiente a la compuerta de unión `AND` con dos entradas.

XOR-Join

La semántica de las compuertas de unión `XOR` se define mediante la siguiente Red de Petri:



El comportamiento de estas compuertas puede codificarse a través de un proceso FSP, de la siguiente manera:

```
1 XOR_JOIN(N=2) = (
2         in[1..N] -> JOINING
3       | reset -> XOR_JOIN
4 ),
5 JOINING = (
6         out -> XOR_JOIN
7       | reset -> /+XOR_JOIN
8 ).
```

Este proceso tiene dos estados. En el primero, espera de manera paralela el arribo de flujos de trabajos de las tareas asociadas (línea 2). Luego, ante la ocurrencia de *una* de ellas, pasa al segundo estado, que se encarga de disparar el evento `out` (línea 6), transfiriendo el control a la tarea decorada. En la Figura 4.8 podemos observar gráficamente el comportamiento de la especificación anterior mediante el LTS generado por el mismo.

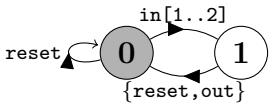


Figura 4.8: LTS correspondiente a la compuerta de unión `XOR` con dos entradas.

OR-Join

Debido a su semántica no local, este tipo de compuertas tiene diferentes interpretaciones dependiendo del lenguaje para la especificación de procesos de negocios elegido. En [91] podemos encontrar una recopilación de las diferentes interpretaciones de la compuerta de unión `OR` en diferentes lenguajes, como Business Process Modeling Notation (BPMN), Event-driven Process Chains (EPCs) (ver también [54]), etc. Además, se discute la complejidad del análisis de este tipo de compuertas en presencia de regiones de cancelación, ciclos, etc.

Si bien en YAWL la evaluación de estas compuertas se analiza por medio de *backward firing* y *cobertura* de las reset nets, la codificación que abordamos se basa en la semántica informal de las mismas (cf. [91], pág. 104). Esta interpretación prioriza la llegada de eventos sobre la transferencia del control a la tarea asociada como salida.

El proceso FSP que captura esta semántica es el siguiente:

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

Como se puede observar gráficamente en la Figura 4.9, el LTS generado por la especificación anterior, este proceso consta de dos estados. El primero espera por el arribo de un flujo entrante. Ante su ocurrencia, el proceso pasa al segundo estado, donde puede seguir recibiendo flujos entrantes o puede pasar el control a la tarea decorada mediante el evento `out`.

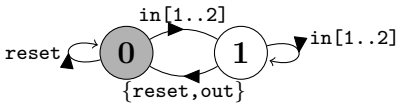


Figura 4.9: LTS correspondiente a la compuerta de unión `OR` con dos entradas.

A modo de ejemplo presentamos la codificación del modelo YAWL de la Figura 4.2:

```
// ----------------PRELUDE---------------- //
...
// ----------------SYSTEM---------------- //
|| SIMPLE = (
              ynet : YNET
       ||     in_cond : CONDITION(1,1)
       ||     out_cond : CONDITION(1,1)
       ||     t1 : TASK
       ||     t1_split : XOR_SPLIT(2)
       ||     t2 : TASK
       ||     t3 : TASK
       ||     t4_join : XOR_JOIN(2)
       ||     t4 : TASK
)
/ {
       ynet.i_cond/in_cond.in[1],
       t1.start/in_cond.out[1],
       t1.end/t1_split.in,
```

```
19        t2.start/t1_split.out[1],
20        t3.start/t1_split.out[2],
21        t2.end/t4_join.in[1],
22        t3.end/t4_join.in[2],
23        t4.start/t4_join.out,
24        t4.end/out_cond.in[1],
25        ynet.o_cond/out_cond.out[1],
26        //----reset gates sincronizations
27        ynet.o_cond/t1_split.reset,
28        ynet.o_cond/t4_join.reset
29 }.
```

Figura 4.10: Modelo FSP correspondiente al workflow de la Figura 4.2

.

Como lo mencionamos anteriormente, el proceso consta de una primera parte (condensado en la línea 2 "...") donde se encuentra la definición de los procesos esquemas detallados anteriormente, correspondientes a los elementos que forman parte del modelo original. Particularmente, en este ejemplo se encuentran los esquemas correspondientes a tareas (`TASK`), condiciones (`CONDITION`), y a las compuertas de bifurcación y unión exclusivas, `XOR_SPLIT` y `XOR_JOIN`, respectivamente.

En la segunda parte (líneas 4-29) se encuentra la especificación del modelo mediante un proceso paralelo, compuesto por las correspondientes instancias de los elementos (líneas 5-13) y su conexión (relación de flujo) por medio de sincronizaciones, como se describió anteriormente. En la Figura 4.11 se encuentra el LTS generado a partir de esta especificación.

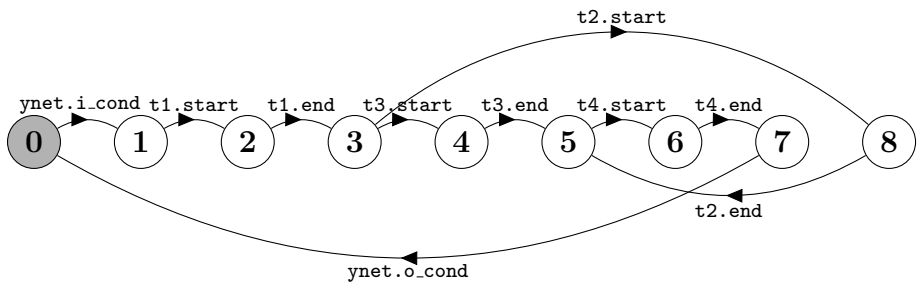
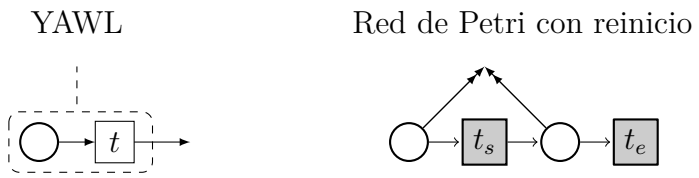


Figura 4.11: LTS correspondiente al modelo YAWL de la Figura 4.2.

4.3.3 Regiones de Cancelación

Las *regiones de cancelación* permiten abortar o inhabilitar un conjunto de tareas o condiciones, es decir, capturan la interferencia entre la ejecución de tareas. Como se define en una especificación YAWL (Definición 4.1), una tarea puede tener asociado un conjunto de tareas o condiciones a *cancelar*. Su semántica, en *reset nets* es la siguiente:



Cuando se ejecuta una tarea que tiene una región de cancelación asociada, todos los sitios de la Red de Petri que especifican los componentes de dicha región son vaciados, es decir, se eliminan los *tokens* que pudieran contener.

Para codificar las regiones de cancelación como procesos FSP, se extienden los procesos para las tareas, condiciones y compuertas que pertenecen a estas regiones. Esta extensión modela la posibilidad de que dichos elementos del lenguaje que se encuentran dentro de una región de cancelación puedan ser canceladas. Por ejemplo, en el caso de las tareas, una vez que comienzan su ejecución pueden terminar o ser canceladas. Los procesos que modelan este comportamiento para las tareas y condiciones son los siguientes:

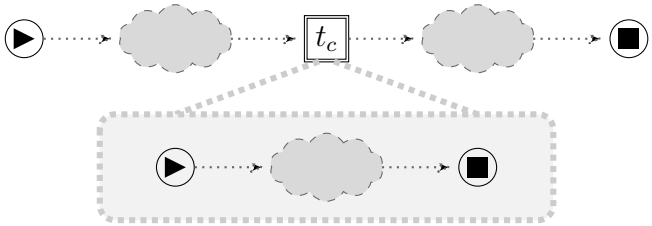
```
CANCELABLE_TASK = CTASK_INACTIVE,
CTASK_INACTIVE = (
         start -> CTASK_ACTIVE
       | cancel -> CTASK_INACTIVE
),
CTASK_ACTIVE = (
         end -> CTASK_INACTIVE
       | cancel -> CTASK_INACTIVE
).

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

Todos los eventos `cancel` correspondientes a las instancias de tareas (líneas 4 y 8) y condiciones (línea 15) que se encuentran dentro de una *región de cancelación*, se sincronizan con el evento `start` de la *tarea cancelante*. De esta manera, cuando esta tarea comienza su ejecución cancela todas las condiciones y tareas que tenga asociada a su región.

4.3.4 Tareas Compuestas

Como lo muestra el siguiente diagrama, en YAWL una tarea puede sintetizar la descripción compleja de un proceso. A éstas se les denomina *tareas compuestas* y tienen asociadas una sub-red YAWL que describe su comportamiento.



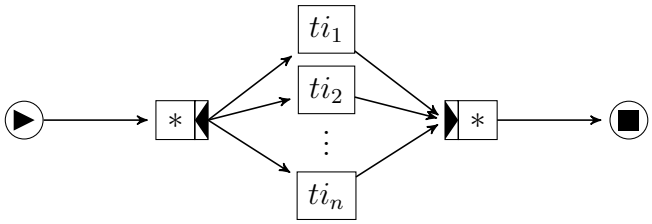
Las tareas compuestas se codifican en FSP mediante la generación de un proceso compuesto que representa la red asociada. Esta codificación se realiza siguiendo los pasos descriptos hasta el momento: codificación de tareas, condiciones, comienzo y fin, compuertas, etc. En caso de contener tareas compuestas, se aplica recursivamente el método. Luego, en el proceso que representa el sistema se instancia el proceso que describe la tarea compuesta y se sincronizan los flujos entrantes y salientes a la misma con los eventos `i_cond` y `o_cond` del proceso que la representa, respectivamente.

4.3.5 Tareas Múltiples

YAWL permite capturar el concepto de instancias de tareas mediante la noción de tareas múltiples. Éstas sintetizan el comportamiento de múltiples instancias de una misma tarea. Según su configuración, ver Definición 4.1, la especificación consta de los límites inferior y superior de cantidad de instancias, el umbral de continuación, y si las instancias se crean de manera *estática* o *dinámica*. El umbral de continuación especifica cuántas instancias finalizadas son suficientes para continuar con la próxima tarea o condición asociada a la tarea múltiple mediante la relación de flujo.

La codificación de este tipo de tareas se realiza mediante una descomposición de las mismas en redes subyacentes que modelan su comportamiento. De esta manera, podemos codificar una tarea de múltiples instancias, de manera similar a las tareas compuestas.

Para el caso de tareas estáticas, una posible descomposición en una red puede ser la siguiente:

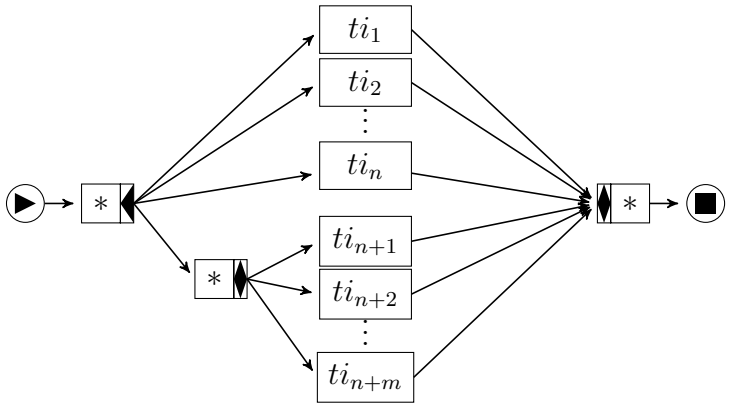


Se lanzan paralelamente, mediante una compuerta de bifurcación `AND` tantas tareas atómicas como instancias se indiquen. Dado que se debe respetar el *umbral de continuación*, la sincronización de las instancias se realiza mediante la extensión de la compuerta de unión `AND` que se define a continuación.

```
MI_AND_JOIN(N=2,T=2) = (
       in[k:1..N] -> MI_AND_JOIN_DEF[1]
),
MI_AND_JOIN_DEF[i:0..N]= (
         when (i<N  && i!=T) in[k:1..N] ->MI_AND_JOIN_DEF[i+1]
       | when (i==T && T<N ) out ->in[k:1..N] ->MI_AND_JOIN_DEF[i+1]
       | when (i==N && T==N) out ->MI_AND_JOIN
       | when (i==N && T<N ) end ->MI_AND_JOIN
).
```

Esta compuerta `AND` sincroniza sus entradas (`in`) con los eventos de finalización de las instancias y, a diferencia de la original, dispara su evento de salida (`out`) cuando se alcanza la cantidad de eventos de entrada que indica el umbral.

Para el caso de creación dinámica de instancias, tenemos una cantidad mínima de instancias, y dinámicamente pueden crearse más instancias hasta un límite dado. La red que modela su comportamiento consta de una compuerta de bifurcación `AND` conectada con un número de instancias igual al límite inferior, y además activa una compuerta de bifurcación `OR` (sin salida predeterminada). Esta compuerta `OR` tiene asociadas a sus puertos de salida un número de tareas igual a la diferencia entre límite superior e inferior. Luego, todas las instancias de tareas se conectan a una compuerta de unión `OR` como la especificada para las tareas múltiples estáticas. Esta compuerta compromete su salida a la finalización de k instancias de tareas asociadas a su entrada, donde k se instancia con el *umbral de continuación*. Gráficamente:



4.3.6 Reinicio de compuertas

Como se observa en las especificaciones anteriores, todas las compuertas de bifurcación y unión poseen el evento especial `reset`. Mediante este evento, todas las instancias de las compuertas se sincronizan con el final del sistema o subsistena (sistema que se corresponde con una tarea compuesta) `o_cond`. De esta manera, cuando el LTS correspondiente a una red YAWL alcanza el estado final, asegura que todas las compuertas vuelven a su estado inicial.

Esta consideración se realiza para evitar el análisis de trazas espurias, ya que si una compuerta queda con alguno de sus componentes activos, puede iniciar flujos aún cuando el sistema hubiere alcanzado el final del sistema. Si bien esto es equivalente a que en Redes de Petri, quedasen *tokens* en lugares de la red (redes que no gozan de la propiedad de *soundness*) sólo puede suceder en *sitios* y no en *transiciones*, ya que éstas últimas son atómicas.

4.3.7 Corrección de la traducción

La semántica de YAWL capturada por el formalismo subyacente para algunos de sus elementos está condicionada en pos de su análisis. Como lo mencionamos anteriormente, la compuerta OR-JOIN es un ejemplo de ello. Por otro lado, en nuestro enfoque aprovechamos la flexibilidad del formalismo para capturar de manera natural los patrones que fundan los modelos de YAWL. Más aún, la semántica formal de la composición de patrones en términos de reset nets no se encuentra de manera explícita, aunque puede asumirse la composición mediante la interconexión entre los sitios y transiciones correspondientes. Debido a esto, podemos enunciar la corrección de la traducción de manera atómica, es decir, demostrar que para cada patrón de flujo que provee YAWL como componente del lenguaje, su comportamiento, como trazas eventos, coincide con la semántica propuesta en términos de LTSs. Para ello,

consideramos las trazas de ejecución, las cuales están compuestas por los eventos de disparos de transiciones. Por ejemplo, para la semántica de las tareas que provee YAWL, como se muestra en A.3, la traza producto del disparo de las correspondientes transiciones t_s y t_e es $t_s t_e$. Como puede comprobarse, esta traza coincide con aquella capturada por el proceso FSP correspondiente a una tarea $t.start\ t.end$.

Para las compuertas de sincronización:

- AND-JOIN: Considerando los eventos de disparo de la transiciones de la que captura la semántica de esta compuerta, las trazas de ejecución indican que deben suceder alguna combinación de *todos* los eventos de finalización de las n tareas sincronizadas previo al comienzo de la tarea decorada t, por ejemplo para $n = 2$, el conjunto de trazas $\{t_{1e}t_{2e}t_s, t_{2e}t_{1e}t_s\}$. El proceso FSP que codifica esta compuerta especifica n eventos *in* que son sincronizados con los correspodinetes eventos *end* de las tareas sincronizadas y un evento *out* que se sincroniza con el evento *start* de la tarea decorada. Las trazas que especifica este proceso modela la ocurrencia, en algun orden, de *todos* los n eventos *in* antes de la ocurrencia de *out*. Considerando las sincronizaciones mencionadas, y para el caso de $n = 2$, el conjunto de trazas que caracteriza este proceso es $\{t_1.end\ t_2.end\ t.start, t_2.end\ t_1.end\ t.start\}$.

- XOR-JOIN: Con un análisis análogo al antetior, las trazas que captura la semántica de esta compuerta indican que debe suceder *exclusivamente* alguno de los eventos de finalización de las n tareas sincronizadas previo al comienzo de la tarea decorada t, por ejemplo para $n = 2$, el conjunto de trazas $\{t_{1e}t_s, t_{2e}t_s\}$. El proceso FSP que codifica esta compuerta especifica que ante la ocurrencia de *uno* de los n eventos *in* segidamente ocurre *out*. Nuevamente, considerando las sincronizaciones con las tareas sincronizadas y la tarea decorada, para el caso de $n = 2$, el conjunto de trazas que caracteriza este proceso es $\{t_1.end\ t.start, t_2.end\ t.start\}$.

- OR-JOIN: Para este tipo de compuertas, YAWL propone una semáncia diferente a la que usualmente se espera para un operador OR. Esto sucede debido que una de las condiciones que requieren los modelos para ser considerados *sound* es *proper completion* que requiere que, una vez finalizada la ejecución, no existan flujos de control (tokens) presentes en los sitios de la red subyacente. Nuestra propuesta de análisis de propiedades considera la seántica usual de este tipo de compuertas considerando todas las posibles trazas de ejecución que éstas inducen.

Para las compuertas de distribución:

- AND-SPLIT: Las trazas de eventos generada capturas por la semántica de esta compuerta, muestran la finalización de la tarea decorada t_e y luego, alguna combinación de *todos* los eventos de comienzo de las n tareas conectadas a la compuerta, por ejemplo para $n = 2$, el conjunto de trazas $\{t_e t_{1s} t_{2s}, t_e t_{2s} t_{1s}\}$. El proceso FSP que codifica esta compuerta especifica n eventos *out* que son sincronizados con los correspodinetes eventos *start* de las tareas sincronizadas y un evento *in* que se sincroniza con el evento *end* de la tarea decorada. Las trazas del LTS que especifica este proceso, considerando las sincronizaciones mencionadas coinciden con las exhibidas anteriormente. Siguiendo el ejemplo mencionado, para el caso de $n = 2$, el conjunto de trazas que caracteriza este proceso es $\{t.end\ t_1.start\ t_2.start, t.end\ t_2.start\ t_1.start\}$.

- XOR-SPLIT: Para esta compuerta, las trazas de disparos de la red de petri subyacente indican que, una vez que ocurre el evento de finalización de la tarea decorada, debe suceder *exclusivamente* alguno de los eventos de comienzo de las n tareas asociadas a esta compuerta t, por ejemplo para $n = 2$, el conjunto de trazas $\{t_e t_{1s} t_s, t_e t_{2s}\}$. Las trazas del proceso FSP que codifica esta compuerta especifica coincide con este conjunto una vez sincronizados los eventos *in* y *out* con las respectivas tareas como lo mencionamos anterioremente.

- OR-SPLIT: Esta compuerta evidencia algunas diferencias entre la semántica propuesta en el lenguaje y aquella que expresa su semántica formal, más precisamente, como parte del lenguaje esta compuerta, admás del comportamiento esperado que espeficias el operador lógico OR, permite determinar cuál de todos sus flujos emergentes es el *predeterminado*. Esta característica no es reflejada explícitamente por la red de petri propuesta que especifica su semántica formal. Para brindar un anális que refleje esta situación, el proceso FSP que proponemos apra caputara su semántica modela esta situación.

La semántica formal que provee YAWL para las regiones de cancelación utiliza el concepto de arco de reseteo, de manera abstracta, si una tarea t ha comenzado su ejecución, es decir, tuvo lugar el disparo de la trasición correspondiente a su comienzo t_s, y dicha tarea es cancelada, no puede ocurrir el disparo de la transición correspondiente a su finalización t_e. Observando las trazas de ejecución, esta situación coincide con aquellas trazas generadas por los procesos correspondientes a las tareas cancelables, donde una vez ocurrido el evento de comienzo $t.start$ y ante la situación de ser canceladas, tiene lugar el evento $t.cancel$ llevando la tarea a un estado donde sólo puede volver a comenzar e impidiendo la ocurrencia de su evento de finlización $t.end$.

4.4 Especificación de Propiedades

En los últimos años, trabajos como [51, 72, 73, 97, 113], entre otros, demuestran el interés en en análisis de procesos de negocios. A diferencia de las propuestas existentes, donde predomina un análisis estructural de propiedades generales, tales como *soundness*, ausencia de *deadlocks*, o esquemas de comportamiento, nuestra propuesta consiste en plantear el uso de un lenguaje lo suficientemente expresivo para describir y analizar situaciones y propiedades particulares de las tareas que conforman un Proceso de Negocio. Para ello proponemos la aplicación de la Lógica Temporal de Tiempo Lineal con Fluentes (FLTL) [37], presentada en la Subsección 2.3.2.

Gracias a la semántica de FLTL, donde la noción de tiempo está presente, esta lógica nos brinda flexibilidad para caracterizar una gran variedad de propiedades sobre los modelos de Procesos de Negocios. Ejemplo de ello puede encontrarse en [75, 89], donde se utiliza este tipo de lógica para especificar modelos basados en restricciones (declarativas) de *Workflows*. En [75], en pos de facilitar la descripción de propiedades al usuario, se propone una herramienta llamada "DECLARE" para la especificación de restricciones. Esta herramienta contiene una serie de *plantillas* que generalizan diferentes propiedades a garantizar sobre un *workflow*. Cada una de ellas tiene asociada su correspondiente especificación mediante una fórmula. Estas especificaciones son utilizadas por la herramienta para verificar que el modelo declarativo es *consistente*. Algunos ejemplos de estas plantillas son las siguientes:

- *Existence(A)*: Especifica que la tarea A es ejecutada al menos una vez. Su especificación es: $\mathbf{F}A$

- *Response(A, B)*: Especifica que si la tarea A es ejecutada, entonces en algún momento se ejecutará la tarea B. Su especificación es: $\mathbf{G}(A \rightarrow \mathbf{F}B)$

- *Precedence(A, B)*: El concepto de precedencia se especifica mediante la fórmula $\neg B\mathbf{W}A$, esto es, la tarea B no puede tener lugar antes que la tarea A.

- *Succession(A, B)*: Exige que B se ejecute al menos una vez y lo haga después de A. Utilizando las propiedades anteriores podemos especificarla como $Response(A, B) \wedge Precedence(A, B)$.

- *Not co-existence(A, B)*: Se especifica como $\neg(\mathbf{F}A \wedge \mathbf{F}B)$, esto es, sólo una de las dos tareas puede tener lugar.

- *1of4(A, B, C, D)*: La plantilla $1 - of - 4$ exige que al menos alguna de las 4 tareas se ejecute al menos una vez: $\mathbf{F}A \vee \mathbf{F}B \vee \mathbf{F}C \vee \mathbf{F}D$.

A diferencia de DECLARE, nuestra propuesta de verificación va más allá de una prueba de consistencia en un modelo declarativo (mediante análisis de satisfactibilidad de un conjutno de restricciones como las definidas anteriormente). El objetivo es verificar propiedades como las anteriores *sobre un modelo procedural*. Como lo anticipamos en la Figura 4.1, una vez traducido el modelo YAWL a FSP, disponemos de la herramienta LTSA para realizar verificaciones sobre el modelo traducido. LTSA ofrece como lenguaje para la especificación de propiedades la Lógica Temporal de Tiempo Lineal extendida con *Fluentes* [62] (ver Subsección 2.3.2). De esta manera, gracias al poder expresivo de esta lógica podemos especificar de manera transparente las plantillas que ofrece DECLARE, ya que la lógica subyacente a dichas plantillas está contenida en la lógica propuesta.

Como podemos apreciar en las plantillas anteriores, las propiedades expresadas predican sobre situaciones donde la ejecución o actividad de las tareas del modelo son los elementos que conforman las fórmulas (proposiciones atómicas). Para capturar ese concepto en los modelos FSP producto de la traducción propuesta, donde los elementos explícitos del modelo son *eventos*, necesitamos identificar el intervalo entre la ocurrencia de los eventos de comienzo (*start*) y fin (*end*) de las tareas. Si bien podemos expresar dicho intervalo por medio de una fórmula lógica, utilizamos la flexibilidad del concepto de *fluente* para describir los estados implícitos correspondientes a las actividades de cada tarea.

4.4.1 Fluentes

Como fue introducido en la Subsección 2.3.2, los fluentes son variables binarias cuyo valor depende de la ocurrencia de eventos, es decir, brindan la posibilidad de representar estados del modelo que no se encuentran explícitamente especificados en el mismo. Por ejemplo, podemos caracterizar el estado de ejecución de una tarea mediante la definición de un fluente que inicialmente es *falso*, y que se activa (se torna verdadero) cuando ocurre el evento de comienzo (*start*) de la misma y se desactiva con el evento de finalización (*end*) correspondiente.

A modo de ejemplo y retomando el *workflow* de la Figura 4.2 cuyo modelo FSP se detalla en la Figura 4.10, para verificar la propiedad que expresa que las tareas t_2 y t_3 son mutuamente excluyentes, primeramente definimos los correspondientes fluentes que caracterizan sus actividades y luego especificamos en LTSA la propiedad mediante una aserción (`assert`), como se muestra a continuación.

```
1 fluent TSK[i:1..4] = <
2        //enabling events
3        t[i].start,
4        //disabling events
```

```
5         t[i].end
6 >
7 assert NO-T2T3 = []!(TSK[2] && TAK[3])
```

En las líneas 1-6 se definen de manera compacta los fluentes de las cuatro tareas que componen el modelo. Luego, la fórmula de la aserción especifica que en ningún momento pueden estar activas las tareas t_2 y t_3 al mismo tiempo, esto es, que la evaluación de sus respectivos fluentes no sea *verdadera* simultáneamente.

Fundamentando el poder expresivo de nuestro enfoque, para caracterizar la propiedades mencionadas anteriormente, a continuación mostramos de manera genérica su especificación como aserciones en LSTA.

```
1 fluent TSK[i:1..N] = <
2         //enabling events
3         t[i].start,
4         //disabling events
5         {t[i].end,t[i].cancel}
6 >
7 assert EXISTENCE(A) = <>TSK[A]
8 assert RESPONSE(A,B) = [](TSK[A] -><>TSK[B])
9 assert PRECEDENCE(A,B) = ((!TSK[A]) W (TSK[B]))
10 assert SUCCESSION(A,B) = (RESPONSE(A,B) && PRECEDENCE(A,B))
11 assert NO_CO_EXISTENCE(A,B) = !(<>TSK[A] && <>TSK[B])
12 assert ONE_OF_FOUR(A,B,C,D) = <>TSK[A] || <>TSK[B] || <>TSK[C] || <>TSK[D]
```

Mas allá de la posibilidad de caracterizar las plantillas mencionadas, la flexibilidad que aporta la definición de fluentes, puede reflejarse en su sencillez para establecer estados *monitores* que no forman parte del modelo. Así, por ejemplo, podemos utilizarlos para conocer el estado de actividad de una tarea compuesta simplemente con definir un fluente que se active con el evento con el cual comienza la misma `i_cond` y retorne a su valor falso cuando tenga lugar el evento de finalización `o_cond` de dicha tarea compuesta. Los fluentes también ayudan a caracterizar trazas de ejecución delimitadas por uno o más eventos del modelo. Además, estos eventos no tienen que estar circunscriptos necesariamente a un mismo elemento del modelo. Por ejemplo, se puede definir un fluente que se active con eventos que pertenecen a diferentes sub-redes. Más adelante, en nuestro caso de estudio (ver Sección 4.6), ensayaremos estas propuestas en propiedades particulares del mismo.

4.5 Automatización de la codificación y asistente para el análisis de *Workflows*.

Como parte de este trabajo se implementaron dos herramientas, una para la traducción automática de modelos YAWL a FSP y otra para asistir al usuario en el análisis de modelos de Workflows.

Una de las herramientas, llamada "YAWL2FSP"[3], toma archivos que contienen la especificación de un modelo YAWL (con formato XML) y produce como salida un archivo con la correspondiente especificación FSP. La herramienta fue desarrollada en el lenguaje de programación Java y se trata de una aplicación con interfaz de línea de comandos (no tiene interfaz gráfica).

La segunda herramienta complementa al traductor, brindando al usuario una interfaz gráfica que le permite visualizar un modelo YAWL para su análisis. En la Figura 4.5 se muestra una captura de pantalla de la misma. Su objetivo es asistir al usuario en la formulación de las propiedades a analizar, no sólo mediante plantillas de operadores de la lógica subyacente, sino también brindando una forma simple e intuitiva de especificar situaciones mediante la definición de fluentes. Una vez establecidas las propiedades sobre el modelo, esta herramienta invoca al traductor y posteriormente a LTSA para su verificación.

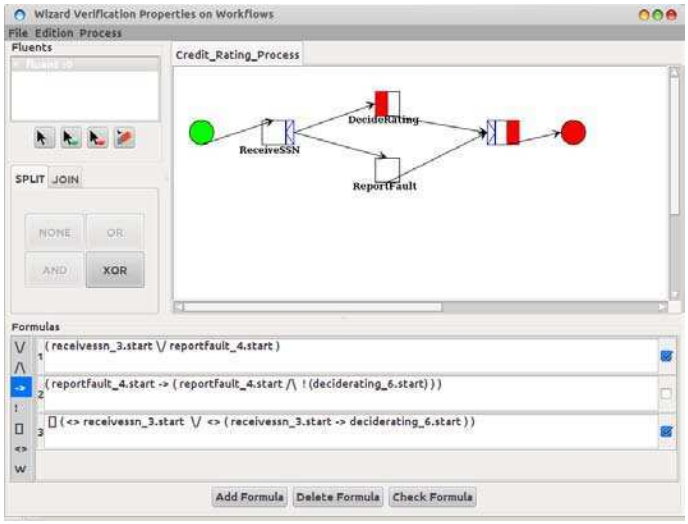


Figura 4.12: Captura de pantalla de la herramienta FLYAnalyser.

[3]Puede descargarse de http://sourceforge.net/projects/yawl2fsp/

4.6 Caso de Estudio

En pos de validar el trabajo, aplicamos nuestra técnica a un caso de estudio. Por su complejidad, el mismo fue adoptado del conjunto de casos presentados con la herramienta YAWL [4]. El caso de estudio describe el proceso que sigue una compañía ficticia para cumplir con un pedido de compra. Este proceso está dividido en tres grandes etapas: solicitud, logística (que incluye la designación del transporte, paquete en tránsito y paquete en destino) y pago. El proceso general, cuyo modelo YAWL se muestra en la Figura 4.13, presenta las etapas descriptas anteriormente como tareas compuestas.

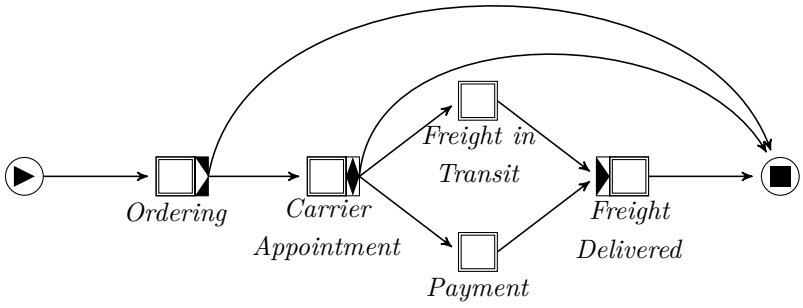


Figura 4.13: Red principal del Caso de Estudio "*Order Fulfillment*"

A continuación se describen brevemente las tareas que componen el modelo:

- *Ordering*: Esta tarea compuesta modela cómo se compra un producto. Se crea una orden de compra, y luego de su aprobación, puede ocurrir que se desee modificarla (volviendo a requerirse su aprobación), o que se confirme, finalizando esta etapa.

- *Carrier Appointment*: Una vez realizada la orden del producto (en la etapa anterior), se debe designar el modo de transporte del paquete. Dentro de esta tarea se clasifican los pedidos de acuerdo al espacio requerido durante su transporte.

- *Freight in Transit*: En esta etapa se describe el viaje de un pedido. Durante el viaje ocurren paralelamente las emisiones de notificaciones de seguimiento y la consulta del estado de tránsito.

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

- *Payment*: Describe el proceso de pago por el pedido realizado.
- *Freight Delivered*: Se modela el arribo del pedido, contemplando el hecho de que pueda ocurrir una posible devolución de la mercadería y el registro de un reclamo por daño o pérdida.

El modelo YAWL correspondiente a la tarea compuesta *Carrier Appointment*, que se muestra en la Figura 4.14, fundamenta en parte la elección debido a la diversidad de componentes que se encuentran en él. Básicamente el modelo especifica que, luego de que se ha confirmado la Orden de Pedido en la etapa previa, se necesita preparar la guía de transporte correspondiente y estimar el modo de transporte. Luego se establece el costo del flete y el tipo de transporte requerido: un camión completo, un camión parcialmente completo, o un paquete. Las gestiones correspondientes a los primeros se sintetizan gráficamente como una tarea compuesta que especifica su comportamiento. Finalmente luego de posibilitar el cambio de información sobre el origen y destino de la carga, la misma está lista para su transporte.

A continuación se muestra la conformación (elementos del lenguaje) de las diferentes redes que forman parte de la especificación YAWL correspondiente al caso de estudio.

Red	Tareas	Condiciones	Compuertas	Tareas Comp.
Overall (Principal)	0	2	3	5
Ordering	5	3	2	0
Carrier Appointment	13	8	4	2
Full Truck Load	8	4	3	0
Less Than Truck Load	5	5	3	0
Freight in Transit	5	4	3	0
Payment	14	2	10	1
Freight in Delivered	5	2	2	2
Return Management	1	2	0	0
Loss or Damage Manag.	1	2	0	0
Process Freight Payment	1	2	0	0

La siguente tabla muestra la cantidad de elementos de cada una de las redes teniendo en cuenta la descomposición de las tareas compuestas.

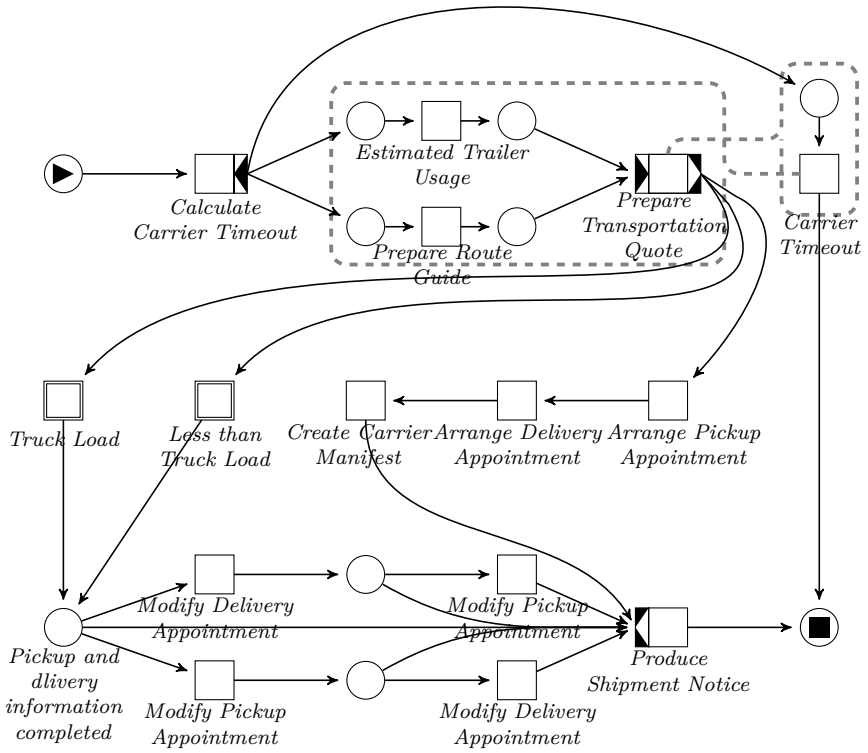


Figura 4.14: Modelo YAWL de la tarea compuesta "*Carrier Appointment*"

Red	Tareas	Condiciones	Compuertas
Overall (Principal)	58	36	30
Ordering	5	3	2
Carrier Appointment	26	17	10
Full Truck Load	8	4	3
Less Than Truck Load	5	5	3
Freight in Transit	5	4	3
Payment	15	4	10
Freight in Delivered	7	6	2
Return Management	1	2	0
Loss or Damage Manag.	1	2	0
Process Freight Payment	1	2	0

En la siguiente tabla se muestran los tamaños y recursos correspondientes a los Sistemas de Transición de estados generados a partir de la codificación de cada una

de las redes como procesos FSP.

Red	Estados	Transiciones	Memoria(Kb)	Tiempo(ms)
Overall (Principal)	13164	59722	28969	298
Ordering	13	16	11648	106
Carrier Appointment	224	430	11969	127
Full Truck Load	34	56	11693	115
Less Than Truck Load	20	30	11663	118
Freight in Transit	35	72	11668	108
Payment	175	376	11686	119
Freight in Delivered	20	24	11484	117
Return Management	4	4	11598	117
Loss or Damage Manag.	4	4	11598	117
Process Freight Payment	4	4	11598	117

4.6.1 Propiedades analizadas

Siguiendo con el caso de estudio, a continuación se enumeran algunas de las propiedades analizadas. El objetivo de las propiedades seleccionadas no es sólo el de ensayar la factibilidad del análisis propuesto por nuestro enfoque, sino también (y principalmente en este capítulo) exhibir la flexibilidad que aporta la presencia de *fluentes* en la formulación de dichas propiedades. Las propiedades analizan situaciones de la tarea compuesta *Carrier Appointment* del caso de estudio, y para cada una de ellas se propone la información a verificar y la expresión en FSP que la especifica, incluyendo la definición de los fluentes que figuran en la misma. En cada caso, el lector podrá observar la relación entre el nombre genérico que asigna el proceso de traducción a las tareas y el nombre de la misma en el proceso por medio de comentarios sobre el código FSP correspondiente.

- Si se supera el tiempo de espera (se cancela el pedido), no puede emitirse ningún aviso de envío de mercadería. En el modelo, si ocurre *timeout* en *Carrier Appointment*, entonces no puede tener lugar *Produce Shipment Notice*.

```
1 fluent CarrierTimeout = <
2  {// Initiating actions
3   carrier_appointment.task[5].start //task[5]=Carrier Timeout
4  },
5  {// Terminating actions
6   carrier_appointment.task[5].end, //task[5]=Carrier Timeout
7   carrier_appointment.task[5].cancel // task[5]=Carrier Timeout
8  }
```

```
9  >
10
11 fluent ProduceShipmentNotice = <
12  {// Initiating actions
13   carrier_appointment.task[13].start //task[13]=Produce Shipment Notice
14  },
15  {// Terminating actions
16   carrier_appointment.task[13].end //task[13]=Produce Shipment Notice
17  }
18 >
19
20 assert PROPERTY_1 = [](CarrierTimeout ->!ProduceShipmentNotice)
```

En la especificación de esta propiedad se puede observar la utilización de fluentes para caracterizar la actividad de las tareas `Carrier Timeout` y `Produce Shipment Notice`.

- Las tareas dentro de las diferentes clasificaciones de transporte son mutuamente excluyentes.

```
1  fluent FullTruckLoad = <
2   {// Initiating actions
3    carrier_appointment.ftl.i_cond
4   },
5   {// Terminating actions
6    carrier_appointment.ftl.o_cond
7   }
8  >
9
10 fluent LessThanTruckLoad = <
11  {// Initiating actions
12   carrier_appointment.ltl.i_cond
13  },
14  {// Terminating actions
15   carrier_appointment.ltl.o_cond
16  }
17 >
18
19 fluent SinglePackage = <
20  {// Initiating actions
21   carrier_appointment.task[6].start //task[6]=Arrange Pickup Appointment
22  },
```

```
23 {// Terminating actions
24  carrier_appointment.task[8].end //task[8]=Create Carrier Manifest
25 }
26 >
27
28 assert PROPERTY_2 = !((FullTruckLoad && LessThanTruckLoad) ||
29                       (FullTruckLoad && SinglePackage) ||
30                       (SinglePackage && LessThanTruckLoad))
```

En la especificación de esta propiedad se puede observar la versatilidad de fluentes para caracterizar la actividad de trazas del proceso definidas por eventos. En este caso las actividades involucradas son las de las tareas compuestas `LessThanTruckLoad` y `FullTruckLoad`, como la traza que representa un envío simple que comienza en `Arrange Pickup Appointment` y finaliza en `Create Carrier Manifest`.

- Sólo se puede producir una notificación de envío si previamente se ha establecido el origen y destino de la mercadería. En el modelo, si se ejecutó *Produce Shipment Notice*, necesariamente se ejecutaron *arrange delivery appointment* y *arrange pickup appointment*, en alguno de los posibles tipos de transportes.

```
1 fluent ProduceShipmentNotice = <
2 {// Initiating actions
3  carrier_appointment.task[13].start //task[13]=Produce Shipment Notice
4 },
5 {// Terminating actions
6  carrier_appointment.task[13].end //task[13]=Produce Shipment Notice
7 }
8 >
9
10 fluent DeliveryAppointmentArranged = <
11 {// Initiating actions
12  carrier_appointment.task[7].end, //task[7]=Arrange Delivery App.
13  carrier_appointment.ftl.task[2].end,//ftl.task[2]=Arrange Delivery App.
14  carrier_appointment.ftl.task[3].end,//ftl.task[3]=Arrange Delivery App.
15  carrier_appointment.ftl.task[7].end,//ftl.task[7]=Arrange Delivery App.
16  carrier_appointment.ltl.task[2].end,//ltl.task[2]=Arrange Delivery App.
17  carrier_appointment.ltl.task[5].end //ltl.task[5]=Arrange Delivery App.
18 },
19 {// Terminating actions
20  carrier_appointment.o_cond
21 }
```

```
22 >
23 fluent PickupAppointmentArranged = <
24  {// Initiating actions
25   carrier_appointment.task[6].end, //task[6]=Arrange Pickup App.
26   carrier_appointment.ftl.task[1].end,//ftl.task[1]=Arrange Pickup App.
27   carrier_appointment.ftl.task[4].end,//ftl.task[4]=Arrange Pickup App.
28   carrier_appointment.ftl.task[6].end,//ftl.task[6]=Arrange Pickup App.
29   carrier_appointment.ltl.task[3].end //ftl.task[3]=Arrange Pickup App.
30  },
31  {// Terminating actions
32   carrier_appointment.o_cond
33  }
34 >
35
36 assert PROPERTY_3 = []( ShipmentNoticeProduced ->
37        (DeliveryAppointmentArranged && PickupAppointmentArranged))
```

Para realizar este análisis utilizamos la flexibilidad de los fluentes para describir monitores de eventos. En particular, además de caracterizar la actividad de la notificación de envío `ProduceShipmentNotice`, se definieron dos fluentes que monitorean si se establecieron el origen y destino para cualquier tipo de transporte que corresponda. Cabe notar que estos fluentes se activan por eventos que pueden estar establecidos en diferentes redes del modelo, particularmente tanto `DeliveryAppointmentArranged` como `PickupAppointmentArranged` se activan por eventos de las redes `carrier_appointment`, `ftl` (Full Track Load) y `ltl` (Less than Track Load).

Los recursos necesarios para su verificación se detallan en la siguiente tabla:

Propiedad	Tiempo (ms)	Memoria (Mb)
PROPERTY_1	154	11.8
PROPERTY_2	152	11.9
PROPERTY_3	185	16.6

Los procesos de traducción y verificación se realizaron en una computadora personal con un procesador Intel Core 2 Duo a 2.2 Ghz, 2 Gb. 667 Mhz DDR2 SDRAM de memoria bajo un sistema operativo basado en Unix.

4.7 Otras Aplicaciones de Especificaciones de Propiedades Temporales para la Construcción de Workflows

La construcción automática de modelos a partir de especificaciones ha sido objeto de numerosas investigaciones, [22, 66] para citar algunas. En los últimos años, una de las técnicas que ha tomado relevancia por sus avances [76] y su aplicación a la ingeniería de software [93], es la *síntesis* de sistemas de eventos discretos a partir de propiedades. Como complemento al análisis de *workflows* presentado en este capítulo mediante propiedades temporales, proponemos la aplicación de la técnica síntesis para generar instancias de *workflows* a partir de restricciones expresadas como fórmulas de la lógica FLTL. Particularmente, acotamos nuestro trabajo a propiedades de *safety*.

La especificación declarativa de *workflows* ha sido considerada en los últimos años como una alternativa o complemento a las clásicas especificaciones procedurales de los mismos [32, 91]. De hecho en [107] se presenta una herramienta de software que permite describir escenarios posibles de *workflows* por medio de restricciones entre las componentes del mismo, para luego hacer un análisis de consistencia, entre otros.

Nuestro proceso consiste en: dado un conjunto de recursos (actividades) y una serie de propiedades declarativas (restricciones) sobre ellas, poder construir, de manera automática, un *workflow* procedural en algún lenguaje, como por ejemplo YAWL. Este *workflow* estará conformado por las actividades del conjunto y el flujo de control (conexiones) entre las mismas. La configuración del flujo de control no sólo deberá respetar las restricciones, sino que también maximizará la independencia entre las actividades.

Este proceso de construcción se basa en la mirada abstracta de un *workflow* detallada en la Sección 4.3, donde cada tarea se caracteriza por sus eventos de comienzo y terminación. Bajo ese concepto, se genera un sistema de transición de estados que representa *todas* las posibles ejecuciones de las tareas (recursos) que componen el *workflow*. Luego, mediante la herramienta LTSA, se *podan* del LTS anterior todas aquellas transiciones que compongan trazas que produzcan violaciones de las restricciones especificadas. Luego, procesar el LTS refinado para obtener algún *workflow* que el mismo caracteriza.

4.7.1 Esquema del proceso

En la Figura 4.15 mostramos el esquema propuesto para este proceso. El mismo comienza con la especificación de los recursos de que se dispone, es decir, la cantidad

de tareas que van a componer el *workflow*, y las restricciones que se exigen sobre ellas. Focalizamos nuestro interés en propiedades de *safety*, ya que son el tipo de propiedades que soporta LTSA para realizar el proceso de refinamiento y además, con ellas podemos caracterizar un amplio espectro de restricciones sobre comportamientos de *workflows* [107]. Una vez que obtenemos el LTS que modela las posibles trazas de ejecución de tareas que respetan la propiedades, generamos el workflow correspondiente.

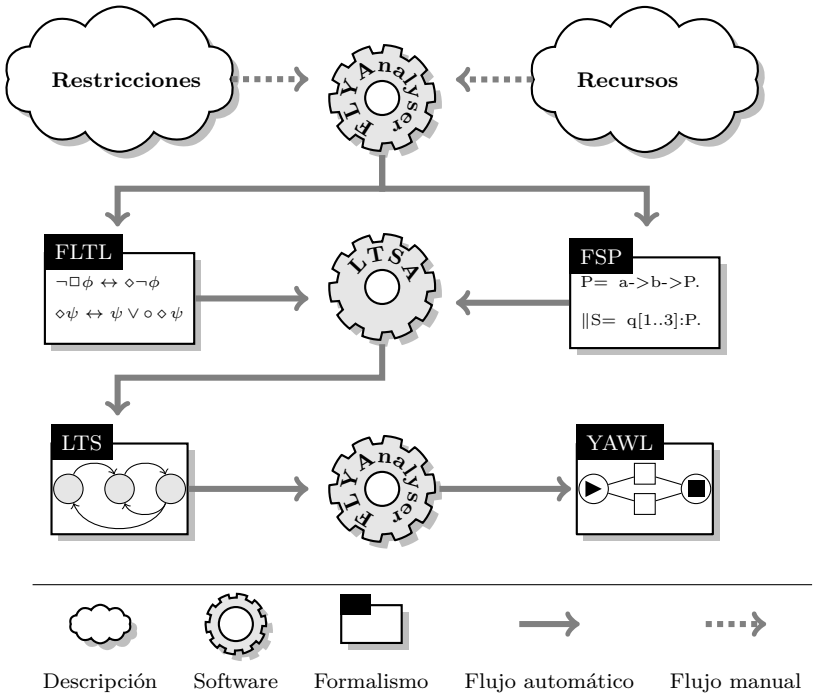


Figura 4.15: Esquema general del proceso de construcción de Workflows

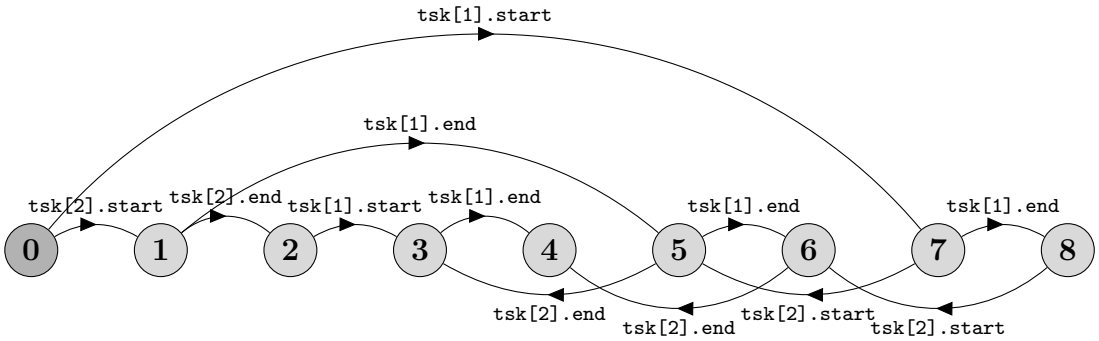
Cabe destacar que para la especificación de propiedades podemos utilizar fluentes para describir estados abstractos del *workflow* que formen parte de las mismas.

Como un ejemplo simple, supongamos que tenemos como recursos dos tareas `tsk[1]` y `tsk[2]` sin ningún tipo de restricciones. Continuando con la descripciones detalladas en el Capítulo 4, para caracterizar el comportamiento (posibles ejecuciones) del sistema, generamos la siguiente especificación FSP:

```
1 const T    = 2     // number of tasks
2 range Tasks = 1..T
3
```

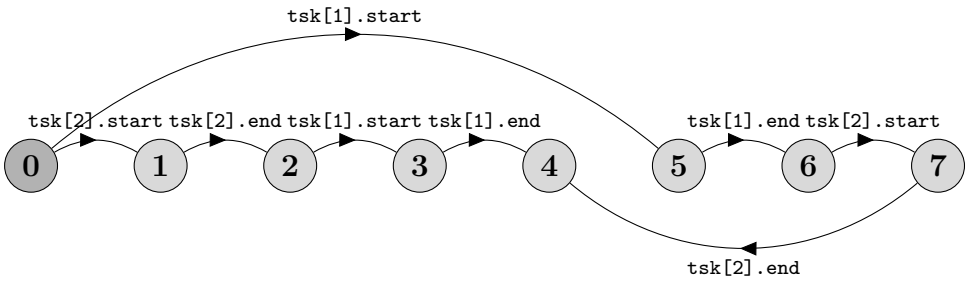
```
4 TASK = (start ->end -> STOP).
5 || SYSTEM =( tks[i:Tasks]:TASK).
```

A continuación se muestra el sistema de transición de estados generado por esta especificación. El mismo consiste en el *interleaving* de los eventos `start` y `end` correspondientes a las tareas. Como puede observarse, la complejidad del LTS es exponencial con respecto a la cantidad de tareas.



Mediante diferentes técnicas que ofrece LTSA, podemos refinar el modelo incorporando restricciones sobre el comportamiento del mismo. Por ejemplo, para imponer una restricción de exclusión mutua entre las dos tareas, podemos hacerlo capturando la actividad de cada una de ellas mediante fluentes y luego, especificar una restricción (`constraint`) sobre la validez simultánea de ambos fluentes. A continuación mostramos el modelo FSP y el LTS correspondiente al modelo refinado.

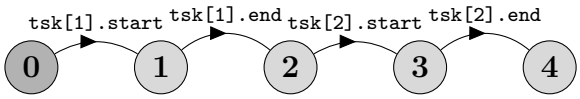
```
1 ...
2 fluent FTks[i:Tasks] = <tks[i].start, tks[i].end>
3 constraint C1 = [] !(FTks[1] && FTks[2]) //mutual exclusion
4
5 || SYSTEM =( tks[i:Tasks]:TASK || C1).
```



Cabe destacar que LTSA sólo permite fórmulas que expresan propiedades de *safety* para refinar modelos mediante `constraint`. Otra técnica que podemos utilizar para refinar el modelo, por ejemplo para imponer orden (precedencia) de ejecución, es la definición de un proceso que, en su composición con el sistema, restrinja el modelo del mismo. A continuación aplicamos esta idea para imponer precedencia de `tsk[1]` sobre `tsk[2]` en modelo anterior.

```
...
PC2 = (tks[1].start ->tks[2].start ->PC2).

|| SYSTEM =( tks[i:Tasks]:TASK || C1 || PC2).
```



Otro tipo de restricciones que podemos utilizar para refinar, es la elección, es decir, que la ejecución de una tarea, excluye a que otra tenga lugar a partir de ese momento. Esta restricción puede capturarse mediante el siguiente `constraint`:

```
constraint C2 = [] (FTks[1] ->! <> FTks[2])
```

A partir del sistema de transición de estados refinado, comienza el proceso de generación de posibles *workflows*. Para ello, una forma de hacerlo es mediante la realización consultas sobre la relación entre la tareas mediante la definición y verificación de propiedades en LTSA, conformando una matriz de relaciones entre las mismas. Las relaciones analizadas son: Precedencia, Posterior y Exclusión mutua.

4.8 Counting Fluent Linear Time Temporal Logic

En las secciones anteriores mostramos las ventajas que ofrece FLTL para la descripción de propiedades de workflows. En particular, cómo los *fluentes* nos permiten caracterizar estados abstractos de los modelos de *workflows*, los cuales pueden utilizarse como parte de las fórmulas que describen dichas propiedades. Como lo expresa la lógica FLTL (ver Subsección 2.3.2), los fluentes son variables binarias cuyo valor está asociado a la ocurrencia de eventos agrupados en dos conjuntos disjuntos que los definen. Un conjunto de eventos que modifican el valor del fluente como *verdadero* y un conjunto que lo vuelven *falso*.

Si bien los fluentes ofrecen una flexibilidad importante a la hora de caracterizar estados abstractos en modelos de sistemas reactivos, existen situaciones en los mismos donde pese a su poder expresivo, especificar ciertas propiedades se torna tedioso

y en ocasiones inmanejable. Más aún, existen situaciones donde la especificación de ciertas propiedades requiere la modificación del modelo generando instrumentos que permitan expresarla. Para ilustrar esta afirmación y extrapolando nuestro interés a descripciones de sistemas reactivos en general, consideremos el siguiente ejemplo simple de un cajero automático (ATM) especificado en FSP.

```
ATM = (insertCard ->E_BadBankPwd[0]),
E_BadBankPwd[i:0..2]=(requestPwd ->(cancel ->Finish
                                    | enterPwd -> verifyPwd -> Verify[i])),
Verify[i:0..2] = ( correctPwd ->Extraction
                 | badPwd -> E_BadBankPwd[i+1]
                 | cancel -> Finish),
E_BadBankPwd[3] = (lockAcct ->Locked),
Finish = (ejectCard ->requestTakeCard ->takeCard -> ATM),
Locked = (unlockAcct ->ATM),
Extraction = (money ->getMoney -> Finish).
```

Inicialmente el cajero requiere al usuario que inserte su tarjeta y que ingrese su contraseña (Pwd). Una vez verificada la contraseña, si la misma es correcta, el usuario puede extraer el dinero y quitar la tarjeta. Si la contraseña es incorrecta, el cajero comienza a contar la cantidad de veces que el usuario la ingresa mal. En caso de que el usuario ingrese mal su contraseña en tres ocasiones consecutivas, el cajero bloqueará la tarjeta para prevenir la extracción de dinero con dicha tarjeta. Cabe destacar que el usuario puede cancelar la operación (antes de tres ingresos fallidos) y retirar la tarjeta, evitando el bloqueo de la misma.

Con el fin de verificar que el cajero implementa correctamente la política de seguridad, podemos especificar las siguientes propiedades:

$$P1 = \neg\langle\rangle(Fl_{\texttt{badPwd}} \wedge \mathbf{X}(BadPass\ \mathbf{U}\ (Fl_{\texttt{badPwd}}\ \wedge \mathbf{X}(BadPass\ \mathbf{U}\ (Fl_{\texttt{badPwd}} \wedge \mathbf{X}(BadPass\ \mathbf{U}\ Fl_{\texttt{money}}))))))$$

$$P2 = \texttt{[]}\neg(\neg BadPass\ \mathbf{U}\ LOCKED) \wedge \texttt{[]}\neg(\neg BadPass\ \mathbf{U}\ (Fl_{\texttt{badPwd}} \wedge \mathbf{X}(\neg Fl_{\texttt{badPwd}}\ \mathbf{U}\ LOCKED)))$$
$$\wedge\ \texttt{[]}\neg(\neg BadPass\ \mathbf{U}\ (Fl_{\texttt{badPwd}} \wedge \mathbf{X}(\neg Fl_{\texttt{badPwd}}\ \mathbf{U}\ (Fl_{\texttt{badPwd}} \wedge \mathbf{X}(\neg Fl_{\texttt{badPwd}}\ \mathbf{U}\ LOCKED)))))$$

Donde $BadPass$ es un fluente definido por los siguientes conjuntos de eventos $\langle\{\texttt{badPwd}\}, \{\texttt{correctPwd}, \texttt{unlockAcct}\}\rangle$, $Fl_{\texttt{badPwd}}$ denota el fluente asociado al evento `badPwd` y $LOCKED \equiv \langle\texttt{lockAcct}, \texttt{unlockAcct}\rangle$ captura el estado de bloqueo de la tarjeta. Intuitivamente la primera propiedad expresa que no puede suceder que luego de tres ingresos erróneos consecutivos de la contraseña, el usuario pueda extraer dinero del cajero. La segunda propiedad especifica que la tarjeta solo puede estar bloqueada si se ha superado el límite de errores en el ingreso de la contraseña.

El lector puede observar, que pese a que el poder expresivo de FLTL permite especificar estas propiedades, la especificación no es simple. El problema que podemos observar en este tipo de propiedades es el anidamiento del operadores temporales necesario para poder capturar la nociones de *veces*.

Con el objetivo de contribuir a la simplificación de la especificación de este tipo de propiedades, proponemos una extensión de FLTL, complementando la definición de fluentes (proposicionales) con la definición de *fluentes contadores*. Estos fluentes contadores pueden interpretarse como variables enteras y su semántica está dada por tres conjuntos, un conjunto de eventos que *incrementan* el valor del fluente, un conjunto de eventos que *disminuyen* su valor y un conjunto de eventos que *restablecen* el mismo a su valor inicial.

A partir de esta especificación de fluentes, extendemos la sintaxis de las fórmulas, permitiendo expresiones aritméticas entre los fluentes, por ejemplo, consultar sobre relación de orden entre valores de fluentes o constantes: `F = c`, `F1 <= F2`. Retomando el ejemplo propuesto, y considerando la definición del fluente contador *ERRORS* inicializado en 0, incrementado por el evento `badPwd` y reiniciado por el evento `correctPwd`, podemos expresar las propiedades *P1* y *P2* de manera natural como se muestra a continuación:

$$P1 = \neg\langle\rangle(ERRORS \geq 3 \wedge Fl_{\texttt{money}}) \qquad P2 = \texttt{[]}(LOCKED \rightarrow ERRORS \geq 3)$$

Si bien esta extensión merece un análisis en cuanto su complejidad y poder expresivo, estamos particularmente interesados en su aplicación a model checking, lo cual impone un límite en la cantidad de valores que los fluentes pueden tomar. Focalizamos nuestro trabajo a poder utilizar esta extensión en el análisis de modelos de sistemas reactivos incluyendo de manera directa los análisis presentados anteriormente en este capítulo.

A continuación presentamos la sintaxis para esta lógica que la denominamos Counting Fluent Linear-time Termporal Logic (CFLTL), su semántica y una interpretación de los fluentes contadores como autómatas de Büchi. Finalmente, con el fin de utilizarla como soporte para la especificación y verificación de propiedades, bajo la imposición de ciertos límites para el rango de los valores de los fluentes contadores, presentamos una reducción de CFLTL a FLTL para implementar su aplicación al análisis de sistemas reactivos mediante LTSA.

Definición 4.2. ***Fluente Contador***

Definimos un fluente contador mediante tres conjuntos de acciones que caracterizan su valor:

$$cFl = \langle I_{cFl}, D_{cFl}, R_{cFl} \rangle initially\ N$$

Donde I_{cFl} es el conjunto de acciones que incrementa el valor del fluente, D_{cFl} es el conjunto de acciones cuya ocurrencia disminuye el valor del fluente y R_{cFl} es el conjunto de acciones que al tener lugar, restablecen su valor inicial N. Además $I_{cFl} \cap D_{cFl} \cap R_{cFl} = \emptyset$.

La interpretación del conjunto de fluentes contadores Φ, es una secuencia infinita sobre $\mathbb{N}^{\Phi}$, la cual asigna a cada instante de la ejecución del sistema el valor de cada fluente contador. De esta manera, una secuencia infinita de acciones $w = a_1, a_2, \ldots$ que caracteriza una traza de ejecución de un sistema, define una interpretación del conjunto de fluentes Φ en un instante i como $\mathcal{V}_{i,w}(cFl \in \Phi)$ sobre $\mathbb{N}^{\Phi}$ de la siguiente manera:

$$\mathcal{V}_{i,w}(cFl) = \begin{cases} N_{cFl} \text{ si } i = 0 \\ N_{cFl} + (\#_{r \leq k \leq i} a_i \in I_{cFl}) - (\#_{r \leq k \leq i} a_i \in D_{cFl}) \text{ si } i > 0 \end{cases}$$

donde $r = \text{Max } l, 0 \leq l \leq i : a_l \in R_{cFl}$ o 0 si $\forall l \in \mathbb{N}, 0 \leq l \leq i : a_l \notin R_{cFl}$.

Dada su naturaleza numérica, el conjunto de expresiones sobre Φ que extienden el alfabeto de FLTL, se forman de la siguiente manera:

$$\phi ::= cFl \sim c \mid cFl_1 \sim cFl_2 \mid cFl_1 \sim cFl_2 + c \mid cFl_1 \sim cFl_2 - c, \text{ donde } \sim \in \{=, >, <\}$$

Extendemos la interpretación de las fórmulas de FLTL definiendo como proposiciones el conjunto de expresiones sobre Φ con la siguiente semántica:

- $w, i \models cFl \sim c$ *sii* $V_{i,w}(cFl) \sim c$
- $w, i \models cFl_1 \sim cFl_2$ *sii* $V_{i,w}(cFl_1) \sim V_{i,w}(cFl_2)$
- $w, i \models cFl_1 \sim cFl_2 + c$ *sii* $V_{i,w}(cFl_1) \sim V_{i,w}(cFl_2) + c$
- $w, i \models cFl_1 \sim cFl_2 - c$ *sii* $V_{i,w}(cFl_1) \sim V_{i,w}(cFl_2) - c$

donde $\sim \in \{=, >, <\}$.

4.8.1 Representación como Autómatas de Büchi

Como fue introducido en la Subsección 2.4.1, para verificar una propiedad P en un modelo M, siguiendo la técnica presentada en [100], en términos generales se intersecan los autómatas de Büchi correspondientes a M y el correspondiente a la negación de fórmula que especifica la propiedad P. Luego la verificación de la propiedad se reduce a comprobar si el lenguaje que acepta este nuevo autómata es vacío.

De manera consistente con la propuesta presentada en [37] en la cual se extiende LTSA con soporte para fluentes binarios, necesitamos expresar las expresiones que involucran fluentes contadores como autómatas de Büchi. En este sentido consideramos dos grupos de expresiones, aquellas unitarias (por ejemplo `CFL = c`), donde básicamente se expresa el valor del fluente, y las binarias, que expresan una relación entre dos fluentes contadores.

Sea $cFl = I_{cFl}, D_{cFl}, R_{cFl}$. Para las expresiones de la forma $cFl \sim c$, limitados por una cota K, construímos el siguiente autómata de Büchi:

$B_{cFl} = \langle Q, A, \delta, q_I, F \rangle$ donde

- $Q = \{q_{-\infty}, q_{-k}, q_{-k+1}, \ldots, q_0, \ldots, q_{k-1}, q_k, q_\infty\}$
- $A = I_{cFl} \cup D_{cFl} \cup R_{cFl} \cup Act_{\overline{cFl}}$
- $q_I = q_N$
- La función de transición se conforma de la siguiente manera:

$$\begin{array}{rl}
\delta = & \{\forall a \in I_{cFl}, -k \leq i \leq k-1 \mid (q_i, a, q_{i+1})\} \\
\cup & \{\forall a \in I_{cFl} \mid (q_k, a, q_{+\infty}), (q_{+\infty}, a, q_{+\infty}), (q_{-\infty}, a, q_{-\infty})\} \\
\cup & \{\forall a \in D_{cFl}, -k+1 \leq i \leq k \mid (q_i, a, q_{i-1})\} \\
\cup & \{\forall a \in D_{cFl} \mid (q_{-k}, a, q_{-\infty}), (q_{-\infty}, a, q_{-\infty}), (q_{+\infty}, a, q_{+\infty})\} \\
\cup & \{\forall a \in R_{cFl}, -k \leq i \leq k \mid (q_i, a, q_N)\} \\
\cup & \{\forall a \in R_{cFl} \mid (q_{-\infty}, a, q_{-\infty}), (q_{+\infty}, a, q_{+\infty})\} \\
\cup & \{\forall a \in Act_{\overline{cFl}}, -k \leq i \leq k-1 \vee i = -\infty \vee i = +\infty \mid (q_i, a, q_i)\}
\end{array}$$

- Los estados de que capturan la validez de las expresiones según la relación $\sim c$ son:

$$\begin{array}{lcl}
cFl = c & \rightarrow & F = \{q_c\} \\
cFl \neq c & \rightarrow & F = \{q_i \in Q : (-k \leq i \leq k) \wedge (i \neq c)\} \\
cFl > c & \rightarrow & F = \{q_i \in Q : (-k \leq i \leq k) \wedge (i > c)\} \\
cFl \geq c & \rightarrow & F = \{q_i \in Q : (-k \leq i \leq k) \wedge (i \geq c)\} \\
cFl < c & \rightarrow & F = \{q_i \in Q : (-k \leq i \leq k) \wedge (i < c)\} \\
cFl \leq c & \rightarrow & F = \{q_i \in Q : (-k \leq i \leq k) \wedge (i \leq c)\}
\end{array}$$

El autómata de Büchi B_{cFl} contiene tantos *estados* como valores posibles (acotados) pueda tomar el fluente mas dos estados que caracterizan los estados en los cuales el fluente ha superado alguno de sus límites, superior o inferior. El *alfabeto* del mismo se compone por los eventos que incrementan, disminuyen o reinician el fluente, junto con el resto de eventos del sistema que no alteran su valor. El *estado inicial* es el que tiene asociado el valor inicial especificado en la definición del fluente. El valor inicial predeterminado, en caso de no tener explícitamente determinado uno, es 0, y el estado inicial correspondiente q_0.

La *función de transición* contiene arcos de incremento y disminución del valor asociado al fluente por los correspondientes eventos. En caso de alcanzar los estados límites $-\infty$ o $+\infty$, los arcos asociados a eventos de incremento o disminución, mantienen como destino al mismo estado de origen. Por cada evento en el conjunto reinicio, un arco con destino el estado inicial. Finalmente, por cada evento del sistema que no altera el valor del fluente, el autómata contiene un arco por cada estado que caracteriza dicha situación.

Los *estados de aceptación* se configuran de acuerdo a la semántica de la expresión numérica de la fórmula. Por ejemplo, en caso de igualdad a un valor específico c, se configura como estado de aceptación aquel estado que tiene asociado dicho valor (q_c).

A continuación, en la Figura 4.16, ilustramos el autómata para la expresión $cFl \geq 1$ fluente contador definido por $cFl = I_{cFl}, D_{cFl}, R_{cFl}$ con valor inicial 0 y cota k.

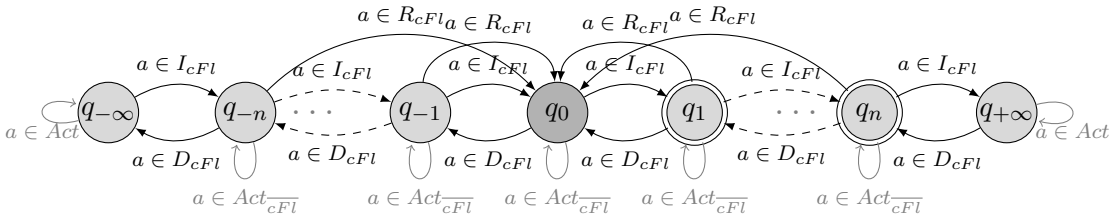


Figura 4.16: Autómata de Büchi que representa la relación $cFl_1 \geq 1$

Para las expresiones de la forma $cFl_1 \sim cFl_2$, la propuesta es construir un autómata que caracteriza la diferencia entre sus valores. Cabe aclarar que la misma asume que los conjunto de reinicio de ambos fluentes coinciden. Sean $cFl_1 = I_{cFl_1}, D_{cFl_1}, R_{cFl}$ y $cFl_2 = I_{cFl_2}, D_{cFl_2}, R_{cFl}$, limitados por una cota K, y con valores iniciales N_{cFl_1} y N_{cFl_2} respectivamente. Construimos el siguiente autómata de Büchi:

$F_{cFl_{1-2}} = \langle Q, A, \delta, q_I, F \rangle$ donde

- $Q = \{q_{-\infty}, q_{-k}, q_{-k+1}, \ldots, q_0, \ldots, q_{k-1}, q_k, q_\infty\}$

- $A = I_{cFl_1} \cup D_{cFl_1} \cup I_{cFl_2} \cup D_{cFl_2} \cup R_{cFl} \cup Act_{\overline{cFl_{1-2}}}$, donde cFl_{1-2} es el conjunto formado por todos los eventos que especifican el valor de cFl_1 y cFl_2.

- $q_I = q_{N_{cFl_1} - N_{cFl_2}}$

- La función de transición se conforma de la siguiente manera:
  Para simplificar la especificaciones consideramos los siguientes conjuntos de eventos:

  - $I_{cFl_{1-2}} = I_{cFl_1} - I_{cFl_2} \cup D_{cFl_2} - D_{cFl_1}$: el conjunto de eventos que incrementan la diferencia $cFl_1 - cFl_2$, son aquellos que sólo incrementan cFl_1 o que sólo disminuyen cFl_2.
  - $D_{cFl_{1-2}} = D_{cFl_1} - D_{cFl_2} \cup I_{cFl_2} - I_{cFl_1}$: el conjunto de eventos que disminuyen la diferencia $cFl_1 - cFl_2$, son aquellos que sólo incrementan cFl_2 o que sólo disminuyen cFl_1.
  - $E_{cFl_{1-2}} = (I_{cFl_1} \cap I_{cFl_2}) \cup (D_{cFl_1} \cap D_{cFl_2})$: el conjunto de eventos que mantienen la diferencia de los valores entre cFl_1 y cFl_2, son aquellos eventos compartidos por los fluentes como eventos de incremento o disminución.

    Cabe notar que I_cFl_{1-2}, D_cFl_{1-2} y E_cFl_{1-2} son conjuntos mutuamente excluyentes y su unión conforma el conjunto de eventos que modifican el valor de los fluentes cFl_1 y cFl_2.

$$
\begin{array}{rl}
\delta = & \{\forall a \in I_{cFl_{1-2}}, -k \leq i \leq k-1 \mid (q_i, a, q_{i+1})\} \\
\cup & \{\forall a \in I_{cFl_{1-2}} \mid (q_k, a, q_{+\infty}), (q_{+\infty}, a, q_{+\infty}), (q_{-\infty}, a, q_{-\infty})\} \\
\cup & \{\forall a \in D_{cFl_{1-2}}, -k+1 \leq i \leq k \mid (q_i, a, q_{i-1})\} \\
\cup & \{\forall a \in D_{cFl_{1-2}} \mid (q_{-k}, a, q_{-\infty}), (q_{-\infty}, a, q_{-\infty}), (q_{+\infty}, a, q_{+\infty})\} \\
\cup & \{\forall a \in R_{cFl}, -k \leq i \leq k \mid (q_i, a, q_I)\} \\
\cup & \{\forall a \in R_{cFl} \mid (q_{-\infty}, a, q_{-\infty}), (q_{+\infty}, a, q_{+\infty})\} \\
\cup & \{\forall a \in Act_{\overline{cFl_{1-2}}} \cup E_{cFl_{1-2}}, -k \leq i \leq k-1 \vee i = -\infty \vee i = +\infty \\
 & \mid (q_i, a, q_i)\}
\end{array}
$$

- Los estados de que capturan la validez de la expresiones según la relación $\sim$ son:

$$
\begin{array}{lcl}
cFl_1 = cFl_2 & \rightarrow & F = \{q_0\} \\
cFl_1 \neq cFl_2 & \rightarrow & F = \{q_i \in Q : i \neq 0\} \\
cFl_1 > cFl_2 & \rightarrow & F = \{q_i \in Q : i > 0\} \\
cFl_1 \geq cFl_2 & \rightarrow & F = \{q_i \in Q : i \geq 0\} \\
cFl_1 < cFl_2 & \rightarrow & F = \{q_i \in Q : i < 0\} \\
cFl_1 \leq cFl_2 & \rightarrow & F = \{q_i \in Q : i \leq 0\}
\end{array}
$$

El autómata de Büchi F_{cFl} contiene tantos *estados* como diferencias posibles (acotadas) entre los fluentes, mas dos estados que caracterizan los estados en los cuales el fluente ha superado alguna de las cotas, superior o inferior. El *alfabeto* del mismo se compone por los eventos que incrementan, disminuyen o reinician cada fluente, junto con el resto de eventos del sistema. Los eventos los podemos agrupar, en aquellos que incrementan, disminuyen o no alteran la diferencia entre los valores de los fluentes.

El *estado inicial* es aquel que se corresponde con el valor inicial de la diferencia entre los valores iniciales de los fluentes $N_{cFl_1} - N_{cFl_2}$.

La *función de transición* contiene arcos de incremento y disminución del valor asociado a la diferencia entre los fluentes por los correspondientes eventos. En caso de alcanzar los estados límites $-\infty$ o $+\infty$, todos los eventos mantienen como destino estos estados. Bajo la restricción de que ambos fluentes compartes sus eventos de reinicio, por cada evento en dicho conjunto, generamos un arco cuyo destino es el estado q_I (aquel cuyo valor asociado se corresponde con la diferencia de los valores iniciales de los fluentes). Finalmente, por cada evento del sistema que no altera el valor de la diferencia entre los fluentes, el autómata contiene un arco por cada estado que caracteriza dicha situación.

Los *estados de aceptación* se configuran de acuerdo a la semántica de la relación entre los fluentes que expresa la fórmula. Por ejemplo, si la fórmula es $cFl_1 > cFl_2$, los estados de aceptación serán aquellos que representan una diferencia positiva de cFl_1 con respecto a cFl_2.

A continuación, en la Figura 4.16, ilustramos el autómata para la expresión $cFl_1 \leq cFl_2$ fluente contador definido por $cFl = I_{cFl}, D_{cFl}, R_{cFl}$ con valor inicial 0 y cota k.

Para la expresiones de la forma $cFl_1 \sim cFl_2 + c$ o $cFl_1 \sim cFl_2 - c$, la idea es similar a la anterior, con la diferencia que se considera a k como $k + c$, y sólo cambia la especificación de los estados de aceptación de la siguiente manera:

$$
\begin{array}{lcl}
cFl_1 = cFl_2 \sim c & \rightarrow & F = \{q_{\sim c}\} \\
cFl_1 \neq cFl_2 \sim c & \rightarrow & F = \{q_i \in Q : i \neq \sim c\} \\
cFl_1 > cFl_2 \sim c & \rightarrow & F = \{q_i \in Q : i > \sim c\} \\
cFl_1 \geq cFl_2 \sim c & \rightarrow & F = \{q_i \in Q : i \geq \sim c\} \\
cFl_1 < cFl_2 \sim c & \rightarrow & F = \{q_i \in Q : i < \sim c\} \\
cFl_1 \leq cFl_2 \sim c & \rightarrow & F = \{q_i \in Q : i \leq \sim c\}
\end{array}
$$

Donde $\sim \in \{+, -\}$.

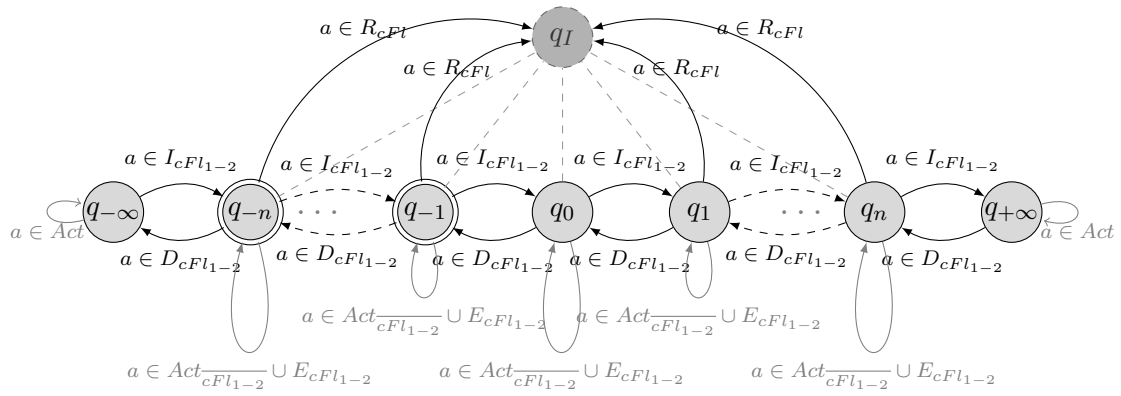


Figura 4.17: Autómata de Büchi que representa la relación entre dos fluentes $cFl_1 \leq cFl_2$

4.8.2 Implementación y Análisis

Como lo mencionamos, nuestra intención con esta extensión de FLTL con fluentes contadores, se enfoca principalmente en su utilización para la especificación de propiedades sobre modelos de sistemas reactivos y su posterior verificación. Es ese sentido nuestra propuesta es la de incorporar a la herramienta LTSA el soporte para la definición de fluentes contadores. Como es de esperarse, una restricción necesaria para ello es establecer un límite (cota superior e inferior) en el rango de valores posibles de los fluentes contadores. Cuando realizamos la especificación de fluentes contadores, podemos diferenciar dos tipo de límites, por un lado aquellos límites que están presentes en el modelo de manera natural a los que llamaremos *bounds* y por el otro, aquellos límites impuestos para poder llevar a cabo el análisis mediante esta técnica que denominamos *scopes*. La diferencia entre ellos radica en que si durante el análisis un *scope* es alcanzado, un evento de *fluent overflow* es notificado y el análisis se torna inconcluso, es decir, a partir de ese punto no podemos inferir nada acerca de la validez o no de la propiedad.

Con este escenario, el procedimiento de análisis puede arrojar los siguientes resultados: *(i) falso*, cuando el proceso encuentra un contraejemplo contenido en los *scopes* especificados, *(ii) verdadero*, cuando el proceso no encuentra un contraejemplo en los límites propuestos y no se a notificado de *fluent overflow*, y *(iii) puede ser*, ningún contraejemplo fue encontrado pero el evento de *fluent overflow* fue notificado.

Con el objetivo de verificar propiedades expresadas en CFLTL, proponemos un enfoque basado en LTSA. Inicialmente consideramos la siguiente sintaxis para la especificación de fluentes contadores:

```
⟨CFluentDef⟩      ::= 'cfluent' ⟨fluent_name⟩ ⟨fluent_bounds⟩ '='
                      '<'⟨incremental_events_set⟩ ',' ⟨decremental_events_set⟩ ','
                         ⟨reset_events_set ⟩ '>' 'initially' ⟨initial_value ⟩
⟨fluent_bounds⟩   ::=  ( '[' | '(' ) ⟨min_value⟩ '..' ⟨max_value⟩ ( ']' | ')' )
```

donde *corchetes* y *paréntesis* indican el tipo de límite, *bound* o *scope* respectivamente. Como ejemplo, consideremos la definición del fluente `ERRORS` y la propieadad `P2` mencionadas anteriormente.

```
cfluent ERRORS [0..3] = <{badPwd},{},{correctPwd,unlockAcct}> initially 0
assert CORRECT_LOCKED = [] (LOCKED ->ERRORS >= 3)                    (1)
```

Teniendo en cuenta la imposición de límites a los fluentes contadores, nuestra propuesta es reducir CFLTL a FLTL para luego realizar la verificación mediante LTSA. De manera general, para operar con los fluentes contadores, generamos como parte del modelo a analizar un proceso monitor que se encarga de analizar la ocurrencia de eventos que modifican los valores de este tipo fluentes. Para analizar la validez de una expresión contadora unitaria (de la forma $cFl \sim c$), el monitor mantiene el valor del fluente contador asociado. Para el caso de expresiones binarias ($cFl_1 \sim cFl_2 \pm c$), el monitor mantiene la relación entre los fluentes que conforman la expresión calculando al diferencia entre ellos. Luego, por medio de fluentes proposicionales, capturamos los eventos que dispara el monitor con el fin de definir estados abstractos que representa el valor (lógico) de cada una de las expresiones contadoras. De esta manera, reemplazando en la formula CFLTL las expresiones contadoras por los correspondientes fluentes proposicionales que las caracterizan, podemos reducir el análisis a FLTL.

Para ejemplificar nuestro enfoque, consideremos su aplicación al ejemplo del cajero automático con la propiedad especificada en (1):

```
CFmon = CFmon_B[0],
CFmon_B[i:0..3] = (
         {cancel,correctPwd,unlockAcct} ->errors[0] ->ok -> CFmon_B[0]
       | when (i<3) badPwd ->errors[i+1] ->ok -> CFmon_B[i+1]
       | when (i>=3) badPwd ->errors[i] ->ok -> CFmon_B[i] ).
fluent ERRORS_GEQ_3 = <errors[3], errors[0..2]> initially false
fluent OK = <ok,{cancel,correctPwd,unlockAcct,badPwd}>
assert CORRECT_LOCKED = [] (OK ->(LOCKED ->ERRORS_GEQ_3))
```

Aquí, `CFmon` es el proceso monitor y `CORRECT_LOCKED` es la propiedad transformada. Cabe notar que el evento `ok`, y su correspondiente fluente proposicional `OK`, aseguran la atomicidad en las tareas de actualización que produce el monitor y este fluente es utilizado como precondición en la fórmula reducida.

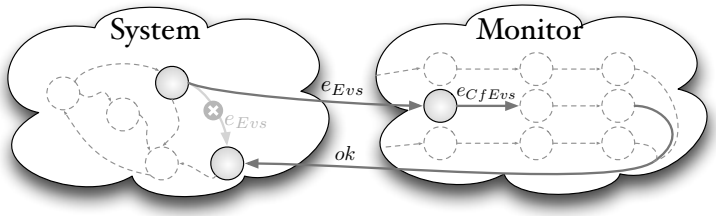
De manera general, dada una especificación FSP de un sistema `Sys` y una propiedad CFLTL ϕ conteniendo las expresiones contadoras $\epsilon_1, \ldots, \epsilon_n$(con referencias a los fluentes contadores $cFl_1, cFl_2, \ldots cFl_m$), para realizar la verificación mediante LTSA, generamos un proceso `Sys'` y la fórmula FLTL reducida ϕ' de la siguiente manera:

`Sys'` se obtiene mediante la composición paralela entre el sistema original `Sys`, el proceso monitor`CFMon` y un sincronizador `SYNCH`, donde el proceso monitor lo llamamos `CFmon` y `SYNCH` es un planificador que evita el *interleaving* entre los eventos de actualización del monitor con los del sistema como lo ilustra la Figura. 4.18 (b). La especificación de `SYNCH` se detalla en la Figura 4.18 (a), donde `Evs` es el conjunto de eventos del sistema, `MonEvs` es el conjunto de eventos monitoreados, y `CfEvs` es el conjunto de eventos de actualización del monitor.

```
SYNCH=({Evs\MonEvs} ->SYNCH
      | MonEvs ->
CFSYNCH),

CFSYNCH=(CfEvs ->CFSYNCH
      | ok -> SYNCH).
```

a)



b)

Figura 4.18: Sincronizador y una descripción abstracta del mismo.

A continuación describimos el proceso de construcción del monitor y la reducción de la fórmula. El monitor utiliza parámetros para mantener el valor de las expresiones contadoras. Para expresiones unitarias el parámetro se corresponde con el valor del fluente asociada a la expresión y para expresiones binarias, el parámetro representa el valor de un fluente relativo que caracteriza la diferencia entre los fluentes que conforman dicha expresión contadora binaria.

Inicialmente generamos el encabezado del proceso monitor que la especificación de sus parámetros, cada uno de éstos inicializados con el correspondiente valor de la expresión contadora. El monitor analiza la ocurrencia de eventos monitoreados, es decir, eventos del sistema que formen parte de los fluentes contadores asociados a las expresiones. Cuando uno de ellos sucede, el monitor dispara eventos que notifican el valor asociado a cada expresión, estos eventos son capturados por fluentes proposicionales que mantiene el valor de verdad de la expresiones.

Para las situaciones donde los parámetros no se encuentra en los límites de sus rangos, por cada evento monitoreado e, se genera un caso del proceso monitor con el siguiente esquema:

```
| when (γ) e ->ϵ1[v'_ϵ1] ->... ->ϵn[v'_ϵn] ->ok ->CFmon_B [v'_ϵ1] ... [v'_ϵn]
```

Donde γ expresa la guarda que garantiza la situación de los límites y los eventos $\epsilon_i[v_{\epsilon_i}]$ notifican los cambios de cada expresión monitoreada. Luego, por cada límite de la expresiones y eventos que puede modificarla se generan de manera excluyente casos de proceso dependiendo del tipo de límite: (i) si el límite es *scope* se notifica con el evento `fluent_overflow` o, (ii) en caso de que el tipo de límite fuera *bound*, simplemente se mantiene el valor actual.

Cabe aclara que por cada evento monitoreado, el proceso monitor analiza *todos* sus posibles modificaciones, es decir, la disyunción de las guardas para cualquier evento siempre es *verdadera*, garantizando que el monitor nunca bloquea al sistema.

Como mencionamos, la reducción de la fórmula CFLTL a FLTL se realiza mediante el reemplazo de expresiones contadoras por los fluentes proposicionales, así dada una fórmula $\phi = \circlearrowright \psi$, donde $\circlearrowright$ representa cualquier operador temporal, generamos la fórmula $\phi' = \circlearrowright (fl_{\texttt{ok}} \wedge \neg fl_{\texttt{fluent_overflow}} \implies \psi[\epsilon_i/fl_i])$, donde cada expresión contadora es reemplazada por su fluente proposicional, y la *precondición* excluye del análisis las situaciones de actualizaciones de proceso monitor o si algún *scope* fue excedido.

Como es de esperar, el fluente $fl_{\texttt{ok}}$ se activa mediante el evento `ok`, y se desactiva por cualquier otro evento monitoreado del sistema, $fl_{\texttt{fluent overflow}}$ se activa con el evento `fluent overflow` y una vez activado no se desactiva mas.

Para cada expresión ϵ_i, se genera el correspondiente fluente proposicional fl_i, cuya configuración varía dependiendo del tipo de expresión. Si ϵ_i es de la forma $cFl \sim c$, entonces el fluente se define como: `fluent` fl_i `= <` $\epsilon_i[\sim c]$, $\epsilon_i[\nsim c]$ `>`, donde $\epsilon_i[\sim c]$ es el conjunto de valores de ϵ_i que satisfacen la relación $\sim c$ bajo los límites l_{ϵ_i} y u_{ϵ_i}. Finalmente, el valor inicial se calcula evaluando la expresión con el valor inicial del fluente. En caso de expresiones binarias, es decir, ϵ_i es de la forma $cFl_1 \sim cFl_2 \pm c$, generamos: `fluent` fl_i `= <` $\epsilon_i[\sim 0 \pm c]$, $\epsilon_i[\nsim 0 \pm c]$ `>`.

Con este proceso, se realizaron pruebas sobre casos de estudio con resultados satisfactorios que validan la necesidad de disponer de este tipo de especificaciones a la hora de analizar modelos de sistema reactivos.

4.9 Sumario

En este capítulo nos concentramos en el análisis de propiedades sobre workflows. Presentamos un lenguaje lo suficientemente expresivo para capturar una gran variedad de situaciones de los mismos. Para su aplicación a modelos procedurales, presentamos una traducción automática de workflows YAWL a especificaciones FSP.

A partir de dicha codificación, utilizamos la herramienta LTSA para automatizar la verificación. Hemos elegido YAWL como la base de nuestro trabajo debido a su semántica formal, y su soporte para una amplia gama de patrones de workflows. Como mencionamos, YAWL permite la verificación de algunas propiedades generales sobre workflows, tales como *soundness* o ausencia de deadlock [97], pero no provee un marco lo suficientemente flexible para expresar otro tipo de propiedades de comportamiento.

Nuestra codificación traduce YAWL a FSP, 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, en las cuales la expresión de propiedades está vinculada a predicar sobre proposiciones en estados. 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 [60, 82]. A diferencia de DECLARE [75], donde se utiliza LTL para especificar restricciones sobre modelos declarativos de workflows, nuestra propuesta captura todas las plantillas de propiedades propuestas en la misma, brindando no sólo una prueba de consistencia en un modelo declarativo, sino la verificación de las mismas sobre un modelo procedural. Otro trabajo relevante es [51], donde se propone un método basado en autómatas para la formalización de workflows, pero con limitaciones de poder expresivo respecto de YAWL.

En el marco del presente trabajo se realizaron dos herramientas que respaldan la aplicación de la propuesta. La primera automatiza el proceso de traducción tomando como entrada un modelo YAWL y generando como salida el correspondiente modelo FSP. La segunda contribuye a la utilización de la técnica, asistiendo al usuario final en la definición de situaciones y propiedades sobre el modelo e integrando el proceso general de verificación. Estas herramientas están públicamente disponibles [5].

Tomando ventaja sobre la caracterización de *workflows* propuesta mediante álgebras de procesos, específicamente por medio de FSP, detallamos un proceso para generación de modelos de *workflows* tomando como premisas, un conjunto de tareas que colaboran en la realización de un objetivo y una serie de restricciones entre las mismas expresadas mediante especificaciones en una lógica temporal, más precisamente mediante fórmulas en FLTL.

Finalmente, y como contribución para la especificación y verificación de propiedades temporales, se propone una extensión a la Lógica Termporal Lineal con Fluentes, que complementa la definición de fluentes proposicionales con *fluentes contadores*

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

que permiten caracterizar de manera directa y natural situaciones donde el análisis expresa la necesidad de *contar* eventos del sistema.

Capítulo 5

Productos como referentes explícitos en el modelado de Workflows

5.1 Introducción

Como mencionamos anteriormente, el constante esfuerzo de distintas organizaciones por el perfeccionamiento de sus procesos, en busca de optimizar los mismos en cuanto a eficiencia y control, ha impulsado el actual auge de diversos métodos y lenguajes para el modelado de procesos de negocios. Existen diversos lenguajes para la descripción de workflows, muchos de los cuales hemos mencionado en el Capítulo 3. Como analizamos en dicho capítulo, si bien la mayoría de los lenguajes poseen en sus modelos elementos comunes, cada uno de ellos brinda un enfoque particular que lo distingue de los demás y lo posiciona para su aplicación.

Un aspecto que los lenguajes existentes suelen relegar, o en el mejor de los casos tratar como un elemento secundario en sus modelos, es la representación de la información manipulada por los procesos. Esta información, que en algunos casos puede estar asociada a ciertas actividades o eventos, es capturada en general como la descripción de la relación *input/output* de las mismas. Dependiendo de las características particulares de los lenguajes, el grado de detalle y especificación de la información varía. Por ejemplo, en caso de YAWL, se provee un formato muy rico (XML) para describir la información; en otros lenguajes, como EPC, que forman parte de un proceso de especificación integral, existen diagramas específicos para representar la información, tales como diagramas de Entidades y Relaciones. En [98] se describen y analizan diferentes patrones de datos (*Data patterns*) relacionados a la *manipulación* de información como parte de un proceso de negocio.

Una característica que consideramos importante en el modelado de procesos de negocios es la posibilidad de contar en el lenguaje con un elemento que permita especificar *productos* (una referencia a un elemento empírico con propiedades) como un referente explícito del modelo. La norma internacional ISO-9001 define un proceso como "una actividad que utiliza recursos, y que se gestiona con el fin de permitir que los elementos de entrada se transformen en resultados" [43]. Creemos que en modelos reales, por ejemplo, en caso de procesos industriales o modelos en los cuales no existe un intercambio directo de información entre los procesos, la necesidad de contar con la especificación de productos en el modelo es tan importante como las especificaciones de los procesos que los manipulan. Además, la especificación de los productos no sólo ayuda a describir condiciones complejas de composición de procesos, sino también nos brindan la posibilidad de analizar el comportamiento de los mismos durante la ejecución del sistema. En particular, nos interesa analizar las ventajas de contar con lenguajes que permitan la especificación de propiedades sobre los productos que forman parte de procesos de negocios y su posterior análisis por medio de técnicas automáticas o semi-automáticas, como por ejemplo *model checking*.

Otra característica en la que estamos interesados analizar de un lenguaje para el modelado de procesos de negocios es la posibilidad de contar con referencias al tiempo real, es decir, poder especificar, en alguna medida temporal concreta, restricciones o condiciones para los procesos o productos modelados. Como se puede observar en [56, pp. 3] y [7, Sección 1.7] entre otros, existen situaciones en el modelado de los procesos para las cuales es clara la necesidad de contar con la especificación de cotas de tiempo real para los procesos. Esta información de tiempo real, además de reflejar características del modelo real, nos brinda la posibilidad de analizar condiciones (propiedades) sobre los modelos que involucren restricciones de tiempo real.

Product Process Modeling Language (PPML) [63] es un formalismo para el modelado de procesos de negocios con una semántica basada en sistemas de transición de estados temporizados [42]. Sus modelos se construyen mediante la especificación y composición de *procesos* y su interacción con los *productos* que manipulan. PPML ofrece una variedad de elementos para la especificación de sus modelos, en particular podemos mencionar la asignación de cotas de tiempo real a los procesos. Esta característica lo hace adecuado para la especificación formal de procesos en los cuales existan restricciones de este tipo.

En este capítulo, tomamos PPML como lenguaje para la especificación de workflows, analizando en detalle su definición y semántica formal. Finalmente, con el objetivo de ampliar el campo de aplicación y facilitar la especificación de procesos de negocios con PPML, en este capítulo proponemos una sintaxis de alto nivel para

el mismo. Con ella podemos brindar soporte automatizado para el análisis de sus modelos como lo abordaremos en el Capítulo 6, donde se propone la verificación de propiedades mediante *model checking*. Finalmente, mostramos un caso de estudio que ilustra la aplicación de PPML para la especificación de workflows.

5.2 Product Process Modelling Language

PPML es un lenguaje formal para la descripción de procesos [63]. La concepción de los procesos que utiliza PPML para la descripción de sus modelos se basa en el método analizado en [74]. Este método comprende la descripción de sistemas como una relación entre *productos* que describen elementos del sistema y *procesos* que los manipulan. PPML provee para la especificación de sus procesos tres construcciones básicas: *productos*, *procesos* y *compuertas*.

Los productos son entidades del lenguaje caracterizadas por un conjunto de atributos. Los productos pueden ser manipulados por procesos que representan comportamientos o tareas del sistema. Los procesos tienen estipulado el tiempo aproximado que demoran en realizar su tarea. La especificación de los procesos se modela mediante una relación de entrada/salida de productos.

Con el objetivo de mantener las especificaciones relativamente simples, y dada la necesidad de tal simplicidad para facilitar la definición de una semántica formal para el lenguaje, PPML establece como restricción en la especificación de sus procesos, entradas y salidas simples. Esto es, sólo un producto de entrada y un producto de salida pueden ser manejados por un proceso. Para abordar la especificación de procesos donde sea necesario más de un producto como entrada o salida, PPML proporciona el concepto de *producto compuesto o empaquetado*. Este tipo de productos proporcionan operaciones para combinar varios productos en uno y proyectar productos que los conforman.

Para capturar el comportamiento del sistema, los procesos pueden componerse por medio de diferentes elementos denominados *compuertas*. Entre las compuertas que provee PPML para la composición de procesos, podemos mencionar los *multiplexores*, *demultiplexores* y *semáforos*.

A continuación presentamos una descripción de cada elemento que brinda PPML para la especificación de sus modelos.

5.2.1 Productos

Los *productos* modelan entidades del sistema en términos de sus atributos observables. Los atributos capturan las características relevantes del modelo empírico,

por ejemplo altura, peso, color, etc. Estas características pueden ser directamente observables o pueden ser calculadas a partir de otras mediante funciones. Los valores que poseen las características deben estar determinados en base a una escala apropiada [33].

Cuando describimos un producto como parte de un sistema, su descripción, es decir, sus atributos, están circunscritos a los valores individuales del mismo. Por ejemplo, si describimos un automóvil, el mismo va a tener asociados a las características relevante sus respectivos valores como por ejemplo: color rojo, modelo 2007, tipo sedan, etc. Si bien los modelos están conformados por elementos (productos) específicos, estamos interesados en poder especificarlos a un nivel de abstracción mayor, en sentido de poder dar una caracterización o *clasificación* general de los mismos. Luego, los elementos individuales que conforman el modelo serán *instancias* de estas clasificaciones. En términos formales, las clasificaciones serán teorías de alguna lógica, por ejemplo Lógica de Primer Orden, y las instancias, interpretaciones de las mismas [30]. En términos de lenguajes de programación, las descripciones son *clases* y las instancias se corresponden con los *objetos* de las mismas [68].

Además de los atributos propios, como parte de la especificación de los sistemas y su posterior análisis, necesitamos asociar a las descripciones (clases) un *nombre* distintivo para referirnos a ellas. Por otro lado, también es necesario contar con una distinción entre elementos o instancias de la misma clase. Si bien en ciertos ambientes es una característica propia del elemento del sistema que se está formalizando, por ejemplo en caso de personas es su documento, a cada producto (instancia de alguna clase) se le asocia un atributo denominado *código* que lo identifica unívocamente dentro del modelo. Como lo mencionamos, dada la presencia del tiempo como parte de los modelos, otra información interesante a considerar de los productos es el instante de tiempo en el cual fue creado o modificado. Esta información, que se asocia como un atributo de cada producto, luego puede ser utilizada para realizar análisis temporales (o de tiempo real) de los modelos.

En la especificación de sistemas generalmente existen situaciones donde es necesario caracterizar elementos complejos, es decir, elementos del modelo que están conformados por otros elementos. Por ejemplo, un automóvil, en un grado de descripción más detallada, está compuesto por una carrocería, un motor y cuatro ruedas. Para referirnos a esta clase de productos PPML propone el concepto de *producto estructurado*, los cuales, además de características del producto como un todo, tiene asociados de manera explícita los productos que lo forman parte del mismo.

La naturaleza de este tipo de situaciones complejas no siempre se corresponde con la representación del mundo real. En particular, existen situaciones donde surge la necesidad de *agrupar* productos como parte de la concepción formal del sistema. Un ejemplo de esto es el caso en que se necesita representar el producto de entrada

de un proceso versátil que puede recibir un conjunto disyunto de posibles productos. Los *productos compuestos* son la herramienta que proporciona PPML para modelar esta clase de elementos. Otra situación común podría ser cuando un proceso obtiene varios productos de entrada para luego componerlos en un producto de salida. En estos casos, los *productos compuestos* permiten lidiar de manera sencilla con la restricción sobre los procesos que impone la unicidad de productos de entrada y salida, simplemente *empaquetando* varios productos en uno.

Productos Atómicos

Los productos atómicos permiten caracterizar, mediante una especificación formal, elementos del sistema a modelar. Estas descripciones caracterizan productos a través de los atributos relevantes de los mismos, y los valores posibles de estos atributos se corresponden con alguna escala o método preciso de medición.

Estos valores deben ser directamente observables o pueden calcularse a partir de otros valores de atributos existentes. Por ejemplo, una habitación rectangular puede caracterizarse mediante atributos como la cantidad de ventanas, largo y ancho, superficie, etc. Si bien todos los atributos son observables, mientras la cantidad de ventanas, largo y ancho son atributos cuyos valores se corresponden con una escala de medición (un número natural para la cantidad de ventanas y longitudes expresadas en metros para largo y ancho), la superficie de la habitación puede derivarse (calcularse) como resultado de la multiplicación entre el largo y el ancho de la misma.

Estas reglas o relaciones entre atributos deben estar expresadas formalmente. Es decir, para considerar un atributo como derivado de otros, deben existir fundamentos formales, por ejemplo funciones matemáticas o leyes en alguna teoría, que oficien de instrumento de cálculo para asignar a este tipo de atributos un valor adecuado a su forma de medición. Estas reglas, junto con los atributos y escalas forman el vocabulario que caracteriza a un producto atómico (especifica la clase).

En un sentido formal, una vez analizadas y establecidas las características del producto a modelar, se formaliza la *signatura* [88, 110] del producto. La misma contiene las referencias a dichas características además de establecer las reglas (operaciones) que definen los atributos derivados. Estas reglas establecen los *axiomas* de la *presentación* $\langle \Sigma, A \rangle$ del producto. De esta manera, definimos un *producto atómico* de la siguiente manera:

Definición 5.1. ***Producto Atómico***

Dados los conjuntos Codigo, Nombre y Tiempo, un producto atómico *p es una tupla* $\langle codigo_p, nombre_p, tiempo_p, atributos_p, \langle \Sigma_p, A_p \rangle \rangle$ *donde:*

- $codigo_p$ *es una constante del conjunto Codigo que identifica el producto unívocamente en la descripción del sistema.*

- *$nombre_p$ es una constante del conjunto $Nombre$ que describe en lenguaje natural una referencia del producto.*

- *$tiempo_p$ es una constante del conjunto $Tiempo$ que especifica el instante del tiempo en el cual los atributos del producto fueron asociados con sus respectivos valores.*

- *$atributos_p$ es un subconjunto de Σ_p. Estas constantes designan las características del producto que especifican. Este conjunto está particionado en dos subconjuntos: $atributos_directos_p$ y $atributos_derivados_p$. El primero consta de referencias a las características observables que pueden medirse directamente mediante procedimientos de medición adecuados. Los $atributos_derivados_p$ son aquellos que pueden calcularse a partir de los directos por medio de* axiomas *especificados en A_p.*

- *$\langle \Sigma, A \rangle$ es la teoría de presentación del producto, que consta de la* signatura *Σ, que contiene todas las constantes definidas en el producto y A el conjunto de axiomas que rigen su comportamiento.*

Cabe destacar el que subconjunto $atributos_derivados_p$ puede ser vacío, cuando necesariamente todos sus atributos sean directamente observables.

Definición 5.2. ***Instancia de un Producto***
Dada la especificación de un Producto $p = \langle codigo_p, nombre_p, tiempo_p, atributos_p, \langle \Sigma_p, A_p \rangle \rangle$, definimos una instancia de p como una interpretación en el sentido lógico de $\langle \Sigma, A \rangle$.

Productos Compuestos

Como lo mencionamos anteriormente, existen situaciones donde es necesario especificar la composición de productos. Tal composición surge con un sentido puramente descriptivo, es decir, que la composición no captura una característica intrínseca del objeto real, sino que la misma sólo forma parte de elementos técnicos del lenguaje con algún objetivo de especificación particular. Para abordar estas situaciones, PPML brinda dos tipos de *productos compuestos*: *empaquetados* y *selectivos*.

Definición 5.3. ***Producto Compuesto Empaquetado***
Un producto compuesto empaquetado es un par $\langle codigo, \times(P_{c_1}, .., P_{c_n}) \rangle$, donde $\times(P_{c_1}, .., P_{c_n})$ es una inyección de los componentes P_{c_i} en el producto cartesiano.

Este tipo de productos constituye una herramienta muy útil lidiar con la restricción de la unicidad de entrada/salida en la especificación de los procesos. Así por

ejemplo, en caso de que un proceso requiera como entrada varios productos para su manipulación, puede apelarse a este tipo de recurso en el cual se empaquetan dichos productos como uno, con la posterior posibilidad de su utilización individual de sus componentes mediante proyecciones sobre él.

Definición 5.4. ***Producto Compuesto Selectivo***

Un producto compuesto selectivo es un par $\langle codigo, \iota(\mathcal{P})\rangle$ *donde* $\mathcal{P}$ *es un conjunto finito de productos y* $\iota(\mathcal{P}) \in \mathcal{P}$.

Estas descripciones de productos compuestos pueden utilizarse en situaciones donde existan procesos versátiles que puedan recibir como entrada diferentes tipos de productos. El tipo de sus componentes se restringe a especificaciones de producto atómicos o empaquetados. Este tipo de descripciones se corresponden con la proyección de un elemento en un coproducto.

Cabe notar que, como lo mencionamos anteriormente, este tipo de descripciones de productos son una herramienta que permite lidiar técnicamente con cuestiones de especificación o modelado, en el proceso de formalización de un sistema. Por este motivo, las especificaciones de productos compuestos sólo poseen, además de las declaraciones de sus componentes, el código que los identifica unívocamente en el modelo de un sistema. En caso de ser necesario, puede suponerse que el tiempo de un producto compuesto es el mayor de los tiempos de sus componentes (en caso de productos empaquetados) o el tiempo del producto elegido (en caso de productos selectivos).

Productos Estructurados

A diferencia de los productos compuestos, el propósito de los productos estructurados es poder capturar la complejidad de un referente empírico, es decir, un elemento del sistema a modelar. En un sentido descriptivo, un producto estructurado es la especificación de un producto en el cual pueden distinguirse en él otros productos que lo componen. Para capturar el concepto, PPML brinda dos tipos de *productos estructurados* que abarcan diferentes aspectos de diseño o arquitectura de los mismos: productos *especializados* y *agregados*.

Definición 5.5. ***Producto Estructurado Especializado***

Dada la definición de un producto $p = \langle codigo_p, nombre_p, tiempo_p, atributos_p, \langle \Sigma_p, A_p\rangle\rangle$, *una* especialización *del mismo, es un nuevo producto* $p' = \langle codigo_{p'}, nombre_{p'}, tiempo_{p'}, atributos_{p'}, \langle \Sigma_{p'}, A_{p'}\rangle\rangle$ *donde:*

- $codigo_{p'}$, $nombre_{p'}$ *y* $tiempo_{p'}$ *son constantes de los conjuntos* $Codigo$, $Nombre$ *y* $Tiempo$ *respectivamente, con la misma semántica que para los producto atómicos.*

- *$atributos_{p'}$ es el conjunto de atributos del producto especializado que incluye los atributos de p.*

- *$\langle \Sigma_{p'}, A_{p'} \rangle$ es una extensión de la presentación $\langle \Sigma_p, A_p \rangle$ incorporando los símbolos que conllevan los nuevos atributos y sus correspondientes axiomas, si los tuvieran.*

Los productos especializados permiten definir un nuevo producto como la extensión de uno existente, incorporando nuevos atributos y axiomas que caracterizan la extensión. Cuando la descripción de un producto involucra a más de un producto componente, a éstos los podemos especificar por medio de los productos estructurados agregados. A diferencia de una especialización, los productos agregados pueden estar conformados por más de un producto componente. Además, estos componentes pueden tener las mismas características. Este requisito en la descripción de un producto agregado nos obliga a introducir el concepto de copia de una presentación.

Definición 5.6. ***Copias de una Presentación***
Dada una presentación $T = \langle \Sigma_T, A_T \rangle$, una serie de n copias *de la misma, anotada como $C_1, C_2, \ldots, C_n : T$, es una serie de n especificaciones de productos donde cada $C_i = \langle codigo_{C_i}, nombre_{C_i}, tiempo_{C_i}, atributos_{C_i}, T \rangle$.*

Las copias de presentaciones son la herramienta formal que permite describir los productos agregados en los cuales pueden existir diferentes componentes de un mismo tipo. Estas copias son una manera compacta de especificar varios productos, que si bien comparte las mismas características, son independientes entre sí, es decir, cada uno de ellos tiene su propia identificación y descripción temporal. Esta idea puede asociarse, en un sentido de lenguaje de programación, a la noción de registro donde varios campos son del mismo tipo.

Teniendo en cuenta el concepto de copias de presentaciones, podemos definir los productos estructurados agregados de la siguiente manera:

Definición 5.7. ***Producto Estructurado Agregado***
Un producto estructurado agregado es $p = P_1 : T_1, P_2 : T_2, \ldots, P_3 : T_3$ en $\langle codigo_p, nombre_p, tiempo_p, atributos_p, T \rangle$, donde:

- *P_i son los productos que lo conforman y T_i, sus respectivas presentaciones.*

- *$codigo_p$, $nombre_p$ y $tiempo_p$ son constantes de los conjuntos $Codigo$, $Nombre$ y $Tiempo$ respectivamente, con la misma semántica que para los producto atómicos.*

- *$atributos_p$ es el conjunto de atributos del producto agregado p.*

- *T es la presentación $\langle \Sigma_p, A_p \rangle$ inducida por la especialización de la unión disjunta de las presentaciones de sus componentes e incorporando los símbolos que conllevan los nuevos atributos y sus correspondientes axiomas, si los tuvieran.*

Como mencionamos, podemos asociar las definiciones de productos con el concepto de *clases* en la teoría de lenguajes de programación Orientada a Objetos y consecuentemente, a las instancias de estas definiciones como los *objetos* en dicha teoría. Siguiendo la analogía, son las instancias de productos las que tendrán lugar en la ejecución de los procesos.

5.2.2 Procesos

Un proceso modela un elemento del sistema que transforma un producto de entrada en un producto de salida. Al igual que los productos, los procesos pueden ser **atómicos** (sin estructura interna o subprocesos) o **estructurados**. El concepto que utiliza PPML para describir las transformaciones de los productos mediante sus procesos es el de *máquina virtual*, es decir, cada proceso tiene asociado una *máquina virtual* que interpreta los comandos básicos necesarios para operar con los productos que el mismo procesa. En términos informales una máquina virtual es un objeto con métodos.

Previo a la definición de un proceso en PPML, introducimos la formalización del concepto subyacente de *máquina virtual* [34] como un *framework lógico* sobre sistemas temporales de transición de estados llamado RETOOL [21]. RETOOL es un lenguaje formal (*modal action logic*) basada en una extensión del operador δ [90], con límites de tiempo real.

Dados un conjunto de símbolos de *atributos* A y un conjunto de símbolos de *acciones* Γ, el lenguaje que brinda RETOOL para sus especificaciones es el siguiente:

- Proposiciones de Estado (SP): $p ::= a \mid \neg p \mid p \rightarrow p'$
- Términos de Acciones (AT): $t ::= g \mid p_l \delta^u q$
- Fórmulas (FOR): $\phi ::= a \mid t_1 \supset t_2 \mid \neg\phi \mid \phi \rightarrow \phi' \mid [t]\phi \mid []p$

donde $a \in A$, $g \in \Gamma$, $p, q \in SP$, $l \in TIME - \infty$, $u \in TIME$ y $t, t_1, t_2 \in AT$. $TIME$ es un conjunto con orden total bajo la relación $\leq$, con elementos mínimo 0 y máximo ∞.

Su semántica está fundada en la noción de *ejecución* sobre un *Timed Object Frame* de un conjunto de variables y acciones.

Definición 5.8. ***Timed Object Frame***

Un Timed Object Frame $\mathcal{F}$ *sobre* $\langle A, \Gamma \rangle$ *es una tupla* $\langle \mathcal{W}, \rightarrow, l, u, I, w_0 \rangle$ *donde:*

- $\mathcal{W}$ *es el conjunto de posibles estados.*
- $\rightarrow: \mathcal{W} \times \Gamma \rightarrow \mathcal{W}$ *es la relación de transición.*
- $l : \Gamma \rightarrow TIME - \infty$, *es una función que establece para cada acción* $g \in \Gamma$ *su límite temporal inferior.*
- $u : \Gamma \rightarrow TIME$, *es una función que establece para cada acción su límite temporal superior. Además* $\forall g \in \Gamma : u(g) \geq l(u)$.
- $I : A \rightarrow 2^{\mathcal{W}}$ *interpreta los atributos sobre los estados.*
- $w_0 \in \mathcal{W}$ *es el estado inicial.*

Al par $\langle A, \Gamma \rangle$ se lo denomina la *signatura* del *Timed Object Frame*.

La semántica del lenguaje sobre un Timed Object Frame $\mathcal{F} = \langle \mathcal{W}, \rightarrow, l, u, I, w_0 \rangle$ se establece de la siguiente manera:

- Para una proposición de estado p, su semántica es el conjunto de estados determinado por:

$$\begin{array}{lcl} [\![a]\!]^{\mathcal{F}} & = & I(a) \\ [\![\neg p]\!]^{\mathcal{F}} & = & \mathcal{W} - [\![p]\!]^{\mathcal{F}} \\ [\![p \rightarrow p']\!]^{\mathcal{F}} & = & (\mathcal{W} - [\![p]\!]^{\mathcal{F}}) \cup [\![p']\!]^{\mathcal{F}} \end{array}$$

- Para un término de acción t, su semántica es el conjunto de transiciones determinado por:

$$\begin{array}{lcl} [\![g]\!]^{\mathcal{F}} & = & \{(w, w') \mid w \xrightarrow{g} w'\} \\ [\![p_l \delta^u q]\!]^{\mathcal{F}} & = & \{(w, w') \mid \exists g \in \Gamma[(w \xrightarrow{g} w') \wedge en(g) \subseteq [\![p]\!]^{\mathcal{F}} \wedge w' \in [\![q]\!]^{\mathcal{F}} \wedge \\ & & \wedge (l \leq l(g) \leq u(g) \leq u)]\} \end{array}$$

  donde $en(g)$ significa que g está habilitada.

- Finalmente, la validez de una fórmula en un estado w de $\mathcal{F}$ está definida por:

$$\begin{array}{lll} \mathcal{F}, w \models p & \text{sii} & w \in [\![p]\!]^{\mathcal{F}} \\ \mathcal{F}, w \models (t_1 \supset t_2) & \text{sii} & [\![t_1]\!]^{\mathcal{F}} \subseteq [\![t_2]\!]^{\mathcal{F}} \\ \mathcal{F}, w \models \neg\phi & \text{sii} & \text{no } \mathcal{F}, w \models \phi \\ \mathcal{F}, w \models \phi \rightarrow \phi' & \text{sii} & \mathcal{F}, w \models \phi \text{ implica } \mathcal{F}, w \models \phi' \\ \mathcal{F}, w \models [t]\phi & \text{sii} & \mathcal{F}, w' \models \phi \text{ para todo } w' \text{ tal que } (w, w') \in [\![t]\!]^{\mathcal{F}} \\ \mathcal{F}, w \models []p & \text{sii} & \mathcal{F}, w_0 \models p \end{array}$$

Definición 5.9. ***Secuencia Temporal de Estados***

Una Secuencia Temporal de Estados *sobre un Timed Object Frame $\mathcal{F}$ es un par $\langle \sigma, T \rangle$ donde σ es una secuencia infinita de estados ($\sigma_i \in \mathcal{W}_{\mathcal{F}}$) y T es una secuencia infinita de tiempos ($t_i \in TIME \backslash_{\infty}$) que satisfacen:*

- *monotonía:* $\forall i \geq 0 : (t_{i+1} = t_i) \vee [(t_{i+1} > t_i) \wedge (\sigma_{i+1} = \sigma_i)]$.
- *progreso:* $\forall t \in TIME - \infty : \exists i \geq 0 : t_i \geq t$.

Estas secuencias establecen una estructura sobre la cual se funda la definición de ejecución como un refinamiento de las mismas, induciendo su correspondencia con las posibles acciones que caracteriza un *Timed Object Frame*.

Definición 5.10. ***Ejecución sobre un Timed Object Frame***

Una ejecución *sobre un Timed Object Frame $\mathcal{F}$ es una secuencia temporal de estados $\langle \sigma, T \rangle$ tal que:*

- *es consistente con el sistema de transición subyacente de $\mathcal{F}$, es decir, para todo $i \geq 0$ existe una transición $\sigma_i \xrightarrow{\sigma(i)} \sigma_{i+1}$.*
- *es consistente con las restricciones temporales:*
  - *Para todo $i \geq 0$ en el dominio de σ, existe un $j \leq i$ tal que $T_i - T_j > l(\sigma(i))$. Además $\sigma(i)$ está habilitada para todo estado σ_k con $j \leq k \leq i$.*
  - *Para todo $g \in \Gamma$ e $i \geq 0$, existe un $j \geq i$ tal que $T_j - T_i \leq u(g)$ y, o bien g no está habilitada en el estado σ_j, o bien $g = \sigma(j)$.*

Cabe notar que la noción que brinda RETOOL para especificar formalmente el concepto de proceso o programa en un ambiente con restricciones de tiempo real mediante el operador δ se asemeja a las especificaciones de Hoare [44] en el sentido de que una tripla $\{p\}g\{q\}$ se relaciona con la fórmula en RETOOL $p \rightarrow [g]q$.

Definición 5.11. ***Máquina Virtual***

Definimos una máquina virtual *(VM) como la presentación $\langle \theta, \Phi \rangle$ donde θ es la signatura de la VM y Φ es un conjunto finito de θ-fórmulas. Un modelo de una VM es un* Timed Object Frame *$\mathcal{F}$ sobre θ, tal que $\forall \phi \in \Phi : \mathcal{F} \models \phi$. Las posibles ejecuciones de VM anotado como comp(VM) es el conjunto de ejecuciones de $\mathcal{F}$.*

Así, una *máquina virtual* nos provee la formalización de funcionalidades (acciones) cuyo comportamiento está determinado en términos de su efecto en variables de estado. Además estas acciones tienen asociadas ciertas precondiciones, que especifican los estados en los cuales pueden ejecutarse, y restricciones (cotas) de tiempo real que determinan los intervalos (temporales) de ejecución de las mismas.

Tomando como soporte la formalización de una máquina virtual, abordamos el concepto de proceso como una transacción que, dado un producto de entrada, lo transforma en un producto de salida. Esta transformación se especifica mediante las condiciones iniciales necesarias para llevarla a cabo y el estado final que captura el objeto de la transformación. Además, la transacción deber respetar restricciones de tiempo real y satisfacer un invariante que garantiza las condiciones que deben cumplirse para llevar a cabo el proceso. Formalmente definimos un proceso atómico como:

Definición 5.12. ***Proceso Atómico***

Un proceso atómico *es un par* $P = \langle proc, VM \rangle$*, donde* $VM = \langle \theta, \Theta \rangle$ *es la especificación de la máquina virtual subyacente y proc esta definido por* $\langle$*codigo, nombre,* $P_{in}, P_{out}, (q, I)_l \Delta^u p \rangle$*, de la siguiente manera:*

- *codigo y nombre identifican y dan nombre al proceso, respectivamente.*
- P_{in}, P_{out} *son los productos de entrada y salida del procesos, respectivamente. Sus atributos son interpretados como variables de estado en* VM*.*
- $(q, I)_l \Delta^u p$ *es la especificación del comportamiento del proceso que caracteriza una* transacción*, es decir, un segmento de ejecución (traza) de* VM *y consta de:*
  - *la* condición inicial q *que establece cuáles son los estados que satisfacen los requisitos necesarios para llevar a cabo el proceso (pre-condición).*
  - *la* condición final p *que especifica cuáles son las propiedades de los estados que satisfacen la condición final del proceso (pos-condición).*
  - *el* invariante I *determinado por una sentencia que debe evaluarse verdadera en todos los estados de la traza.*
  - *las* cotas inferior y superior (l, u) *que especifican el tiempo mínimo y máximo que puede demorar el proceso en completarse.*

Definición 5.13. ***Semántica de un Proceso Atómico***

La semántica de un proceso, denotada por $comp(p)$*, está definida por el subconjunto de* $comp(VM)$ *que satisface* $(q, I)_l \Delta^u p$*. Formalmente:*

$$\sigma, T, i \vDash (q, I)_l \Delta^u p \Longleftrightarrow \exists j, k : i + l \leq j < k \leq i + u :$$

$$(\forall m : i \leq m \leq j : \sigma, T, m \vDash q) \wedge (\sigma, T, k \vDash p) \wedge (\forall n : j \leq n \leq k : \sigma, T, n \vDash I)$$

La semántica especifica el comportamiento de un proceso como una ejecución de la máquina virtual subyacente al proceso, donde existe una subsecuencia temporal que satisface la pre-condición q del proceso (con demora de al menos l unidades de tiempo) y luego, en no más de u unidades de tiempo, se llega a un estado que satisface la pos-condición p. Además, durante este último lapso de la secuencia todos sus estados cumplen con la especificación del invariante I.

Definición 5.14. ***Instancia de un Proceso Atómico***
Una instancia *de un proceso atómico p es una ejecución $p^I \in comp(p)$.*

Como lo mencionamos en la Subsección 3.3.1, en las descripciones de workflows, se necesitan operadores que permitan especificar la distribución del flujo de control a través de los diferentes procesos que forman parte del workflow. A diferencia de los lenguajes mencionados en el Capítulo 3, donde los operadores de sincronización y distribución inducen la composición entre los procesos intervinientes, en PPML existe una distinción entre la composición de procesos como un *proceso estructurado* y las herramientas que el lenguaje provee para coordinar la distribución de los productos que estos procesos manipulan.

Previo a la descripción de los productos estructurados como herramienta para la especificación de composición de procesos, comenzamos con la definición de diferentes elementos que permiten la manipulación de productos que denominamos *compuertas*.

5.2.3 Compuertas

PPML provee el concepto de **compuerta** como herramienta para la estructuración de productos necesarios para la ejecución de los procesos del sistema. Además de ser el soporte para lidiar con la restricción de productos con una entrada y una salida, las compuertas permiten especificar situaciones complejas de distribución y sincronización de productos. Por ejemplo, cuando es necesario distribuir varios productos como entrada de diferentes procesos o para *sincronizar* productos mediante una condición antes de formar parte de la entrada de cierto proceso, se utilizan compuertas.

Cabe destacar que las compuertas sólo forman parte del aspecto técnico del lenguaje, es decir, su especificación no se corresponde directamente con elementos del sistema a modelar sino que proporcionan una herramienta para establecer las condiciones que impone el método. Al igual que lo procesos, las compuertas operan sobre la definición de una máquina virtual subyacente que establece las acciones básicas que pueden emplearse para manipular los productos.

Existen tres tipos básicos de compuertas: *multiplexores*, *demultiplexores* y *semáforos*:

Multiplexores

Este tipo de compuertas permite empaquetar varios productos, que conforman la estrada de la compuerta, en un solo producto como salida de la misma. Formalmente definimos un *multiplexor* como:

Definición 5.15. ***Multiplexor***
Una compuerta multiplexor m_c *es una tupla* $\langle codigo, \mathcal{P}, P, F \rangle$ *donde:*

- *codigo identifica la compuerta unívocamente en el modelo.*
- $\mathcal{P}$ *es el conjunto de productos de entrada de la compuerta.*
- P *es el producto compuesto empaquetado que retorna la compuerta.*
- F *es la función de correspondencia que especifica cómo el producto de salida* P *se define en términos de los* $p_i \in \mathcal{P}$.

Demultiplexores

Las compuertas de tipo *demultiplexor*, como lo expresa su denominación, son la contraparte de las compuertas multiplexor. Estas compuertas reciben como entrada un producto empaquetado y distribuyen su contenido en una cantidad finita de salidas que la compuerta tiene asociadas. Formalmente definimos un *demultiplexor* como:

Definición 5.16. ***Demultiplexor***
Una compuerta multiplexor d_c *es una tupla* $\langle codigo, P, \mathcal{P}, F \rangle$ *donde:*

- *codigo identifica la compuerta unívocamente en el modelo.*
- P *es el producto compuesto empaquetado de entrada a la compuerta.*
- $\mathcal{P}$ *es el conjunto de productos que retorna la compuerta.*
- F *es la función de correspondencia que especifica cómo cada producto de salida* $p_i \in \mathcal{P}$ *se define en términos de* P.

Cabe destacar que tanto para las compuertas multiplexores como para demultiplexores, el producto de salida y entrada, respectivamente, solo puede ser un producto compuesto empaquetado y no un producto estructurado. Si fuera necesario manipular un producto que refleje un objeto del sistema (producto estructurado), dicha manipulación (composición o descomposición) debería estar modelada a través de un proceso.

Semáforos

En la Subsección 3.3.1, mencionamos la presencia de elementos en los lenguajes para el modelado de procesos de negocios que capturan situaciones particulares del modelo que explicitan o imponen ciertas condiciones al flujo de control. En concordancia con ello, PPML propone el concepto de *semáforo* como herramienta para condicionar el flujo de control. Para ello, una compuerta semáforo sólo permite el traspaso del producto de entrada que recibe hacia la salida cuando la fórmula, que especifica una condición de continuación, es verdadera.

Definición 5.17. ***Semáforo***
Una compuerta semáforo s_c *es una tupla* $\langle codigo, P, S\rangle$ *donde:*

- *codigo identifica la compuerta unívocamente en el modelo.*
- *P es el producto que oficia de entrada y salida de la compuerta.*
- *S es una fórmula que especifica la condición que debe satisfacerse para que el producto de entrada se traslade (continúe) como producto de salida de la compuerta.*

5.2.4 Composición de procesos

Como describimos en el Capítulo 4, en el proceso de modelar sistemas, al igual que en programación, necesitamos herramientas para poder *componer* procesos. En los lenguajes de modelado de proceso de negocios pueden observarse desde una perspectiva abstracta, dos tipos de composiciones que determinan una organización jerárquica del modelo: La *composición horizontal* determina la interrelación entre los procesos en un mismo nivel de abstracción y la *composición vertical* permite genera instancias de abstracción del modelo. PPML proporciona diferentes herramientas para abordar ambos tipo de composiciones de procesos.

Comenzamos mostrando los operadores que brinda PPML para la composición horizontal de procesos: *Secuencial*, *Condicional* y *Paralela*.

Consideremos $p1 = \langle codigo_{p1}, nombre_{p1}, P-in_{p1}, P-out_{p1}, (q_{p1}, I_{p1})_{l_{p1}} \Delta^{u_{p1}} p_{p1} \rangle$ y $p2 = \langle codigo_{p2}, nombre_{p2}, P-in_{p2}, P-out_{p2}, (q_{p2}, I_{p2})_{l_{p2}} \Delta^{u_{p2}} p_{p2} \rangle$ dos procesos atómicos sobre una misma máquina virtual.

Composición Secuencial

La composición secuencial entre dos procesos, denotada por ;, especifica la combinación de dos procesos de manera secuencial, es decir, la ejecución del primer proceso y, una vez finalizado éste, la ejecución del segundo. Su semántica está definida en términos de un nuevo proceso de la siguiente manera:

Definición 5.18. ***Composición secuencial de procesos***
La composición secuencial $p1; p2$ *de dos procesos* $p1$ *y* $p2$*, es un proceso* $p = \langle codigo_p, nombre_p, P-in_p, P-out_p, (q_p, I_p)_{l_p} \Delta^{u_p} p_p \rangle$*, donde:*

- $codigo_p$ *y* $nombre_p$ *tienen nuevos valores para identificarlo.*
- $P-in_p = P-in_{p1} \wedge P-out_p = P-out_{p2}$
- $q_p = q_{p1}$
- $p_p = p_{p2}$
- $I_p = q_1 \mathtt{U}_{>l_{p1}} (I_{p1} \mathtt{U}_{\leq u_{p1}-l_{p1}} (p_{p1} \Rightarrow true \mathtt{U} (q_{p2} \mathtt{U}_{>l_{p2}} (I_{p2} \mathtt{U}_{\leq u_{p2}-l_{p2}} p_{p2}))))$
- $l_p = l_{p1} + l_{p2} \wedge u_p = u_{p1} + u_{p2}$

El producto de entrada de la composición es el producto del primer proceso componente y su salida la del segundo proceso. La condición inicial y final del nuevo proceso son la condición inicial del primero y la condición final del segundo respectivamente. Las cotas inferior y superior se determinan como la suma de las cotas inferiores y superiores de los procesos componentes respectivamente. El invariante del proceso especifica que, una vez que cumple la pre-condición del primer proceso, se cumple el invariante hasta que se satisface la pos-condición del mismo en coherencia con las cotas especificadas y a partir de allí, sucede lo propio con el segundo componente con las restricciones temporales actualizadas con respecto a la ejecución del primer proceso.

Composición condicional

La composición condicional, denotada por $;_s$, es análoga a la composición secuencial con una restricción adicional que condiciona el comienzo del segundo proceso

componente a la satisfacción de una fórmula. La semántica de la composición condicional se define en términos de un nuevo proceso y la configuración de una compuerta semáforo que especifica la condición de continuación.

Definición 5.19. ***Composición condicional de procesos***

La composición condicional $p1;_s p2$ *de dos procesos* $p1$ *y* $p2$ *a través de una compuerta semáforo* $s = \langle codigo_s, P_s, S_s \rangle$*, es un proceso* $p = \langle codigo_p, nombre_p, P-in_p, P-out_p, (q_p, I_p)_{l_p}\Delta^{u_p}p_p \rangle$*, donde:*

- $codigo_p$*,* $nombre_p$ *y* $codigo_s$ *tienen sus respectivos valores para identificarlos en el sistema.*
- $P-in_p = P-in_{p1} \wedge P-out_p = P-out_{p2} \wedge P_s = P-out_{p1} = P-in_{p2}$
- $q_p = q_{p1}$
- $p_p = p_{p2}$
- $I_p = q_{p1}\mathtt{U}_{>l_{p1}}(I_{p1}\mathtt{U}_{\leq u_{p1}-l_{p1}}(p_{p1} \Rightarrow true\mathtt{U}((q_{p2} \wedge S_s)\mathtt{U}_{>l_{p2}}(I_{p2}\mathtt{U}_{\leq u_{p2}-l_{p2}}p_{p2}))))$
- $l_p = l_{p1} + l_{p2} \wedge u_p = u_{p1} + u_{p2}$

La compuerta semáforo, además de establecer la condición de continuación, oficia de intermediario entre la salida del primer proceso y la entrada del segundo ($P_s = P-out_{p1} = P-in_{p2}$). La definición del proceso p que establece la composición, es similar a la de la composición secuencial, fortaleciendo el invariante con la condición del semáforo que debe ser verdadera para que el segundo proceso pueda comenzar.

Composición Paralela

La composición paralela expresa la posibilidad de que dos procesos puedan realizar su tarea simultáneamente y de manera independiente entre sí. De manera similar a la anteriores, su semántica se define en términos de un nuevo proceso y dos compuertas, una demultiplexora y una multiplexora que se encargan de la distribución y sincronización de las entradas y salidas de los procesos, respectivamente. La notación para este tipo de composición es `[ ]( )`, donde los procesos se especifican entre corchetes y las compuertas se especifican entre paréntesis.

Definición 5.20. ***Composición paralela de procesos***

La composición paralela $[p1|p2](d_c, m_c)$ *de dos procesos* $p1$ *y* $p2$ *utilizando las compuertas demultiplexora* $d_c = \langle codigo_d, P_d, \mathcal{P}_d, F_d \rangle$ *y multiplexora* $m_c = \langle codigo_m, \mathcal{P}_m, P_m, F_m \rangle$*, es un proceso* $p = \langle codigo_p, nombre_p, P-in_p, P-out_p, (q_p, I_p)_{l_p}\Delta^{u_p}p_p \rangle$*, donde:*

- *$codigo_p$, $nombre_p$, $codigo_d$ y $codigo_m$ tienen sus respectivos valores para identificarlos en el sistema.*
- $P - in_p = P_d \wedge P - out_p = P_m$
- $q_p = q_{p1} \wedge q_{p2}$
- $p_p = p_{p1} \wedge p_{p2}$
- $I_p = (q_{p1} \wedge q_{p2}) \Rightarrow (\tau_1[(p_{p1}\mathtt{U}(p_{p1} \wedge p_{p2}))/p_{p1}] \wedge (\tau_2[(p_{p2}\mathtt{U}(p_{p1} \wedge p_{p2}))/p_{p2}]))$
- $l_p = max((l_{p1}, l_{p2})) \wedge u_p = max((u_{p1}, u_{p2}))$

En la definición del invariante, la fórmula $\tau_i[(p\mathtt{U}q)/r]$ denota el reemplazo de la condición final r en τ_i por la condición $(p\mathtt{U}q))$. Este reemplazo en las condiciones finales de cada proceso componente especifica que, cuando los procesos están habilitados para comenzar, realizan su trabajo en paralelo y luego de que uno de ellos termina, debe esperar que el segundo finalice su tarea. La compuerta demultiplexora d_c se encarga de distribuir los productos a cada uno de los procesos intervinientes. Esta distribución se encuentra especificada en la fórmula F_d. Así, por ejemplo, una posible distribución de un producto de entrada $\times(P_1, P_2)$ podría ser $P - in_{p_i} = P_i$. De manera simétrica, la compuerta multiplexora se encarga de empaquetar los productos de cada proceso componente en el producto resultante de la composición.

5.2.5 Procesos estructurados

Para la capturar la composición vertical de procesos, es decir, la posibilidad de encapsular en un proceso una estructura o composición horizontal de procesos, PPML propone los *procesos estructurados*.

Definición 5.21. ***Proceso estructurado***

Un proceso estructurado $p = \langle codigo_p, nombre_p, P - in_p, P - out_p, f - expr_p \rangle$ es un proceso atómico definido en términos de la composición horizontal de un conjunto $\{p_1, \ldots, p_n\}$ de procesos, donde:

- *$codigo_p$, $nombre_p$ tienen sus respectivos valores para identificar el proceso en el sistema.*
- *$P - in_p, P - out_p$ son las especificaciones de los productos de entrada y salida, respectivamente.*
- *$f - exp_p$ es una expresión que especifica el proceso en términos de composiciones entre los procesos $\{p_1, \ldots, p_n\}$.*

La definición de un proceso estructurado permite describir una actividad compleja del sistema en dos sentidos: (i) permitiendo la especificación explícita su comportamiento como una interacción entre una serie de actividades que la componen, y (ii) ocultando dicha composición bajo un elemento que puede ser utilizado de manera atómica.

5.3 Sintaxis y Representación gráfica de PPML

En trabajos previos sobre PPML, todos los elementos que forman parte de la especificación de un modelo son definidos formalmente. Con el objetivo de brindar un modo de especificar modelos PPML en un nivel menos riguroso, proponemos una sintaxis para los productos, procesos y compuertas. Esta sintaxis tiene dos objetivos: proveer una forma más flexible y accesible de escribir especificaciones en PPML (como en otros lenguajes de modelado de procesos de negocios) y estandarizar el lenguaje, para poder construir herramientas para el lenguaje como *parsers*, analizadores, asistentes, etc.

A continuación mostramos la sintaxis para los *productos atómicos* y *estructurados*:

⟨*Product*⟩	::=	**'Product'** ⟨*Product_name*⟩ **'{'** ⟨*Product_body*⟩ **'}'**
⟨*Product_body*⟩	::=	⟨*Product_refs*⟩ **','** ⟨*Product_atts*⟩ | ⟨*Product_atts*⟩
⟨*Product_refs*⟩	::=	⟨*Product_ref*⟩ **','** ⟨*Product_refs*⟩ | ⟨*Product_ref*⟩
⟨*Product_atts*⟩	::=	⟨*Product_att*⟩ **','** ⟨*Product_atts*⟩ | ⟨*Product_att*⟩
⟨*Product_ref*⟩	::=	⟨*Product_ref_name*⟩ ⟨*Product_ref_ID*⟩
⟨*Product_att*⟩	::=	⟨*Product_direct_att* ⟩ | ⟨*Product_derived_att* ⟩
⟨*Product_direct_att* ⟩	::=	⟨*Product_att_type* ⟩ ⟨*Product_att_ID*⟩
⟨*Product_derived_att* ⟩	::=	⟨*Product_att_type* ⟩ ⟨*Product_att_ID*⟩ **'='** ⟨*Expression*⟩
⟨*Product_att_type* ⟩	::=	**'int'** | **'boolean'**
⟨*Expression*⟩	::=	⟨*Expression*⟩ ⟨*Binary_op*⟩ ⟨*Expression*⟩
	|	⟨*Unary_op*⟩ ⟨*Expression*⟩
	|	⟨*Product_att_ID*⟩ | ⟨*Product_ref_name*⟩**'.'**⟨*Product_att_ID*⟩
	|	**'('**⟨*Expression*⟩**')'** | ⟨*NAT_const*⟩ | ⟨*BOOLEAN_const*⟩
⟨*Binary_op*⟩	::=	**'+'** | **'−'** | **'∗'** | **'/'** | **'=='** | **'!='** | **'and'** | **'or'**
⟨*Unary_op*⟩	::=	**'−'** | **'not'**

La sintaxis para *productos compuestos* se resume a una representación como tuplas o conjuntos de productos. Cabe recordar que este tipo de productos no representan un elemento directo del modelo, sino que nos brindan una técnica para abordar la restricción de simple producto de entrada y salida de los procesos.

⟨*Composite_Product*⟩	::=	⟨*Package_Product*⟩ | ⟨*Choice_Product*⟩

⟨*Package_Product*⟩	::=	**'['** ⟨*Product_refs*⟩ **']'**
⟨*Choise_Product*⟩	::=	**'⟨'** ⟨*Product_refs*⟩ **'⟩'**

La descripción de procesos está conformada por la especificación de la interfaz, esto es, qué producto recibe el proceso como entrada y cuál es el producto que éste produce, junto con la especificación, al estilo de *contratos* [68], de las expresiones correspondientes a la pre-condición, invariante y pos-condición. La expresión correspondiente a la pos-condición especifica cómo el producto de salida es construido a partir del producto de entrada. Además se determinan los límites temporales correspondientes a la realización de la tarea vinculada al proceso.

⟨*Process*⟩	::=	**'Process'** ⟨*Process_name*⟩ **'{'** ⟨*Process_body*⟩ **'}'**
⟨*Process_body*⟩	::=	**'input:'** ⟨*Product_ref*⟩ **','**
		'output:' ⟨*Product_ref*⟩ **','**
		'invariant:' ⟨*Expression*⟩ **','**
		'requires:' ⟨*Expression*⟩ **','**
		'ensures:' ⟨*Expression*⟩ **','**
		'l_time:' NAT **','** **'u_time:'** NAT

Las compuertas se describen especificando, según su función, el/los producto/s de entrada, demultiplexor/multiplexor respectivamente, y de manera dual, su salida. Además, se detalla mediante una expresión la relación entre sus interfaces, es decir, cómo se compactan o distribuyen los productos según su finalidad.

⟨*Gate*⟩	::=	**'Gate'** ⟨*Gate_name*⟩ **'{'** ⟨*Gate_body*⟩ **'}'**
⟨*Gate_body*⟩	::=	⟨*Multiplexer_body*⟩ | ⟨*Demultiplexer_body*⟩
⟨*Multiplexer_body*⟩	::=	**'input:'** ⟨*Multiple_Product_refs* ⟩ **','**
		'output:' ⟨*Simple_Product_ref*⟩ **','**
		'map:' ⟨*Expression*⟩
⟨*Demltiplexer_body*⟩	::=	**'input:'** ⟨*Simple_Product_ref*⟩ **','**
		'output:' ⟨*Multiple_Product_refs*⟩ **','**
		'map:' ⟨*Expression*⟩
⟨*Simple_Product_ref*⟩	::=	⟨*Product_ref*⟩ | ⟨*Composite_Product_ref*⟩
⟨*Multiple_Product_ref* ⟩	::=	⟨*Simple_Product_ref*⟩
	|	⟨*Simple_Product_ref*⟩ **','** ⟨*Multiple_Product_ref* ⟩

5.3.1 Representación Gráfica

Como lo mencionamos en la Sección 3.3, los lenguajes para la descripción de workflows, en su gran mayoría, proveen una descripción gráfica de sus modelos. Este tipo de descripciones gráficas propician su utilización, aportando una visión integral arquitectónica (posiblemente en diferentes niveles de abstracción) de un modelo de

workflow. PPML no es la excepción en este sentido. Además de la sintaxis para la especificación de los productos y el detalle *funcional* de cada proceso, los modelos de PPML pueden describirse gráficamente de la siguiente manera.

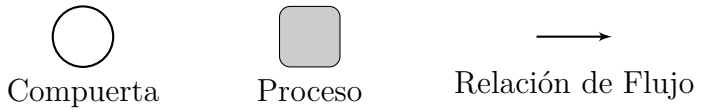


Figura 5.1: Representación gráfica de los componentes de un modelo PPML

Las compuertas se representan mediante un círculo con su identificador, los procesos con un rectángulo con su identificador, y el flujo de control, es decir, la composición de estos elementos, mediante flechas, las cuales se anotan con los nombres (tipos) de productos que comunican.

5.3.2 Simple caso de estudio

A continuación ilustramos la aplicación de PPML para modelar un caso de estudio simple, en el cual se puede apreciar el enfoque que propone este lenguaje en la descripción de sus modelos.

El caso de estudio describe una versión simplificada de una línea de producción de una empresa que produce *single-board computers (SBC)*[1]. Particularmente, se especifica el proceso de ensamblado del *microprocesador* y la *memoria* en la *MotherBoard*.

El sistema de ensamblado comienza con un proceso $assem_1$ que recibe una MotherBoard y un Procesador, los ensambla en un intervalo de entre 5 y 10 unidades de tiempo. Seguidamente el proceso $assem_2$ recibe el producto de salida del proceso anterior y una Memoria, y los ensambla con la misma restricción temporal. Luego se prueba el funcionamiento del Procesador y de la Memoria ensamblados de forma paralela por los procesos $test_1$ y $test_2$, respectivamente, cuyos tiempos se estiman entre 3 y 5 unidades.

La representación gráfica del modelo PPML que especifica este proceso se muestra en la Figura 5.3.2. Cabe aclarar que en la misma, por cuestiones de espacio, las anotaciones sobre las flechas son una abreviación del tipo de producto cuya transferencia éstas representan. Por ejemplo, utilizamos M para referirnos a $MotherBoard$, $Proc$ para $Processor$ y Mem para $Memory$.

[1]SBC: es una computadora completa en un sólo circuito. Su diseño se centra en un sólo microprocesador con la RAM, E/S y todas las demás características de un computador funcional en una

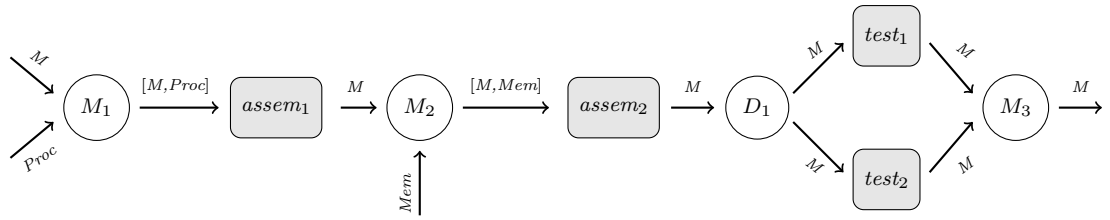


Figura 5.2: Diagrama PPML del caso de estudio.

La especificación de los productos se detalla a continuación:

```
Product Memory{
       int memory_size,
       boolean tested
}
Product Processor{
       int weight,
       boolean tested
}
Product MotherBoard{
       Processor id_Processor,
       Memory id_Memory,
       int weight,
       int esambled_weight = weight + id_Processor.weight,
       int memory_size = id_Memory.memory_size
       boolean tested = id_Processor.tested and id_Memory.tested
}
```

La especificación de los procesos $assem_1$, $assem_2$ y $test_1$ es la siguiente:

```
Process Assem1{
       input:[MotherBoard M, Processor Proc],
       output: MotherBoard M,
       invariant: M.id_Memory = null,
       requires: M.id_Processor = null,
       ensures: M.id_Processor = Proc,
       l_time: 5, u_time: 10
}
Process Assem2{
       input:[MotherBoard M, Memory Mem],
```

sola tarjeta que suele ser de tamaño reducido

```
11        output: MotherBoard M,
12        invariant: M.id_Processor != null,
13        requires: M.id_Memory = null,
14        ensures: M.id_Memory = Mem,
15        l_time: 5, u_time: 10
16 }
17 Process Test1{
18        input: MotherBoard M,
19        output: MotherBoard M,
20        invariant: True,
21        requires: M.id_Processor != null,
22        ensures: M.id_Processor.tested = True,
23        l_time: 3, u_time: 5
24 }
```

Finalmente, a continuación, mostramos la especificación para M_2 y D_1:

```
1 Gate M2{
2        input: MotherBoard M, Memory Mem
3        output: [MotherBoard M, Memory Mem]
4 }
5 Gate D1{
6        input: MotherBoard M
7        output: MotherBoard M, MotherBoard M
8 }
```

Cabe destacar algunas abreviaturas sintácticas tanto en la especificación de los procesos como de las compuertas. Utilizamos el mismo identificador en la especificación de productos de entrada y salida, para hacer referencia a la misma instancia de un producto.

5.4 Sumario

Hemos analizado el estado del arte en el ámbito de métodos formales para la especificación de procesos de negocios. Particularmente, notamos que la gran mayoría de lenguajes para tal fin no provee elementos dentro del lenguaje que permitan modelar productos (entidades del modelo).

Retomamos un formalismo para la especificación de procesos de negocios y lo adaptamos para facilitar y extender su aplicación. Focalizamos nuestro interés en investigar las ventajas de contar con "productos" como elementos del lenguaje. Propusimos una sintaxis para la especificación de modelos PPML e ilustramos a través

de su descripción gráfica, un simple caso de estudio. Actualmente estamos trabajando en un prototipo de entorno para asistir en la creación de modelos PPML, para luego integrar en él diferentes herramientas de análisis para los mismos. Creemos que este tipo de especificaciones de procesos de negocios aporta una perspectiva alternativa a las encontradas actualmente en los lenguajes de workflows y brinda soporte para realizar diferentes tipos de análisis sobre sus modelos. Por ejemplo, creemos que puede ser adecuado para modelar sistemas *transaccionales*, esto es, poder especificar modelos en los cuales los procesos que intervienen en el mismo realizan cambios en los productos que manipulan. Para ello pueden combinarse el concepto de producto junto con técnicas formales como por ejemplo *compensaciones* [20].

Capítulo 6

Verificación de Propiedades de Productos

6.1 Introducción

Uno de los objetivos principales de este trabajo es el de brindar un marco teórico formal para la verificación de propiedades de procesos de negocios. En ese sentido, como lo mencionamos anteriormente, estamos interesados en poder describir, por medio de especificaciones, propiedades que capturen situaciones del comportamiento de un proceso.

En el capítulo anterior analizamos la necesidad de contar con descripciones explícitas de los elementos que forman parte del flujo entre los procesos que especifican un procesos de negocio y presentamos un lenguaje formal, denominado PPML, el cual permite especificar procesos contando con dichas descripciones. PPML además incorpora en la especificación de sus procesos, cotas temporales que permiten modelar restricciones del tiempo de ejecución que cada procesos debe respetar.

En pos de alcanzar los objetivos mencionados, en este capítulo nos centramos en la especificación y verificación de propiedades de modelos PPML. Para ello, utilizamos un lenguaje para describir propiedades de dichos modelos que luego serán verificadas de manera automática. Un punto interesante que marcó el rumbo de esta etapa fue el de disponer de un lenguaje que permita especificar propiedades que capturen situaciones establecidas por la interacción entre procesos y productos, tomando ventaja de una de las principales características relevantes de PPML. Además, dada la existencia de restricciones temporales como parte de los modelos PPML, pretendemos describir propiedades que puedan incorporar referencias a las mismas.

Análogamente a la propuesta del Capítulo 4, en este Capítulo presentamos la uti-

lización de una lógica temporal para la especificación de propiedades sobre modelos PPML y la verificación de las mismas la efectuamos mediante la técnica de *model checking*. Debido a la presencia de restricciones de tiempo real en los modelos, para posibilitar la verificación de propiedades que involucren este tipo de especificaciones, utilizamos herramientas para la verificación de sistemas de tiempo real. Entre las herramientas más utilizadas para tal fin, podemos mencionar Kronos [19] y Uppaal [13]. Por su aplicación y versatilidad a nuestros propósitos elegimos como herramienta para la verificación a Uppaal. Como lo detallamos en la Subsección 2.4.1, Uppaal es una herramienta basada en la teoría de autómatas temporizados para la descripción de sus modelos y provee un subconjunto de Computational Tree Logic (CTL) [23,64,65] como lenguaje de para la especificación de propiedades a verificar.

Presentamos una traducción de modelos PPML a especificaciones Uppaal. Con esta traducción, disponemos de la flexibilidad de la lógica CTL para especificar propiedades que luego serán verificadas de manera automática. Para ilustrar la traducción, utilizaremos como ejemplo, el simple caso de estudio presentado en la Subsección 5.3.2, simplificando las propiedades de los productos involucrados.

Finalmente, describimos una semántica alternativa para PPML basada en π-Cálculo con el objetivo de analizar en los modelos del lenguaje el aspecto de movilidad que dicho formalismo provee.

6.2 Caracterización de PPML mediante Uppaal

Con el objetivo de utilizar Uppaal como herramienta para la verificación de propiedades de procesos de negocios especificados en PPML, detallamos una traducción *automática* de modelos PPML a Uppaal. Mediante esta traducción disponemos del poder de la lógica CTL para expresar propiedades que podemos analizar. De manera general describimos un modelo Uppaal como un conjunto de esquemas que especifican, mediante un autómata temporizado, elementos del sistema cuyas instancias conforman la configuración del sistema (ver Subsección 2.4.1).

La idea general de traducción consiste en generar por cada clase de *producto* del modelo PPML un esquema Uppaal que describe sus estados dentro del sistema. De manera similar, por cada *proceso* especificado en PPML, se construye un esquema que describe su comportamiento. La semántica de las *compuertas* se integra dentro de los esquemas de productos y procesos. La comunicación entre productos y procesos se realiza mediante variables compartidas (globales) y mensajes de sincronización.

Este diagrama de traducción preserva la modularización del modelo PPML y nos permite variar de manera directa la cantidad de instancias de productos, simplemente indicando su cantidad. Debido a la presencia de productos como elementos

explícitos del modelo, la representación de los mismos refleja de manera exhaustiva su ciclo de vida dentro del sistema.

Para ayudar a la comprensión de la traducción, en cada paso ensayamos la misma con la especificación en PPML del proceso correspondiente al caso de estudio propuesto, que se muestra en la Figura 5.3.2.

6.2.1 Traducción de Productos

PPML presenta tres clases de productos: atómicos, compuestos y estructurados. Los productos atómicos son la representación básica de un objeto que forma parte del sistema, en la cual se especifican las características relevantes del mismo. Los productos compuestos sólo forman parte del aspecto técnico del lenguaje para abordar restricciones que impone el formalismo subyacente. Finalmente, los productos estructurados especifican composiciones de productos que reflejan la complejidad estructural de un objeto del sistema.

A continuación describimos la representación de productos de un modelo PPML en Uppaal. Presentamos en primera instancia la traducción de productos compuestos y luego abordamos la traducción de los productos atómicos y estructurados.

Traducción de productos compuestos

Los productos compuestos que presenta PPML son los productos empaquetados y selectivos. Este tipo de productos no son contraparte de ningún elemento del dominio del sistema, sino una estructura propia del lenguaje. Las compuertas (multiplexores y demultiplexores) son los únicos elementos de un modelo PPML que pueden "crear" este tipo de productos. Es decir, su presencia se encuentra asociada a la salida o entrada de estas compuertas.

La traducción de este tipo de productos se realiza generando en las declaraciones globales del sistema un arreglo (secuencia indexada) de identificadores por cada producto compuesto asociado a una compuerta. Así, para la traducción del producto compuesto empaquetado $[M, Proc]$ del ejemplo propuesto, que se corresponde con la salida de la compuerta multiplexora M_1, generamos el siguiente código dentro de las declaraciones globales del modelo Uppaal.

```
1 //Composite products declarations
2 int gate_M1_out[2] //[0 MotherBoard, 1 Processor]
```

Como se puede observar en la línea2, por cada producto compuesto asociado a una compuerta se genera un arreglo cuyo tamaño se corresponde con la cantidad de elementos que configuran dicho producto compuesto. Es decir, para un producto

compuesto empaquetado de n componentes o para un producto selectivo sobre n posibles opciones, se genera una arreglo de n identificadores. El tipo utilizado para los identificadores de productos es el tipo entero (`int`) reservando el número `0` como referencia al producto *nulo* para la descripción de productos parciales. La identificación unívoca dentro del modelo de un producto se consigue con la combinación de estos identificadores junto con la clase (nombre) del mismo.

Para el caso de productos empaquetados, el índice del arreglo identifica el número de componente. Para los productos selectivos, el índice se utiliza como referencia de la clase de las posibles opciones.

La diferencia entre los dos tipos de productos compuestos se refleja en las condiciones previas en la traducción de los procesos. En caso de productos empaquetados, la condiciones exigen la *conjunción* (`and`) de los componentes (identificadores no nulos) y para los selectivos una *disyunción* (`or`) de los mismos.

Traducción de Productos Atómicos y Estructurados

El diseño de la traducción para cada clase de producto atómico o estructurado se presenta como un esquema Uppaal, en el cual se representan los diferentes estados del mismo dentro del modelo PPML. Para describir la traducción de estos productos, asumimos sin pérdida de generalidad, que a cada proceso le precede una compuerta. Dado que para las composiciones condicional y paralela las compuertas forman parte de su definición, en caso de composición secuencial, la asumimos como una composición condicional cuya condición (fórmula de la compuerta semáforo subyacente) es `true`.

Podemos gestionar la traducción de estos productos como la composición entre la representación de la información que los mismos capturan y la caracterización de sus posibles estados dentro del sistema. La diferencia entre las codificaciones de los productos compuestos y estructurados sólo radica en que los segundos pueden contener como atributos referencias a otros productos y, en caso de reflejar productos parciales o incompletos, que dichas referencias sean *nulas*. Como se describe en la Subsección 5.2.1, los productos atómicos describen elementos de dominio del sistema por medio de atributos que capturan la información relevante de los mismos. Para representar esta información en Uppaal utilizamos definiciones de registros. En caso de productos estructurados, dicha estructura se verá reflejada al tener como campos del registro referencias a otros registros.

Para especificar el dinamismo del producto dentro del sistema, generamos un esquema por cada clase de producto que describe el comportamiento (modificaciones) que alteran sus características. El autómata de dicho esquema representa, mediante sus estados, el ciclo de vida del producto a lo largo del sistema. Bajo la suposición

de compuertas previas a cada proceso, los estados del autómata capturan las situaciones previas y posteriores del producto cuando es manipulado por algún proceso del sistema.

A continuación detallamos la traducción para cada clase de producto PPML. Dentro de las declaraciones globales del modelo Uppaal generamos:

- Una definición de rango $0..n$ de enteros que oficia de lista de identificadores de las instancias de productos de esta clase. Como mencionamos con anterioridad, simplemente cambiando el valor n podemos alterar la cantidad de instancias de los productos en el modelo a verificar. Cabe destacar que el identificador 0 se reserva para representar a productos nulos, por ejemplo para especificar que cierto producto estructurado aún no está completamente conformado, se asigna el identificador 0 a las partes (referencias a otros productos) que aún no tiene asociadas.

- Una definición de tipo registro `struct` que representa la información del producto. Esto es, el registro contiene la definición de un campo por cada atributo del producto, aún para los atributos calculados. Los valores de estos últimos se actualizan luego de cada modificación que el producto pueda recibir dentro del sistema.

  En caso de productos estructurados, por cada uno de sus componentes (productos), el registro contiene un campo que referencia a su componente asociado por medio de su identificador. En caso de productos parciales, se utiliza el identificador nulo 0 para aquellos campos que especifiquen componentes que aún no han sido asociados.

- Finalmente, declaramos un arreglo cuyos componentes son del tipo de registro definido en el punto anterior y cuya longitud es igual a la cantidad de instancias especificadas en la definición del rango del primer punto. Este arreglo se utiliza para mantener la información particular de cada instancia del producto especificado.

Además, por cada clase de producto generamos un esquema con su *nombre* conteniendo:

- Una variable de tipo reloj `clock` que representa el tiempo relativo del producto. Esta variable puede ser utilizada para realizar consultas de propiedades temporales de cada instancia de producto.

- Como parte de las declaraciones locales al esquema, generamos un método que se encarga de actualizar los atributos calculados en función de otros atributos del producto.

- Como parámetro, el esquema tiene asociada una *constante* que oficiará de código unívoco dentro de su clase, permitiendo identificar cada instancia del producto.
- El autómata correspondiente al esquema contiene los siguientes *estados*:
  - un estado inicial calificado como *urgente*.
  - un estado por cada compuerta o proceso que manipula el producto.
  - un estado final que describe el final de modificaciones que el producto puede recibir en el sistema.
- Considerando la existencia de una compuerta antes de cada proceso, las *aristas* del autómata se generan de la siguiente manera, considerando dos posibles escenarios:
  - Aristas con origen en un estado correspondiente a una *compuerta* y con destino en un estado correspondiente a un *proceso*: Estas aristas caracterizan la disposición del producto para formar parte del proceso siguiente, puesto a consideración por la compuerta correspondiente al mismo. Para ello, se etiquetan estas aristas con la siguiente información: como *guarda* se genera una condición que consulta si la compuerta (arreglo que modela su producto de salida) está vacía en la posición correspondiente,(actual identificador igual a 0). Luego como *modificaciones* se actualiza dicha posición con el identificador del producto colocándolo en posición de poder ser procesado cuando todas las condiciones de proceso se hayan alcanzado.
  - Aristas con origen en un estado correspondiente a un *proceso* y con destino en un estado correspondiente a una *compuerta*: Estas aristas reflejan los efectos que produce la ejecución de un proceso en el producto. Las mismas se etiquetan con la siguiente información: la *sincronización* especifica la espera por el canal correspondiente al evento que determina que el proceso tuvo lugar. Como este canal es de difusión, la *guarda* certifica que el producto se corresponde con identificador. Finalmente como *modificaciones*, se actualizan sus atributos calculados invocando la función local `update()`.

  Cabe destacar que cuando un producto puede ser procesado de manera paralela por dos procesos, en el autómata se generan todas las posibles combinaciones (*interleavings*) que modelan, mediante concurrencia, dicha situación.

Finalmente, dentro de las declaraciones que describen el sistema Uppaal, generamos una instancia del esquema especificado arriba por cada instancia del producto indicada en la definición del rango.

Para ilustrar la traducción de los productos, a continuación mostramos el modelo parcial Uppaal generado a partir de la especificación del producto *MotherBoard* del ejemplo propuesto. Comenzamos con las declaraciones globales del modelo Uppaal y luego con el esquema particular del producto.

```
1  //Codes types definitios for products with and without null descriptor
2  typedef int[1,1] n_motherboards;
3  typedef int[0,1] n_motherboards0;
4
5  //Products structure definitions
6  typedef struct{
7         n_processors0 id_Processor;
8         n_memories0 id_Memory;
9         bool tested;
10 } TMotherBoard;
11
12 //Array definitions for Products information handling
13 TMotherBoard DataMotherBoards[n_motherboards];
```

En las líneas 1 y 2, se definen los rangos (cantidad de instancias) para el producto. Cabe destacar que se definen dos rangos, uno que incluye al identificador 0 que corresponde al el producto *nulo* y otro que no lo incluye. El rango con descriptor nulo se utiliza para caracterizar referencias a la correspondiente clase de producto dentro de especificaciones de producto estructurados. Por ejemplo, en la línea 7 y 8, como parte de la descripción del producto `MotherBoard` se definen referencias a un procesador y memoria con la posibilidad de que éstos sean nulos (cuando aún no se han ensamblado). El rango sin presencia de dicho descriptor se utiliza como referencia en la especificación de características de las instancias de productos. Por ejemplo, en la línea 13 se utiliza para describir el tamaño del arreglo que alojará la información de cada instancia del producto en cuestión. En las líneas 6-10 se encuentra la definición correspondiente al tipo que captura la información del producto. En este caso particular, como se trata de un producto estructurado, contiene dos campos que especifican la referencia a sus productos componentes, además de un campo calculado `tested` cuyo valor indica si ambos componentes fueron probados luego de ser ensamblados. En la línea 13 se declara el arreglo que se encarga de alojar la información de cada instancia del producto.

En la Figura 6.1 se muestra el autómata temporizado generado a partir de la especificación del producto *MotherBoard*. Como lo detallamos anteriormente, los

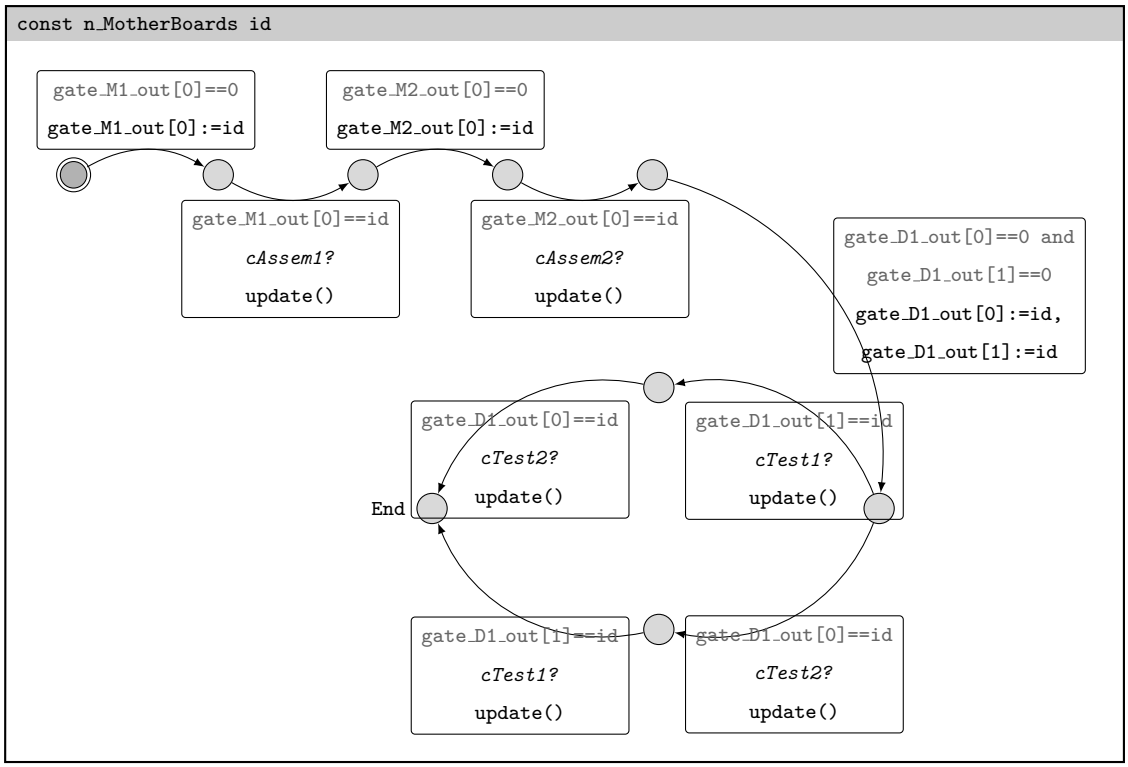


Figura 6.1: Autómata temporizado correspondiente al esquema del producto *MotherBoard*

autómatas modelan las diferentes etapas de transformación del producto y contienen al menos tantos estados como compuertas y procesos atraviesen los mismos en sus ciclos de vida dentro del sistema. En particular, el autómata correspondiente al producto *MotherBoard* contiene, además del inicial, ocho estados. De manera intercalada, las aristas modelan situaciones del producto previas a una compuerta y situaciones correspondientes a cuando el producto es procesado. Como puede observarse en el último estado del autómata, los estados pueden etiquetarse con un nombre `End` para luego ser utilizados en la especificación de propiedades. Esta situación la analizaremos en la Sección 6.3.

Las primeras aristas describen la competencia de cada instancia por participar de un proceso necesario para su propósito en el sistema. Por ejemplo, en caso de la primera arista, si la compuerta `M1` que empaqueta una Motherboard con un Procesador tiene disponible (es 0) el puerto correspondiente a la Motherboard, entonces modifica el valor de dicho puerto con el identificador de la instancia del producto

para luego ser procesado.

Las aristas que describen cuándo un producto es procesado esperan por el correspondiente canal de notificación, y en caso de coincidir con la instancia de producto procesada, aquella que forma parte de la compuerta anterior actualiza sus valores calculados mediante la invocación a la función `update`. Cabe destacar que, como analizamos en la Subsección 6.2.2, las modificaciones que alteran el estado de las instancias de los productos son especificadas en el comportamiento de los procesos que las manipulan.

Como puede observarse en el autómata del ejemplo, sobre el final (siguiendo el sentido de las aristas) existe un bifurcación que confluye en el estado final del mismo. Esta situación modela, de manera concurrente, las posibles formas de terminación de los procesos `test1` y `test2` que, en correspondencia con el modelo PPML (Figura 5.3.2), pueden probar el procesador y la memoria ensamblados de manera paralela.

El código local al esquema correspondiente al producto MotherBoard es el siguiente:

```
clock x;
void update(){
//MotherBoard Update;
  if ((DataMotherBoards[id].id_Memory !=0) and
    (DataMotherBoards[id].id_Processor !=0)) {
    DataMotherBoards[id].tested:=
      DataMemories[DataMotherBoards[id].id_Memory].tested and
      DataProcessors[DataMotherBoards[id].id_Memory].tested;
  }
  else{
    DataMotherBoards[id].tested:=false;
  }
}
```

En caso del producto MotherBoard, la función `update`, que actualiza los campos calculados, se encarga de modificar el valor del atributo `tested` dependiendo de los respectivos valores `tested` correspondientes al procesador y memoria que lo componen. Como puede observarse el acceso a la información de cada instancia de producto se logra mediante la *indirección* obtenida a partir de los identificadores de instancias de productos en el correspondiente arreglo.

Finalmente, en las declaraciones del sistema se declaran tantas instancias como fueran especificadas, y se componen las mismas como parte del sistema:

```
MotherBoards(const n_motherboards id) = MotherBoard(id);

```

```
3 system MotherBoards, ... ;
```

6.2.2 Traducción de Procesos

De manera similar a la de los productos, la traducción de los procesos consta de declaraciones globales del modelo Uppaal, un esquema particular por cada proceso y la especificación del mismo como parte del sistema.

Como parte de las declaraciones globales, por cada proceso en PPML se genera:

- Un canal de difusión (*broadcast*) mediante el cual el proceso indica que su ejecución ha ocurrido.
- Un arreglo que modela la compuerta previa que se encarga de la logística de los productos que el proceso manipula. El arreglo es de tipo entero y contiene los valores de los identificadores que el proceso manipula. Recordemos que si bien en PPML los procesos sólo tienen un producto de entrada, éste puede ser un producto compuesto (empaquetado).

El esquema correspondiente al proceso se conforma de la siguiente manera:

- Una declaración de una variable reloj que es utilizada para controlar la demora y límites temporales del proceso.
- Una función que especifica la transformación que el proceso produce en su producto de entrada para retornarlo como salida. Como lo mencionamos previamente, el acceso a la información de los productos, esto es, los valores de sus atributos, se realiza por medio de indirecciones utilizando como índice el identificador del producto correspondiente. En caso de productos estructurados, para modificar alguno de sus componentes, se utiliza como índice el identificador almacenado en el campo correspondiente a dicho componente.
- El autómata correspondiente al esquema contiene los siguientes estados:
  - un estado inicial o de proceso ocioso (*idle*) que refleja la situación en la cual el proceso espera por la llegada de los productos que necesita para ejecutarse.
  - un estado que indica que el proceso se encuentra listo para comenzar (*ready*). Esto es, todos los productos y condiciones requeridos por el proceso están garantizados.

  - un estado *final* que captura la situación de pos proceso, es decir, cuando el mismo ha finalizado su ejecución. Este estado es de tipo urgente ya que cuando el proceso se encuentra en el mismo ya ha finalizado su ejecución y el tiempo en el mismo no debe tenerse en cuenta.

- Las aristas del autómata se configuran de la siguiente manera:
  - una arista del estado *idle* al estado *ready* anotada con la guarda que garantiza que los productos se encuentran en condiciones de ser procesados. Esto es, el arreglo de la compuerta previa al proceso contiene identificadores distintos de 0. Y como modificaciones, esta arista se encarga de *reiniciar* (poner en 0) el reloj del proceso.
  - una arista del estado *ready* al estado *final* con la siguiente información: la guarda establece el cumplimiento de las cotas temporales, es decir, el valor del reloj del procesos (inicializado en el estado anterior) debe ser mayor o igual que el límite inferior y menor o igual que el límite superior del proceso. Como sincronización, esta arista escribe (difunde) sobre el canal correspondiente al proceso informando a los productos intervinientes que han sido procesados. Finalmente, como modificación se captura el comportamiento del proceso mediante la invocación a la función que especifica la transformaciones que produce el proceso en los productos que manipula.
  - para finalizar, una arista con origen en el estado *final* y destino en el estado *idle* que se encarga de inicializar el proceso para una nueva ejecución. Esta inicialización simplemente limpia, les asigna 0, a los valores correspondientes a los identificadores de los productos de la compuerta previa al proceso.

- Dentro de la declaración del sistema, se genera una instancia correspondiente al proceso, la cual forma parte de la composición que define el modelo del sistema.

A modo de ejemplo, consideremos el proceso `Assem1` del caso de estudio propuesto. Este proceso recibe como entrada un producto empaquetado compuesto por una MotherBoard y un Procesador, y entre 5 y 10 unidades de tiempo retorna como salida la MotherBoard con el Procesador ensamblado. Las declaraciones globales generadas para el mismo son:

```
1 //Processes channels definitions
2 broadcast chan cAssem1;
3
```

```
4 //Gates arrays definitions
5 int gate_M1_out[2]; //(0 MotherBoard, 1 Processor)
```

En la línea 2 se define el canal por el cual el proceso difunde que ha finalizado su ejecución. En la línea 4 se encuentra la declaración del arreglo correspondiente a la compuerta M_1, previa al proceso, que se encarga de empaquetar una MotherBoard y un Procesador para formar el producto de entrada del proceso en cuestión. Como mencionamos, estos arreglos son de tipo entero y sus valores corresponden a identificadores de productos.

El esquema correspondiente al producto *Assem1* está conformado por el autómata de la Figura 6.2 y las siguientes declaraciones locales:

```
1 clock x;
2
3 void pAssem1(){
4  //MotherBoard Update;
5  DataMotherBoards[bMultiplexer1[0]].id_Processor:=bMultiplexer1[1];
6 }
```

En la línea 1 se declara una variable reloj que es utilizada para medir el tiempo que demora el proceso en cada ejecución. Seguido, en las líneas 3-6, tiene lugar la función que captura el comportamiento del proceso. En este caso particular, `pAssem1` especifica la transformación que realiza este proceso sobre los productos que manipula. Particularmente, `Assem1` se ocupa de ensamblar un procesador como parte de una motherboard.

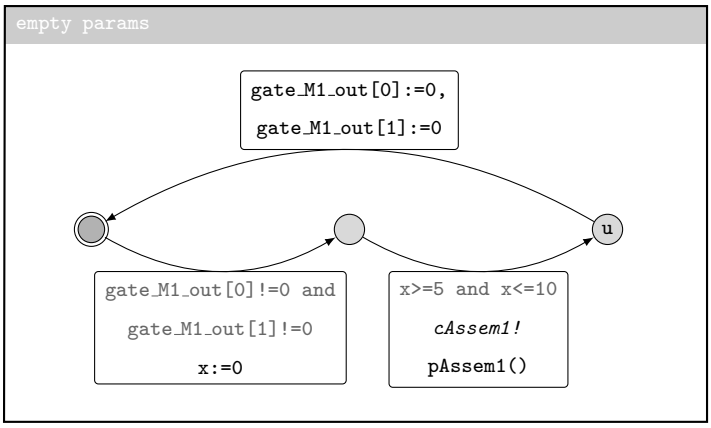


Figura 6.2: Autómata temporizado correspondiente al esquema del proceso *Assem1*

El autómata que modela los estados de proceso `Assem1`, siguiendo la traducción

propuesta, consta de tres estados: *idle*, *ready* y *final*. La arista de *idle* a *ready* especifica la espera por los productos (motherboard y procesador) que este proceso manipula y que son gestionados por la compuerta `M1`. Además, una vez que estos productos se encuentran disponibles para su procesamiento, se inicializa el reloj del proceso. Seguidamente, la arista con origen en el estado *ready* y con destino en *final*, que modela la ejecución del proceso, se encuentra anotada con las restricciones temporales como guarda, la invocación a la función `pAssem1` que detalla su comportamiento y la comunicación a los productos que manipula escribiendo en el canal `cAssem1!`. Finalmente, mediante la arista *final*→*idle*, se reestablecen los valores para que el proceso pueda volver a ejecutarse nuevamente con una nueva entrada.

6.3 Especificación de Propiedades

Con el objetivo de verificar propiedades sobre modelos PPML, una vez traducidos en términos de autómatas temporizados como lo describimos en la sección anterior, podemos emplear la herramienta Uppaal para analizarlos. De manera análoga a la propuesta del Capítulo 4, utilizamos una lógica temporal como lenguaje de especificación de propiedades temporales. Particularmente, aquí empleamos, como lenguaje de especificación de propiedades de modelos PPML la Computational Tree Logic (CTL) [23, 64, 65]. Uppaal utiliza esta lógica para la especificación de propiedades imponiendo como único límite, sólo un cuantificador de caminos por fórmula a verificar. Pese a esta limitación, podemos caracterizar un amplio espectro de propiedades interesantes sobre los modelos PPML. Como ventaja, dado que Uppaal provee la especificación del tiempo en sus modelos, nos permite analizar, de manera transparente, las restricciones temporales de los procesos. Esta característica nos permite incorporar, en las propiedades a verificar, referencias a variables que capturan la noción del tiempo como información del sistema.

Cabe destacar que, a diferencia del análisis propuesto en el Capítulo 4 y centrándonos en las ventajas de PPML con respecto a otros lenguajes para el modelado de procesos de negocios, el principal aporte de nuestra propuesta es la posibilidad de especificar y analizar propiedades de modelos donde los principales actores en dichas propiedades son productos del sistema.

Para especificar propiedades del sistema, la sintaxis que provee Uppaal para la especificación de propiedades es la de Computational Tree Logic (CTL), detallada en la Subsección 2.3.2. Como allí se describe, esta lógica propone un razonamiento sobre diferentes posibles futuros para cada instante de tiempo. Por tal motivo, sus fórmulas comienzan con cuantificadores de caminos (`A` o `E`), que especifican el espectro de la fórmula sobre dichos futuros.

A continuación y a modo de ejemplo, consideramos las siguientes propiedades, sobre el modelo PPML propuesto, que fueron verificadas y que muestran la flexibilidad y versatilidad de nuestro enfoque para caracterizar propiedades de procesos de negocios especificados en PPML.

1. *Los procesadores y la memorias no pueden ser compartidos, es decir, sólo pueden ensamblarse en una única Motherboard.*

```
A[] forall (i:n_motherboards) forall (j:n_motherboards)
i != j imply ((MotherBoards(i).End and MotherBoards(j).End)
imply (DataMotherBoards[i].id_Memory != DataMotherBoards[j].id_Memory
and DataMotherBoards[i].id_Processor != DataMotherBoards[j].id_Processor))
```

2. Toda MotherBoard del sistema que llegue al final de la línea de producción, tiene asociados un procesador y una memoria probados.

```
A[] forall (i:n_motherboards) (MotherBoards(i).End imply
DataMotherBoards[i].tested)
```

3. La producción de un SBC puede obtenerse en menos de 14 unidades de tiempo. Cabe notar que para que esto suceda, la fase de prueba debe ser realizada en paralelo.

```
E<> exists(i:n_motherboards)(MotherBoards(i).End and
MotherBoards(i).x<=13)
```

4. Existe la posibilidad de que todas las MotherBoards puedan ser probadas con resultado exitoso.

```
E<> forall(i:n_motherboards)(DataMotherBoards[i].tested)
```

Estas propiedades fueron verificadas en una computadora personal con un procesador Intel Core 2 Duo a 2.2 Ghz, 2 Gb. 667 Mhz DDR2 SDRAM de memoria bajo un sistema operativo basado en Unix. En la siguiente tabla se detalla la información para cada una de las pruebas realizadas con diferentes cantidades de instancias de productos:

- Prueba 1: dos instancias de cada producto, es decir, dos motherboard, dos procesadores y dos memorias.
- Prueba 2: tres instancias de cada producto.

- Prueba 3: cuatro instancias de cada producto.
- Prueba 4: tres instancias del producto motherboard, y cinco instancias de procesadores y memorias.
- Prueba 5: seis instancias de cada producto.

	Prueba 1	Prueba 2	Prueba 3	Prueba 4	Prueba 5
1)	0,231 s	4,651 s	22,093 m	2,694 m	-
2)	0,013 s	2,583 s	14,371 m	2,413 m	-
3)	0,001 s	0,792 s	1,642 s	1,931 s	20,053 s
4)	0,001 s	3,970 s	13,339 m	6,017 m	-

En la tabla, (s) indica segundos, (m) minutos y (-) indica que el proceso de verificación se detuvo cuando la memoria del sistema fue agotada, es decir, cuando la cantidad de memoria virtual utilizada por el proceso ha duplicado la cantidad de memoria física. Consideramos que luego de ese punto, debido al *thrashing*[1], la verificación se torna inviable.

Cabe destacar que los experimentos se llevaron a cabo teniendo en cuenta la cantidad de instancias necesarias de procesadores y memorias en relación a la cantidad de instancias del producto MotherBoard, es decir, por lo menos un procesador y memoria por motherboard. Si ensayamos el experimento donde las cantidades mencionadas no son respetadas, algunas propiedades, como por ejemplo la 2 o 4, no pueden verificarse.

En los experimentos puede observarse cuán costosa, en relación a los recursos, es la verificación de modelos para sistemas de tiempo real. En efecto, a pesar de que el ejemplo propuesto es muy simple, la cantidad de instancias con las que logramos verificar el modelo es pequeña. El lector podría argumentar que la ineficiencia es producto de la traducción propuesta. Sin embargo, comprobar una caracterización Uppaal *ad hoc* del ejemplo arrojó resultados de análisis similares.

Estos resultados muestran que los mecanismos de abstracción como los productos estructurados que se proponen PPML, son una herramienta clave para lograr mejorar escalabilidad del análisis.

[1]Se denomina *thrashing* o hiperpaginación a la situación en la que la cantidad de recursos utilizados crece de forma inversamente proporcional a la cantidad de procedimiento realizado. Particularmente, se da cuando se cargan y descargan sucesiva y constantemente partes de la imagen de un proceso entre la memoria física o principal y la memoria virtual o espacio de intercambio.

6.4 Formalización y Análisis de Procesos de Negocios Mediante π-Cálculo

Un aspecto interesante a considerar sobre los productos es su *movilidad* dentro del proceso al cual están asociados. Por "movilidad" de los productos nos referimos a la capacidad y frecuencia con la cual se producen, modifican y quitan instancias de los mismos, como parte del comportamiento o ejecución del proceso de negocio.

Con el objetivo de poder formalizar y realizar análisis sobre este aspecto, presentamos una variante del lenguaje PPML, que denominamos π-PML. Como principal contribución, éste cuenta con una semántica enteramente basada en π-cálculo. Esta semántica nos brinda la posibilidad de expresar de manera transparente la movilidad de productos. Además, existen actualmente diferentes herramienta de software, como por ejemplo la mencionada en [103], que proveen un ambiente para el análisis algorítmico de ciertas propiedades.

6.4.1 π-Cálculo

π-cálculo es un álgebra de procesos que puede ser utilizada para describir movilidad en la especificación de sistemas concurrentes [71, 87]. Los objetos que conforman las especificaciones en π-cálculo son *agentes* y *nombres*, donde los *nombres* son un término común que se utiliza para modelar datos (información), referencias identificadores, acciones, etc. Los *agentes* utilizan los nombres como elementos de interacción y su comportamiento es modelado por medio de acciones. Las acciones posibles se expresan por medio de los siguientes *prefijos*: $\bar{x}\langle y\rangle$ envío o escritura (envía y a través de x), $x(y)$ recepción o lectura, τ es la acción oculta o inobservable y $[x = y]\pi$ es la acción condicional en el sentido de que la ejecución de π depende de la igualdad entre x e y.

Los *agentes* se definen en términos de:

$$P ::= \ M \ | \ P_1|P_2 \ | \ \textbf{new}\ zP \ | \ A(x_1, \ldots, x_n)$$

donde $P_1|P_2$ especifica las ejecución paralela de P_1 y P_2, $\textbf{new}zP$ define el alcance de nombre z a P y $A(x_1, \ldots, x_n)$ especifica la recursión parametrizada sobre un conjunto de identificadores. Finalmente M puede ser:

$$M ::= \ \mathbf{0} \ | \ \pi.P \ | \ M_1 + M_2$$

donde $\mathbf{0}$ es el proceso *nulo*, $\pi.P$ especifica uno alguna de las acciones con su correspondiente prefijo y M_1+M_2 define la elección exclusiva entre M_1 y M_2. Además existe una extensión de π-cálculo denominada *polyadic* la cual permite más de un nombre como parámetro en las acciones de envío o recepción.

6.4.2 Descripción general de la semántica de π-PML

La semántica de π-PML preserva el interés de PPML en relación a la presencia de productos como referentes explícitos en la especificación de procesos. En ese sentido, basados en la caracterización de los productos por medio de sus atributos, inicialmente caracterizamos el comportamiento de un atributo, más precisamente una instancia del mismo, mediante la siguiente expresión:

$$
\begin{aligned}
ATT() &= (\textbf{new}\ ref_{att})\overline{new_{att}}\langle ref_{att}\rangle.(ATT_{Instance}(ref_{att}, nil)|ATT()) \\
ATT_{Instance}(ref_{att}, value) &= \overline{ref_{att}}\langle value\rangle.ATT_{Instance}(ref_{att}, value)+ \\
&\qquad ref_{att}(new_{value}).ATT_{Instance}(ref_{att}, new_{value})
\end{aligned}
$$

Básicamente la especificación de un atributo consta de dos expresiones, la primera que simula lo que en programación orientada a objetos se denomina constructor, la cual genera una instancia de atributo con su correspondiente identificador. La segunda modela el comportamiento del atributo, que de manera intuitiva, reproduce el comportamiento de una variable, es decir, permite la lectura ($value$) y modificación (new_{value}) de su valor, utilizando como canal el identificador del atributo.

Cabe destacar que inicialmente y con el fin de simplificar las especificaciones sólo tenemos en cuenta tipos lógicos y enumerados para caracterizar los correspondiente valores posibles. Además reservamos el nombre nil como valor para representar valores de atributos indefinidos.

Contando con la especificación de atributos, la instancia de un producto puede caracterizarse, mediante las siguientes expresiones:

$$
\begin{aligned}
PROD() = (\textbf{new}\ ref_{prod})&\overline{new_prod}\langle ref_{prod}\rangle.(\\
&PROD_I(ref_{prod}, nil_1, \ldots, nil_n, ref_att_1, \ldots, ref_{att_n}) \\
&|PROD() \\
&)
\end{aligned}
$$

$$
\begin{aligned}
&PROD_I(ref_{prod}, val_1, \ldots, val_n, ref_{att_1}, \ldots, ref_{att_n}) = \\
&\quad \overline{ref_{prod}}\langle ref_{att_1}, \ldots, ref_{att_n}\rangle.PROD_I(ref_{prod}, val_1, \ldots, val_n, ref_{att_1}, \ldots, ref_{att_n})+ \\
&\quad \textstyle\sum_{i=1}^{n} ref_{att_i}(n_val_i).PROD_I(ref_{prod}, val_1, \ldots, val_n[^{val_i}/_{nval_i}], ref_{att_1}, \ldots, ref_{att_n})+ \\
&\quad \textstyle\sum_{i=1}^{n} \overline{ref_{att_i}}\langle val_i\rangle.PROD_I(ref_{prod}, val_1, \ldots, val_n, ref_{att_1}, \ldots, ref_{att_n})+
\end{aligned}
$$

De manera similar que para los atributos, la primera expresión representa el constructor de una instancia de producto, donde new_prod se considera un nombre reservado que define esta clase de producto. Cuando un proceso se comunica con este canal, se genera una nueva instancia del producto teniendo como a nil como el valor por defecto para cada uno de sus atributos. En caso de necesitar generadores

con otros valores por defecto, simplemente se establecen los mismo como parte del constructor. Además esta instancia contiene un nombre para identificar la instancia y una referencia por cada uno de sus atributos.

La segunda expresión captura el comportamiento que ofrece un producto. Inicialmente brinda, mediante su identificador como canal de comunicación, los nombres (API[2]) para acceder a sus atributos. Luego, las correspondientes opciones de lectura o escritura de cada uno de ellos. Cabe destacar que utilizamos de manera abreviada el operador de elección mediante la siguiente espresión $\sum_{i=1}^{m}(M_i) = M_1+, \ldots, +M_m$.

Los productos los podemos ver como un *producto compuesto dummy* que consta únicamente de referencias a sus componentes. En caso de *productos estructurados*, éstos se caracterizan asociando, para cada producto componente, un atributo que contiene como valor una referencia (nombre) a la instancia del producto que lo compone.

El comportamiento de las compuertas es caracterizado por medio de agentes que sincronizan o distribuyen referencias a productos. Por ejemplo, una compuerta multiplexora que recibe dos producto y retorna un producto compuesto, se especifica mediante la siguiente expresión:

$$GATE() = Gate_1_in_1(p1).Gate_1_in_2(p2).new_comp2(o).\overline{o}\langle p1, p2\rangle.\overline{Gate_1_out}\langle o\rangle.GATE()$$

La semántica para los *procesos atómicos* consiste en la siguiente expresión:

$$P_{name}(in, out) = in\langle i_p\rangle.REQ(i_p).DO(i_p, o_p).\overline{out}\langle o_p\rangle.P_{name}(in, out)$$

Esta expresión caracteriza de manera abstracta un producto atómico P_{name} que utiliza a *in* y *out* como canales utilizados para recibir y retornar el producto que procesa. La especificación costa de dos agentes *REQ* y *DO* que detallan el comportamiento particular del proceso. *REQ* se encarga de constatar de que el producto de entrada i_p cumple con los requisitos necesarios para la ejecución del mismo. *DO* especifica el comportamiento del proceso, detallando cada una de las modificaciones que éste realiza sobre el producto de entrada. Finalmente, escribiendo sobre el canal de salida, el proceso retorna el identificador del producto procesado y queda en condiciones de volver a ejecutarse.

La semántica de composición de procesos (secuencial, condicional y paralela) se obtiene mediante los operadores de composición de procesos que brinda π-cálculo.

[2] *Application Programming Interface* es el conjunto de funciones y procedimientos que ofrece cierta biblioteca para ser utilizado por otro software como una capa de abstracción.

6.4.3 Análisis de modelos

La semántica formal basada en π-cálculo que describimos ofrece nuevas oportunidades para el análisis automatizado de las especificaciones. Por ejemplo, verificación de propiedades mediante *model checkers*, análisis de simplificación/minimización de procesos a través de los diferentes conceptos de bisimulación. A diferencia del formalización propuesta en [77, 78], una de las más relevantes y cercanas, nuestro objetivo se enfoca en tomar ventaja de la información complementaria que ofrece la descripción de los productos como parte del modelo, explotando su aspecto de movilidad.

6.5 Sumario

En este capítulo presentamos una técnica para analizar procesos de negocios especificados mediante el lenguaje PPML. A diferencia de numerosos trabajos que proponen este tipo de análisis, en el presente trabajo tomamos ventaja de dos características particulares presentes en las especificaciones PPML. La primera es la presencia de *productos* como elementos del lenguaje que capturan la descripción de los objetos del dominio del sistema que éste manipula. Las segunda característica relevante de PPML es la posibilidad de especificar restricciones temporales del modelo. Estas restricciones son asociadas con límites temporales, inferior y superior, a los procesos del sistema.

La técnica presentada propone una traducción de lenguaje para la especificación de procesos de negocios PPML a un formalismo que pueda capturar su semántica y que brinde soporte para la verificación de propiedades. Particularmente escogimos Uppaal, cuyas descripciones se basan en autómatas temporizados, permitiendo caracterizar los elementos de los modelos PPML. Además, como lenguaje para la especificación de propiedades, Uppaal provee la Computational Tree Logic, que permite expresar una amplia gama de propiedades incluyendo propiedades de *safety* y *liveness*. También, gracias a su soporte directo, como parte de las propiedades podemos incorporar análisis de restricciones temporales, como por ejemplo, tiempos de respuesta, entre otras.

Para ensayar la traducción propuesta, se analizaron propiedades sobre un modelo de procesos de negocios muy simple que resaltan las características relevantes de nuestro trabajo.

Finalmente proponemos una variante del lenguaje PPML que denominamos π-PML, en sintonía con el formalismo subyacente que define su semántica. Esta variante tiene como principal objetivo el análisis de la movilidad de los elementos que integran un proceso de negocio, en particular, en el análisis de movilidad que

establece el ciclo que vida de los productos dentro del mismo.

Capítulo 7

Conclusiones y Trabajos Futuros

El principal objetivo alcanzado en el desarrollo de este trabajo fue el de poder capturar formalmente la noción de proceso de negocios, particularmente el concepto de *workflow*. A partir de estas especificaciones, propusimos diferentes métodos para realizar análisis de los modelos, mediante la verificación formal de propiedades temporales.

Nuestro objetivo es el de contribuir a la necesidad manifiesta, no sólo a nivel académico sino también del ámbito industrial, de poder analizar complejas descripciones, en general informales, de procesos de negocios. Esta necesidad ha tomado relevancia en los últimos años con el surgimiento de diferentes métodos y técnicas de análisis, que en su gran mayoría hemos mencionado en nuestro trabajo.

A modo general, un desafío que subyace a todo nuestro trabajo fue el de conectar dos áreas de las Ciencias de la Computación, que generalmente se encuentran alejadas. Por un lado la informalidad de las descripciones de procesos de negocios, particularmente descripciones de *workflows*, y por el otro la rigurosidad que imponen las técnicas de análisis formales.

Con el objetivo de establecer un marco teórico de referencia, en la Sección 3.4 presentamos un análisis que describe nuestra perspectiva sobre los conceptos comunes que se encuentran presentes a la hora de describir un proceso de negocio. Además, como ejemplo para evidenciar nuestro análisis, transitamos estos conceptos a través de diferentes lenguajes (cada uno con características particulares) utilizados para este tipo de descripciones. A partir de allí, enfocamos nuestro interés en la posibilidad de expresar (especificar) y analizar de manera automática propiedades que capturen situaciones sobre el comportamiento de procesos de negocios, particularmente de los denominados *workflows*.

Los aportes realizados contribuyen en la realización de los objetivos propuestos, mencionados en la Sección 1.2. Abordamos como hipótesis la idea de utilizar lógi-

cas temporales como lenguaje para describir propiedades de procesos de negocios. Realizamos un estudio de las características que estas lógicas nos proveen y el análisis de diferentes métodos para su aplicación a nuestros objetivos. Particularmente consideramos relevante la idea de *fluente* como elemento del lenguaje de especificación de propiedades y avanzamos en ese sentido, en su directa aplicación a nuestro propósito, favoreciendo la caracterización de situaciones complejas de procesos de negocios de manera simple. Este enfoque fue publicado en [84].

Justificados por nuestro interés en el análisis, investigamos sobre la necesidad de representar los elementos, a los que denominamos *productos*, que forman parte del dominio de los procesos. Creemos que contar con un objeto dentro del modelo que represente la descripción de los productos que los proceso manipulan, nos brinda una perspectiva interesante para el análisis de procesos de negocios. Para ello retomamos el lenguaje PPML, un lenguaje para la descripción de procesos de negocios donde los productos se especifican como parte de los modelos. Basados en estas descripciones, propusimos el análisis de dichos modelos enfocando nuestro interés en explotar las ventajas de estas descripciones más ricas de procesos [83]. Además trabajamos en la definición de una variante del lenguaje PPML que denominamos π-PML, cuyo principal objetivo es el análisis de la movilidad de los elementos que integran un proceso de negocio, en particular, en el análisis de movilidad que establece el ciclo que vida de los productos dentro del mismo.

Encontramos en la técnica de *model checking* un recurso potente que nos permite utilizar las lógicas temporales como lenguaje de especificación de situaciones de procesos de negocios a verificar. Se estudiaron diferentes herramientas de software que implementan esta técnica con el propósito de ensayar con las mismas los análisis propuestos. Particularmente empleamos LTSA y Uppaal como soporte para la verificación automática de procesos de negocios. Para tal fin, se expusieron codificaciones (una totalmente automatizada) a los correspondientes formalismos que forman parte del dominio de dichas herramientas. Tomando ventaja sobre estas caracterizaciones abstractas, investigamos y propusimos un proceso para generación de modelos de *workflows* a partir de un conjunto de tareas que colaboran en la realización de un objetivo y una serie de restricciones entre las mismas.

Finalmente, en pos de favorecer su potencial utilización, desarrollamos un prototipo de ambiente que brinda asistencia para la descripción y análisis de propiedades de *workflows* [85], utilizando como lenguaje la lógica FLTL. Partiendo de una especificación de un *workflow*, esta herramienta permite de manera gráfica caracterizar estados abstractos (fluentes) de manera sencilla a partir de eventos del modelo. A partir de allí, con estos estados o eventos del modelos, pueden describirse situaciones (como fórmulas) que luego se verifican. Las herramienta asiste al usuario en la edición de las fórmulas y provee una base de datos con patrones de propiedades como

exclusión mutua, etc., que pueden ser instanciadas con elementos del modelo bajo análisis.

El interés por la flexibilidad a la hora de describir propiedades, nos condujo a investigar y proponer una extensión a la lógica FLTL, que llamamos Counting Fluent Linear-time Temporal Logic, que permite incorporar fluentes contadores como complemento a los fluentes proposicionales. Creemos que esta extensión es de gran utilidad a la hora de expresar ciertas propiedades, además, en pos de un resultado aplicable al contexto de este trabajo, desarrollamos un método que permite realizar *model checking*, bajo ciertos límites, de propiedades de sistemas expresados con CFLTL.

7.1 Trabajos Futuros

Entre las actividades futuras a continuar en el contexto de este trabajo, podemos mencionar las siguientes:

- Un tema que creemos de gran interés y consideramos profundizar es en la generación de workflows a partir de especificaciones de escenarios y restricciones o propiedades sobre ellos. El procedimiento hallado hasta el momento tiene problemas de escalabilidad y estamos investigando sobre diferentes técnicas, como por ejemplo, técnicas de abstracción para ser aplicadas al proceso de construcción del *workflow*

- Otro trabajo a profundizar, es el estudio del poder expresivo de la Lógica CFLTL propuesta, como así también otros aspectos teóricos de la misma como su complejidad y decibilidad.

- Con respecto al análisis de descripciones de procesos de negocios con referencias explícitas de productos, estamos interesados en explotar la semántica que nos proporciona π-Cálculo.

Bibliografía

[1] Luca Aceto, Anna Ingólfsdóttir, Kim Guldstrand Larsen, and Jiri Srba, *Reactive systems: modelling, specification and verification*, Cambridge University Press, 2007. 2.2

[2] Bowen Alpern and Fred B. Schneider, *Defining liveness*, Inf. Process. Lett. **21** (1985), no. 4, 181–185. 2.3.2

[3] Rajeev Alur, Costas Courcoubetis, and David L. Dill, *Model-checking in dense real-time*, Inf. Comput. **104** (1993), no. 1, 2–34. 2.3.2

[4] Rajeev Alur and David L. Dill, *Automata for modeling real-time systems*, ICALP, 1990, pp. 322–335. 2.2.3

[5] ———, *A theory of timed automata*, Theor. Comput. Sci. **126** (1994), no. 2, 183–235. 2.2.3

[6] Toshiro Araki and Tadao Kasami, *Some decision problems related to the reachability problem for petri nets*, Theoretical Computer Science **3** (1977), no. 1, 85 – 104. 2.2.2

[7] Scott Fordin et al. Assaf Arkin, Sid Askary, *Web service choreography interface (wsci) 1.0*. 5.1

[8] Ahmed Awad and Frank Puhlmann, *Structural detection of deadlocks in business process models*, BIS, 2008, pp. 239–250. 3.4

[9] J.C.M. Baeten, *Applications of process algebra*, Cambridge Tracts in Theoretical Computer Science, Cambridge University Press, 1990. 1.1

[10] Christel Baier and Joost-Pieter Katoen, *Principles of model checking*, The MIT Press, May 2008. 1.3, 2.1

[11] Gerd Behrmann, Alexandre David, and Kim Guldstrand Larsen, *A tutorial on uppaal*, SFM, 2004, pp. 200–236. 2.4.1, 2.4.1

[12] Mordechai Ben-Ari, *Principles of the spin model checker*, Springer, 2008. 1.1

[13] Johan Bengtsson, Kim Guldstrand Larsen, Fredrik Larsson, Paul Pettersson, and Wang Yi, *Uppaal - a tool suite for automatic verification of real-time systems*, Hybrid Systems, 1995, pp. 232–243. 2.4.1, 6.1

[14] J.A. Bergstra, A. Ponse, and S.A. Smolka, *Handbook of process algebra*, Elsevier Science, 2001. 1.1

[15] Patrick Blackburn, Johan F. A. K. van Benthem, and Frank Wolter, *Handbook of modal logic, volume 3 (studies in logic and practical reasoning)*, Elsevier Science Inc., New York, NY, USA, 2006. 1.3, 2.1

[16] Tommaso Bolognesi and Ed Brinksma, *Introduction to the iso specification language lotos*, Computer Networks **14** (1987), 25–59. 1.1

[17] Grady Booch, James Rumbaugh, and Ivar Jacobson, *Unified modeling language user guide, the (2nd edition)*, Addison-Wesley Professional, 2005. 3.3.1, A.4

[18] Egon Börger, *Approaches to modeling business processes: a critical analysis of bpmn, workflow patterns and yawl*, Software and System Modeling **11** (2012), no. 3, 305–318. 3.3.1

[19] Marius Bozga, Conrado Daws, Oded Maler, Alfredo Olivero, Stavros Tripakis, and Sergio Yovine, *Kronos: A model-checking tool for real-time systems*, CAV, 1998, pp. 546–550. 6.1

[20] Roberto Bruni, Hernán C. Melgratti, and Ugo Montanari, *Theoretical foundations for compensations in flow composition languages*, POPL, 2005, pp. 209–220. 1.1, 5.4

[21] S.E.R. Carvalho, J.L. Fiadeiro, and E.H. Haeusler, *A formal approach to real-time object oriented software*, Workshop on Real-Time Programming, 1997, pp. 91–96. 5.2.2

[22] Edmund M. Clarke and E. Allen Emerson, *Design and synthesis of synchronization skeletons using branching-time temporal logic*, Logic of Programs, 1981, pp. 52–71. 4.7

[23] Edmund M. Clarke, E. Allen Emerson, and A. Prasad Sistla, *Automatic verification of finite-state concurrent systems using temporal logic specifications*, ACM Trans. Program. Lang. Syst. **8** (1986), no. 2, 244–263. 6.1, 6.3

[24] Edmund M. Clarke, Orna Grumberg, and Doron Peled, *Model checking*, MIT Press, 2001. 2.4.1, 4.1

[25] P. DasGupta, *A roadmap for formal property verification*, Springer London, Limited, 2007. 1

[26] Frank DeRemer and Hans Kron, *Programming-in-the large versus programming-in-the-small*, SIGPLAN Not. **10** (1975), no. 6, 114–121. 1.1

[27] Edsger W. Dijkstra and W. H. J. Feijen, *A method of programming*, Addison-Wesley, 1988. 2.2

[28] E. Allen Emerson and Edmund M. Clarke, *Using branching time temporal logic to synthesize synchronization skeletons*, Sci. Comput. Program. **2** (1982), no. 3, 241–266. 2.3.2

[29] E. Allen Emerson and Joseph Y. Halpern, *"sometimes" and "not never" revisited: on branching versus linear time temporal logic*, J. ACM **33** (1986), no. 1, 151–178. 2.3.2

[30] H. Enderton and H.B. Enderton, *A mathematical introduction to logic*, Elsevier Science, 2001. 5.2.1

[31] Alexandre Alves et. al., *Web services business process execution language version 2.0*, Tech. report, OASIS Web Services Business Process Execution Language (WSBPEL) TC, 2007. 3.3

[32] Dirk Fahland, Jan Mendling, Hajo A. Reijers, Barbara Weber, Matthias Weidlich, and Stefan Zugal, *Declarative versus imperative process modeling languages: The issue of maintainability*, Business Process Management Workshops, 2009, pp. 477–488. 3.4, 4.7

[33] Norman E. Fenton and Shari Lawrence Pfleeger, *Software metrics - a practical and rigorous approach (2. ed.)*, International Thomson, 1996. 5.2.1

[34] José Luiz Fiadeiro and T. S. E. Maibaum, *Temporal theories as modularisation units for concurrent system specification*, Formal Asp. Comput. **4** (1992), no. 3, 239–272. 5.2.2

[35] Howard Foster, Wolfgang Emmerich, Jeff Kramer, Jeff Magee, David S. Rosenblum, and Sebastián Uchitel, *Model checking service compositions under resource constraints*, ESEC/SIGSOFT FSE, 2007, pp. 225–234. 1.1

[36] Howard Foster, Sebastián Uchitel, Jeff Magee, and Jeff Kramer, *Ltsa-ws: a tool for model-based verification of web service compositions and choreography*, ICSE'06, 2006, pp. 771–774. 1.1

[37] Dimitra Giannakopoulou and Jeff Magee, *Fluent model checking for event-based systems*, ESEC SIGSOFT FSE, 2003, pp. 257–266. 2.3.2, 4.1, 4.4, 4.8.1

[38] Claude Girault and Rüdiger Valk, *Petri nets for systems engineering - a guide to modeling, verification, and applications*, Springer, 2003. 1.1, 2.2.2, 3.3.1, A.2

[39] Object Management Group, *Business process modeling notation, v1.1.* 3.3.1, A.1

[40] Nicolas Guelfi and Amel Mammar, *A formal semantics of timed activity diagrams and its promela translation*, APSEC, 2005, pp. 283–290. 1.1

[41] D. Harel and A. Pnueli, *On the development of reactive systems*, Logics and models of concurrent systems (Krzysztof R. Apt, ed.), Springer-Verlag New York, Inc., New York, NY, USA, 1985, pp. 477–498. 2.2

[42] Thomas A. Henzinger, Zohar Manna, and Amir Pnueli, *Timed transition systems*, Real-Time: Theory in Practice, REX Workshop, Mook, The Netherlands, June 3-7, 1991, 1991, pp. 226–251. 5.1

[43] Holger Hinrichs and Thomas Aden, *An iso 9001:2000 compliant quality management system for data integration in data warehouse systems*, In Proc. DMDW'01, 2001, p. 2001. 3.4, 5.1

[44] C. A. R. Hoare, *An axiomatic basis for computer programming*, Commun. ACM **12** (1969), no. 10, 576–580. 2.2, 5.2.2

[45] ———, *Communicating sequential processes*, Prentice-Hall, 1985. 1.1, 2.2, 2.2.1

[46] David Hollingsworth, *The workflow reference model*, Technical report, Workflow Management Coalition, 1995. 1, 3.2

[47] Gerard J. Holzmann, *The spin model checker - primer and reference manual*, Addison-Wesley, 2004. 1.1

[48] J.E. Hopcroft, R. Motwani, and J.D. Ullman, *Introduction to automata theory, languages, and computation*, Pearson/Addison Wesley, 2007. 1.1

[49] Michael Huth and Mark Dermot Ryan, *Logic in computer science - modelling and reasoning about systems (2. ed.)*, Cambridge University Press, 2004. 2.3.2

[50] K. Jensen and L.M. Kristensen, *Coloured petri nets: Modelling and validation of concurrent systems*, Springer, 2009. 2.2.2

[51] Christos T. Karamanolis, Dimitra Giannakopoulou, Jeff Magee, and Stuart M. Wheater, *Model checking of workflow schemas*, EDOC, 2000, pp. 170–181. 1.1, 4.4, 4.9

[52] G. Keller, M. Nüttgens, and A. W. Scheer, *Semantische Prozeßmodellierung auf der Grundlage Ereignisgesteuerter Prozeßketten (EPK)*, Tech. Report 89, Universität des Saarlandes, Germany, Saarbrücken, Germany, 1992. 3.3.1, A.3

[53] Bartosz Kiepuszewski, *Expressiveness and suitability of languages for control flow modelling in workflows*, Ph.D. thesis, Queensland University of Technology, 2003. A.2.2

[54] Ekkart Kindler, *On the semantics of epcs: Resolving the vicious circle*, Data Knowl. Eng. **56** (2006), no. 1, 23–40. 3.3.1, 4.3.2

[55] Donald E. Knuth, *backus normal form vs. backus naur form*, Commun. ACM **7** (1964), no. 12, 735–736. 2.3.1

[56] Jana Koehler, Giuliano Tirenni, and Santhosh Kumaran, *From business process model to consistent implementation: A case for formal verification methods*, In EDOC, IEEE Computer Society, 2002, pp. 96–106. 5.1

[57] Oliver Kopp, Daniel Martin, Daniel Wutke, and Frank Leymann, *The difference between graph-based and block-structured business process modelling languages*, Enterprise Modelling and Information Systems Architectures **4** (2009), no. 1, 3–13. 3.3.1

[58] Ron Koymans, *Specifying real-time properties with metric temporal logic*, Real-Time Systems **2** (1990), no. 4, 255–299. 2.3.2

[59] Leslie Lamport, *The temporal logic of actions*, ACM Trans. Program. Lang. Syst. **16** (1994), no. 3, 872–923. 1.1

[60] Nazia Leyla, Ahmed Shah Mashiyat, Hao Wang, and Wendy MacCaull, *Towards workflow verification*, Proceedings of the 2010 Conference of the Center for Advanced Studies on Collaborative Research (Riverton, NJ, USA), CASCON '10, IBM Corp., 2010, pp. 253–267. 4.9

[61] Frank Leymann and Dieter Roller, *Production workflow - concepts and techniques*, Prentice Hall, 2000. 3.3

[62] Jeff Magee and Jeff Kramer, *Concurrency - state models and java programs (2. ed.)*, Wiley, 2006. 2.2, 2.2.1, 2.2.1, 2.4.1, 4.3, 4.4

[63] T. S. E. Maibaum, *An overview of the mensurae language: Specifying business processes*, Rigorous Object-Oriented Methods, ROOM 2000, York, UK, 17, January 2000, 2000. 5.1, 5.2

[64] Zohar Manna and Amir Pnueli, *The temporal logic of reactive and concurrent systems - specification*, Springer-Verlag New York, Inc., 1992. 2.3.2, 6.1, 6.3

[65] Zohar Manna and Amir Pnueli, *Temporal verification of reactive systems: safety*, Springer-Verlag New York, Inc., New York, NY, USA, 1995. 2.2, 2.3.2, 6.1, 6.3

[66] Zohar Manna and Pierre Wolper, *Synthesis of communicating processes from temporal logic specifications*, ACM Trans. Program. Lang. Syst. **6** (1984), no. 1, 68–93. 4.7

[67] Cristian Masalagiu, Wei-Ngan Chin, Stefan Andrei, and Vasile Alaiba, *A rigorous methodology for specification and verification of business processes*, Formal Asp. Comput. **21** (2009), no. 5, 495–510. 1.1

[68] Bertrand Meyer, *Object-oriented software construction, 2nd edition*, Prentice-Hall, 1997. 5.2.1, 5.3

[69] Rob Miller and Murray Shanahan, *The event calculus in classical logic - alternative axiomatisations*, Electron. Trans. Artif. Intell. **3** (1999), no. A, 77–105. 2.3.2

[70] Robin Milner, *A calculus of communicating systems*, Lecture Notes in Computer Science, vol. 92, Springer, 1980. 2.2, 2.2.1

[71] ———, *Communicating and mobile systems: the pi-calculus*, Cambridge University Press, 1999. 2.2, 3.4, 6.4.1

[72] Marco Montali, Maja Pesic, Wil M. P. van der Aalst, Federico Chesani, Paola Mello, and Sergio Storari, *Declarative specification and verification of service choreographiess*, TWEB **4** (2010), no. 1, 3:1–3:62. 3.4, 4.1, 4.4

[73] Shoichi Morimoto, *A survey of formal verification for business process modeling*, Computational Science - ICCS 2008, 8th International Conference, Kraków, Poland, June 23-25, 2008, Proceedings, Part II (Marian Bubak, G. Dick van Albada, Jack Dongarra, and Peter M. A. Sloot, eds.), Lecture Notes in Computer Science, vol. 5102, Springer, 2008. 1.1, 3.4, 4.4

[74] M. Myers and A. Kaposi, *A first systems book: Technology and management*, Imperial College Press, 2004. 5.2

[75] Maja Pesic, M. H. Schonenberg, Natalia Sidorova, and Wil M. P. van der Aalst, *Constraint-based workflow models: Change made easy*, OTM Conferences (1), 2007, pp. 77–94. 1.1, 4.4, 4.9

[76] Nir Piterman, Amir Pnueli, and Yaniv Sa'ar, *Synthesis of reactive(1) designs*, VMCAI, 2006, pp. 364–380. 4.7

[77] Frank Puhlmann, *Soundness verification of business processes specified in the pi-calculus*, OTM Conferences (1), 2007, pp. 6–23. 1.1, 3.4, 6.4.3

[78] Frank Puhlmann and Mathias Weske, *Using the pi-calculus for formalizing workflow patterns*, Business Process Management, 2005, pp. 153–168. 3.4, 6.4.3

[79] ———, *A look around the corner: The pi-calculus*, T. Petri Nets and Other Models of Concurrency **2** (2009), 64–78. 3.4

[80] Yi Qian, Yuming Xu, Zheng Wang, Geguang Pu, Huibiao Zhu, and Chao Cai, *Tool support for bpel verification in activebpel engine*, ASWEC, 2007, pp. 90–100. 1.1

[81] Zongyan Qiu, Shuling Wang, Geguang Pu, and Xiangpeng Zhao, *Semantics of bpel4ws-like fault and compensation handling*, FM, 2005, pp. 350–365. 1.1

[82] Fazle Rabbi, Hao Wang, and Wendy MacCaull, *Yawl2dve: An automated translator for workflow verification*, SSIRI, 2010, pp. 53–59. 4.9

[83] Germán Regis, Nazareno Aguirre, and T. S. E. Maibaum, *Specifying and verifying business processes using ppml*, Formal Methods and Software Engineering, 11th International Conference on Formal Engineering Methods, ICFEM 2009, Rio de Janeiro, Brazil, December 9-12, 2009. Proceedings (Karin Breitman and Ana Cavalcanti, eds.), vol. 5885, Lecture Notes in Computer Science, no. 1, Springer, 2009, pp. 737–756. 7

[84] Germán Regis, Nicolás Ricci, Nazareno Aguirre, and T. S. E. Maibaum, *Specifying and verifying declarative fluent temporal logic properties of workflows*, Brazilian Symposium on Formal Methods, SBMF 2012, 2012, pp. 147–162. 7

[85] Germán Regis, Fernando Villar, and Nicolás Ricci, *Fluent logic workflow analyser: A tool for the verification of workflow properties*, LAFM, 2014, pp. 46–51. 7

[86] A. W. Roscoe, C. A. R. Hoare, and Richard Bird, *The theory and practice of concurrency*, Prentice Hall PTR, Upper Saddle River, NJ, USA, 1997. 1.1

[87] Davide Sangiorgi and David Walker, *The pi-calculus - a theory of mobile processes*, Cambridge University Press, 2001. 3.4, 6.4.1

[88] Donald Sannella and Andrzej Tarlecki, *Foundations of algebraic specification and formal software development*, EATCS Monographs on theoretical computer science, Springer, 2012. 5.2.1

[89] Dennis M. M. Schunselaar, Fabrizio Maria Maggi, Natalia Sidorova, and Wil M. P. van der Aalst, *Configurable declare: Designing customisable flexible process models*, OTM Conferences (1), 2012, pp. 20–37. 1.1, 3.4, 4.4

[90] Krister Segerberg, *The delta operator at three levels of analysis*, Logic, Action, and Information, 1996, pp. 63–78. 5.2.2

[91] Arthur H. M. ter Hofstede, Wil M. P. van der Aalst, Michael Adams, and Nick Russell (eds.), *Modern business process automation - yawl and its support environment*, Springer, 2010. 1, 1.1, 3.2, 3.3.1, 4.2, 4.3, 4.3.2, 4.7

[92] The Institute of Electrical and Eletronics Engineers, *Ieee standard glossary of software engineering terminology*, IEEE Standard, September 1990. 1

[93] Sebastián Uchitel, Greg Brunet, and Marsha Chechik, *Synthesis of partial behavior models from properties and scenarios*, IEEE Trans. Software Eng. **35** (2009), no. 3, 384–406. 4.7

[94] Franck van Breugel and Maria Koshkina, *Models and verification of bpel*, Tech. report, York University, 2006. 1.1

[95] Wil M. P. van der Aalst, *Process mining - discovery, conformance and enhancement of business processes*, Springer, 2011. 1

[96] Wil M. P. van der Aalst and Arthur H. M. ter Hofstede, *Yawl: yet another workflow language*, Inf. Syst. **30** (2005), no. 4, 245–275. 3.3.1, 4.1, 4.2, A.2

[97] Wil M. P. van der Aalst, Kees M. van Hee, Arthur H. M. ter Hofstede, Natalia Sidorova, H. M. W. Verbeek, Marc Voorhoeve, and Moe Thandar Wynn, *Soundness of workflow nets: classification, decidability, and analysis*, Formal Asp. Comput. **23** (2011), no. 3, 333–363. 3.3.1, 3.4, 4.4, 4.9

[98] W.M.P. van der Aalst, A.H.M. ter Hofstede, B. Kiepuszewski, and A.P. Barros, *Workflow patterns*, Distributed and Parallel Databases **14** (2003), no. 1, 5–51. 1.1, 3.3.1, 3.3.1, 3.3.1, 3.4, 4.1, 4.2, 5.1, A.2

[99] Jan Martijn E. M. van der Werf, H. M. W. Verbeek, and Wil M. P. van der Aalst, *Context-aware compliance checking*, BPM, 2012, pp. 98–113. 1.1

[100] Moshe Y. Vardi and Pierre Wolper, *Automata-theoretic techniques for modal logics of programs*, J. Comput. Syst. Sci. **32** (1986), no. 2, 183–221. 4.8.1

[101] H. M. W. Verbeek, Moe Thandar Wynn, Wil M. P. van der Aalst, and Arthur H. M. ter Hofstede, *Reduction rules for reset/inhibitor nets*, J. Comput. Syst. Sci. **76** (2010), no. 2, 125–143. 1.1

[102] H. M. W. (Eric) Verbeek, Twan Basten, and Wil M. P. van der Aalst, *Diagnosing workflow processes using woflan*, Comput. J. **44** (2001), no. 4, 246–279. 1.1

[103] Björn Victor and Faron Moller, *The mobility workbench - a tool for the pi-calculus*, CAV, 1994, pp. 428–440. 6.4

[104] J. Wang, *Timed petri nets: Theory and application*, The Kluwer international series on discrete event dynamic systems, Kluwer Academic Publishers, 1998. 2.2.2

[105] David A. Watt and William Findlay, *Programming language design concepts*, Wiley, 2004. 2.4.1

[106] Mathias Weske, *Business process management: Concepts, languages, architectures*, Springer, 2007. 3.2

[107] Michael Westergaard and Fabrizio Maria Maggi, *Declare: A tool suite for declarative workflow modeling and enactment*, BPM (Demos), 2011. 4.7, 4.7.1

[108] S.A. White and D. Miers, *Bpmn modeling and reference guide*, Future Strategies Incorporated, 2008. 3.3.1, A.1

[109] Roel Wieringa, *Design methods for reactive systems - yourdon, statemate, and the uml*, Morgan Kaufmann, 2003. 2.2, 2.2

[110] Martin Wirsing, *Algebraic specification*, Handbook of Theoretical Computer Science, Volume B: Formal Models and Sematics (B), The MIT Press, 1990, pp. 675–788. 5.2.1

[111] Petia Wohed, Nick Russell, Arthur H. M. Ter Hofstede, and Birger Andersson, *Patterns-based evaluation of open source bpm systems: The cases of jbpm, openwfe, and enhydra shark*. A.2.2

[112] Peter Y. H. Wong and Jeremy Gibbons, *Formalisations and applications of bpmn*, Sci. Comput. Program. **76** (2011), no. 8, 633–650. 1.1

[113] _____, *Property specifications for workflow modelling*, Sci. Comput. Program. **76** (2011), no. 10, 942–967. 1.1, 3.4, 4.1, 4.4

[114] Moe Thandar Wynn, H. M. W. (Eric) Verbeek, Wil M. P. van der Aalst, Arthur H. M. ter Hofstede, and David Edmond, *Reduction rules for yawl workflows with cancellation regions and or-joins*, Information & Software Technology **51** (2009), no. 6, 1010–1020. 3.3.1

Apéndice A

Lenguajes para el Modelado de Procesos de Negocios

A.1 The Business Process Modeling Notation (BPMN)

The Business Process Modeling Notation [108] surgió en 2004 como un estándar para la descripción de procesos de negocios. Uno de sus objetivos fue la unificación de diferentes lenguajes y herramientas existentes al momento. Debido a su aprobación por la comunidad académica y su utilización en la industria, en 2008 el *Object Management Group* (OMG)[1] publicó la versión 1.1 [39] y 2011 su versión BPMN 2.0.

Una especificación en BPMN es un diagrama (BPD) compuesto por elementos: *Objetos de Flujo*, *Canales*, *Artefactos* y *Conectores*. Su representación gráfica se puede observar en la Figura A.1.

A.1.1 Objetos de Flujo

Los Objetos de Flujo (*Flow Objects*) describen diferentes objetos que forman parte del Proceso de Negocio, estos pueden ser:

- Actividades (*Activity*): Especifican tareas que deben ser realizadas. Las actividades pueden ser atómicas o compuestas, estas últimas representan un subdiagrama (sub-proceso) que describe la misma. Las actividades puede tener una única instancia o pueden tener *múltiples instancias* y en este último caso puede ser creadas secuencial o paralelamente.

[1] Es un consorcio formado por diversas compañías y organizaciones sin fines de lucro dedicado al cuidado y el establecimiento de diversos estándares de tecnologías orientadas a objetos

- Eventos (*event*): Representan sucesos que ocurren o pueden ocurrir durante un proceso. Existen tres grupos de eventos y en cada uno ellos existen diversos tipos de eventos (gráficamente diferenciados por iconos alusivos):
  - *Iniciales*: estos eventos indican el comienzo de un proceso, esto es, inician el flujo de control, por lo tanto no tienen conexiones de entrada. Los tipos de eventos que pueden dar comienzo a un proceso son: Simple, Mensaje, Programado, Condicional, Señal o Múltiple. Los eventos *simples* carecen de restricciones para su ocurrencia, generalmente son utilizados como inicio de sub-procesos. Los *mensajes* representan la interacción con otro proceso. La ocurrencia de los eventos *programados* está definida como un cronograma temporizado (e.j. cada 1 hora, el 1 de agosto a las 12 pm, etc). Como lo expresa su nombre, los eventos *condicionales* son especificados mediante una fórmula l'ogica (condición), cuando ésta es verdadera, el evento tiene lugar. Los eventos caracterizados por *Señales* ocurren ante el disparo de la señal que los caracteriza, a diferencia de los mensajes, cuando una señal es difundida (*broadcast*), afecta a *todos* los eventos marcados son ella. Los eventos *múltiples* están definidos por más de una forma de ocurrencia, generalmente son una composición de los anteriores.
  - *Intermedios*: ocurren entre el comienzo y fin del proceso, y pueden utilizarse para recibir o lanzar el evento. Los tipos de eventos intermedios pueden ser: Simple, Mensaje, Programado, Condicional, Señal, Múltiple o Vínculo. Este último puede utilizarse como alternativa conectar el flujo entre dos partes de un proceso.
  - *Terminales*: como dualidad de los eventos iniciales, los eventos terminales indican la finalización de un proceso. Estos eventos no pueden tener conexiones salientes, es decir, el flujo de control termina en ellos. Los tipos de eventos de finalización son: Simple, Mensaje, Señal, Múltiple o Final. El evento *final* indica la finalización total del proceso.
- Compuertas (*gateway*): Se utilizan distribuir o sincronizar el flujo del proceso entre sus actividades. Las compuertas puede ser:
  - *Exclusivas*: como compuerta de distribución, cumplen el rol de una sentencia condicional en un lenguaje de programación, es decir, definen un punto dentro del proceso en el cual *sólo una* de sus alternativas (conexiones salientes) es elegida. Esta elección puede realizarse en base a información del proceso o a eventos del mismo. Aunque no es obligatorio, se puede definir una conexión saliente por defecto. Estas compuertas también pueden ser utilizadas para controlar la correcta sincronización

(filtro) de uno o más flujos entrantes, esto es, dada una compuerta exclusiva con conexiones entrantes, *sólo una* de ellas será propagada, como saliente, en base a una condición o evento dado.

- *Inclusivas*: el funcionamiento de estas compuertas es similar a la anterior, pero sin la restricción de que unicidad de los flujos o conexiones seleccionadas. En las compuertas inclusivas, *al menos* una conexión es elegida, pudiendo seleccionarse más de una.
- *Paralelas*: estas compuertas pueden utilizarse como puntos de sincronización de *todas* las conexiones entrantes, o como un "*fork*" (en lenguaje de programación), en caso de su funcionalidad distributiva. Es decir, cuando un flujo de control alcanza una compuerta paralela, ésta propaga el flujo en *todas* sus conexiones salientes.
- *Complejas*: las compuertas complejas se utilizan como una abreviación de una combinación de compuertas simples interconectadas.

A.1.2 Canales

Los Canales (*Swimlanes*) son utilizados para organizar las actividades del flujo en diferentes categorías, por ejemplo áreas funcionales, roles, participantes, etc.. Los canales pueden ser:

- *Pool*: contenedor de procesos, generalmente caracteriza un participante (entidad del proceso).
- *Lanes*: son sub-divisiones dentro de un *pool* y se utilizan para representar subroles dentro de una entidad del proceso.

A.1.3 Artefactos

Los Artefactos (*Artifacts*) describen información pertinente al proceso que no influye en el flujo de control. Los artefactos que provee BPMN son:

- Anotaciones: Utilizadas para brindar información adicional en cualquier punto del proceso, por ejemplo, para describir una actividad.
- Grupos: Se utilizan para agrupar ciertos elementos del diagrama pero sin alterar el flujo de control, sino para fines de documentación o análisis.
- Datos: representan información que manipula el proceso, generalmente se asocian a los flujos de control entre actividades.

A.1.4 Conectores

Los Conectores se utilizan para intercontectar elementos de un diagrama. Los conectores pueden ser:

- *Composición secuencial* (*Secuence Flow*): definen una secuencia de flujo entre los elementos que conecta (origen y destino). Estos elementos pueden ser Actividades, Eventos o Compuertas dentro de un mismo Canal.
- *Mensajes*: representan la comunicación (interacción) entre diferentes entidades (Pools).
- *Asociaciones*: se utilizan para asociar información adicional sobre el proceso, es decir, para conectar artefactos.

A.2 Yet Another Workflow Language (YAWL)

Yet Another Workflow Language es un lenguaje para modelar y analizar *workflows* [96]. El mismo está basado en Redes de Petri (PN) [38] y entre una de sus principales características podemos destacar que sus modelos son producto de la composición de patrones de flujo de trabajos (*workflow patterns*). Estos patrones, construcciones genéricas y recurrentes en las descripciones de procesos de negocios, fueron identificados por la *Workflow Pattern Initiative*[2] con el objetivo de brindar bases conceptuales para la especificación de procesos de negocios [98].

A.2.1 Patrones de Flujos de Trabajo

Los patrones están agrupados según tres perspectivas conceptuales:

- Flujos de Control (*Control Flow patterns*): describen las principales características y operaciones para la especificación del flujo de control a través de las diferentes tareas que componen un proceso de negocio. Como elementos de este grupo, podemos encontrar los patrones correspondientes a la distribución del control (*branching*), los de sincronización, multiplicidad, cancelación, etc.
- Datos (*Data patterns*): detallan las características correspondientes a la representación y manipulación de la información que se necesita durante un proceso de negocio. Entre sus patrones podemos mencionar los referidos a visibilidad de información, interacción, transferencia y toma de decisiones. Estos últimos

[2]Más información en http://www.workflowpatterns.com

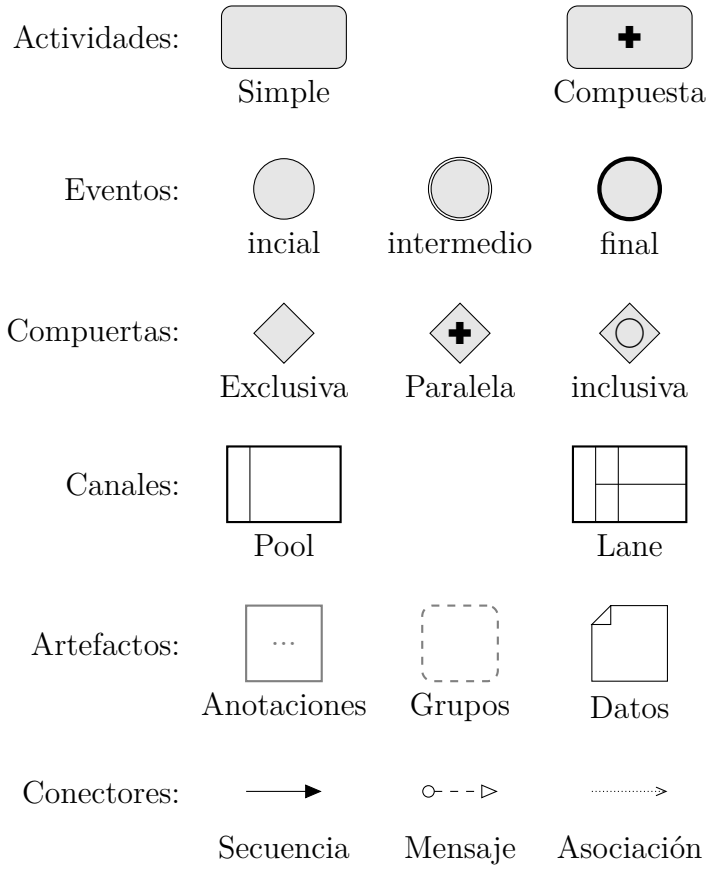


Figura A.1: Notación gráfica de BPMN

caracterizan la influencia de la información en otras perspectivas del procesos, como por ejemplo, en el flujo de control.

- Recursos (*Resource patterns*): estos patrones caracterizan las diferentes formas de agendar y distribuir los trabajos en los recursos que forman parte del proceso. Ejemplos de patrones son: formas de creación de trabajos, formas de distribución y de delegación de los mismos entre los recursos disponibles, visibilidad sobre trabajos pendientes, etc.

A.2.2 Lenguaje YAWL

Como consecuencia de la identificación de los patrones mencionados, han surgido trabajos de investigación [53, 111] en los cuales se analiza el poder expresivo y diferencias entre distintos lenguajes para la especificación de procesos de negocios, según su capacidad de abordaje a cada uno de los patrones, teniendo en cuenta la perspectiva de cada lenguaje estudiado.

La especificación de un proceso de negocio mediante el lenguaje YAWL consta de un conjunto jerárquicamente organizado de *redes* de las cuales una de ellas es la *raíz* o principal. Cada red está formada por *tareas* y *condiciones* relacionadas directamente o a través de *puertas* que reflejan distintos patrones de flujo de control. Las tareas pueden recibir, modificar y retornar información representada por medio de variables y representan algún tipo de actividad que contribuye al cumplir con el objetivo del proceso de negocio. Las *condiciones* son componentes del lenguaje que, si bien no están vinculadas a una actividad real del modelo, se utilizan para modelar estados en el flujo de control del proceso de negocio. Por ejemplo el comienzo y fin de un proceso o estados intermedios que reciben o distribuyen el control de tareas. YAWL provee diferentes tipos de tareas:

- Las tareas *atómicas* constituyen el nivel de descripción básico del sistema.
- Las tareas *compuestas* modularizan un sub-proceso y están asociadas con una red YAWL correspondiente que modela su comportamiento.
- Las tareas de *instancia múltiple* (MI) representan una colección de instancias de una actividad que se realizan concurrentemente.

Puertas y Regiones de Cancelación

Las tareas pueden tener asociadas (estar decoradas con) *puertas* que sirven como modificadores del flujo de control, implementando algunos de los patrones mencionados. Los patrones de flujo de control que provee YAWL para la especificación de sus modelos, cuya notación se muestra en Figura A.2, son los siguientes:

- Distribución del control (*branching*):
  - AND-split: cuando la puerta está habilitada, pasa el control a *todas* las tareas conectadas a la misma.
  - OR-split: cuando la puerta está habilitada, pasa el control a *al menos* una de las salidas conectadas.

  - XOR-split: cuando la puerta está habilitada, pasa el control a *exactamente una* de las salidas conectadas.

- Sincronización del control (*joining*):
  - AND-join: cuando *todas* las conexiones entrantes a este tipo de puerta están habilitadas, la tarea saliente asociada comienza su ejecución.
  - OR-join: cuando *al menos* una de las tareas o condiciones entrantes a la puerta está han terminado su ejecución, queda habilitada la tarea saliente para su ejecución.
  - XOR-join: la tarea o condiceón saliente asociada a esta puerta comienza su ejecución cuando *exactamente* una de las conexiones entrantes a la misma está habilitada.

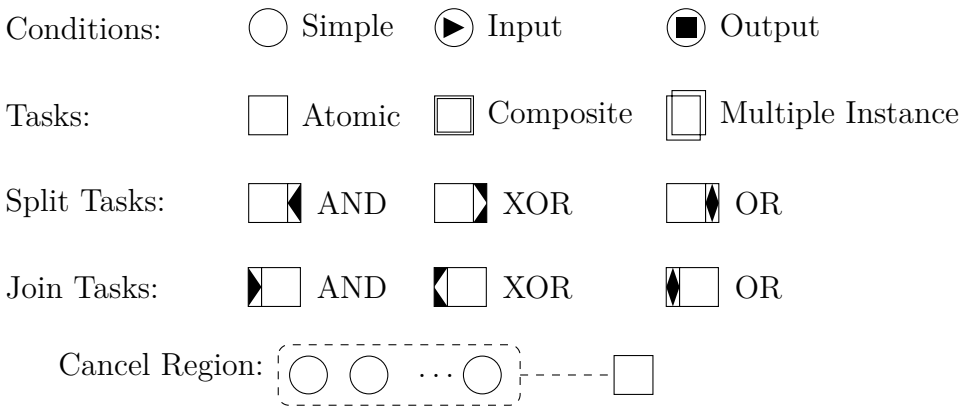


Figura A.2: Notación gráfica de YAWL

Además de las puertas, una tarea t puede tener asociada una *región de cancelación*, esto es, un conjunto de condiciones y tareas que serán abortadas cuando t finalice su ejecución.

A.2.3 Semántica formal de YAWL

En la Figura A.3 se muestra la semántica formal para cada elemento del lenguaje YAWK en términos de Redes de Petri con reinicio. Para ello se presentan, en cada caso, la notación en YAWL con su correspondiente Red de Petri que captura su semántica.

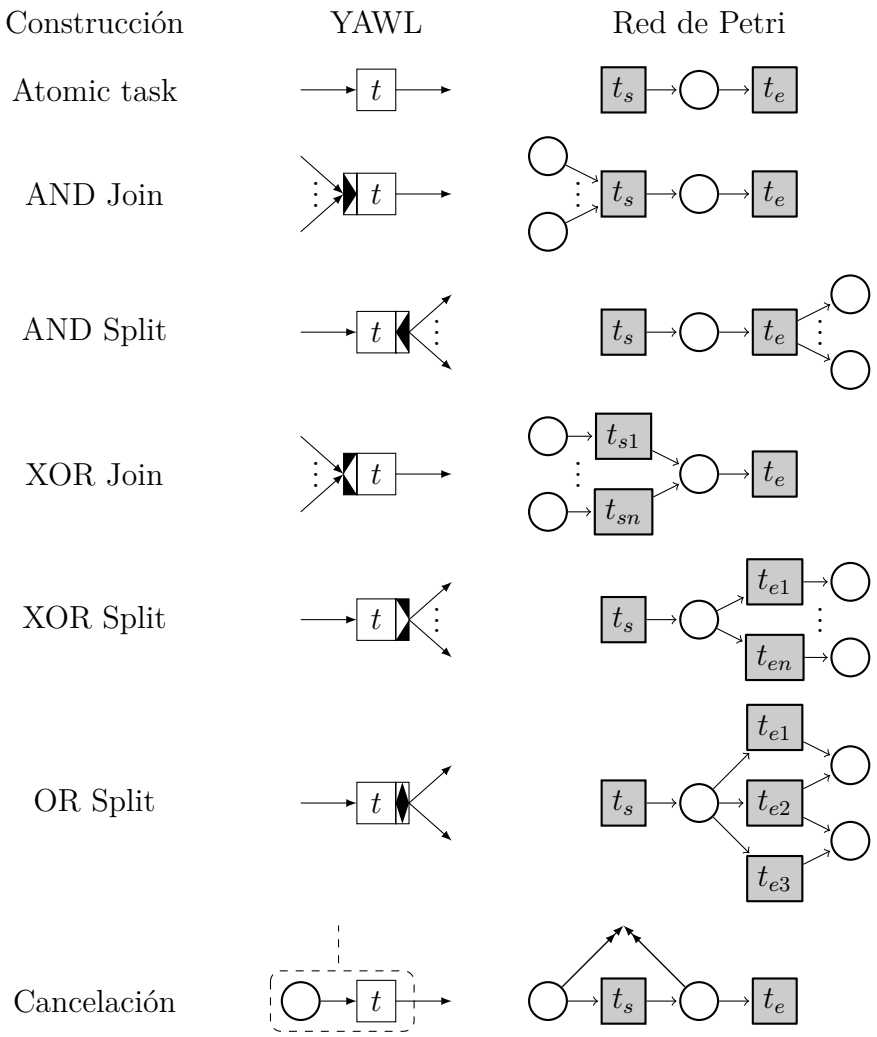


Figura A.3: Semántica de construcciones YAWL en términos de Redes de Petri con reinicio

A.3 Event-Driven Process Chain (EPC)

Event-Driven Process Chain (EPC) [52] es un lenguaje gráfico desarrollado a principios de los años 90 como parte del *framework Architecture of Integrated Information System (ARIS)* para la descripción de procesos de negocios. ARIS ofrece diferentes enfoques para el análisis y desarrollo de un sistema de información. Cada uno de estos enfoques: Organizacional, Funcional, Datos y Control, provee diferentes dia-

gramas específicos para describir, por ejemplo, diagrama de Árbol de Funciones para el enfoque Funcional, diagrama de Entidades y Relaciones para el enfoque de Datos, un Organigrama para describir las responsabilidades en el enfoque Organizacional. Dentro del enfoque de Control y con un rol integrador entre los distintos enfoques, se utiliza el diagrama EPC que describe la ejecución (flujo de control) del proceso.

Un diagrama EPC está compuesto por diferentes elementos cuya ilustración se puede observar en la Figura A.4 y que podemos agrupar como: *Objetos de Control*, *Objetos Complementarios* y *Conectores*.

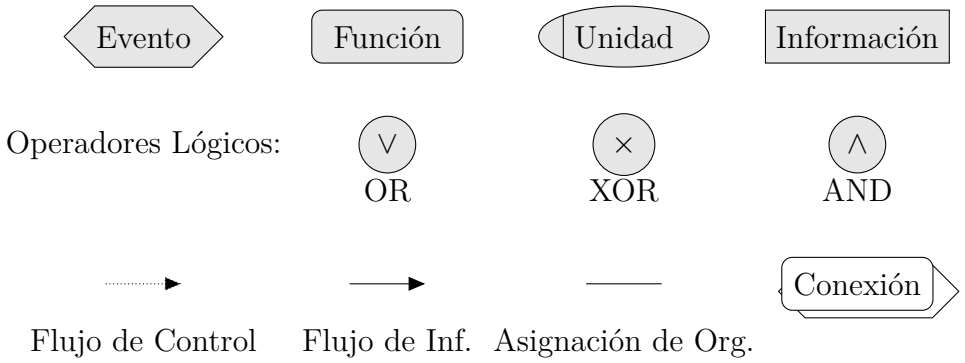


Figura A.4: Notación gráfica de EPC

A.3.1 Objetos de Control

Son elementos del lenguaje que describen sucesos o actividades que tienen lugar durante la realización del proceso. Los objetos de control son las *funciones*, *eventos* y *operadores lógicos*.

- *Funciones*: modelan actividades o tareas que son realizadas durante el procesos en pos de cumplir con los objetivos del mismo. Éstas definen transformaciones de estados. Las funciones pueden describir tareas complejas (desde un punto de vista descriptivo), en estos casos, cada una de esta funciones está refinada (detallada) por otro diagrama EPC.

- *Eventos*: describen condiciones sobre el proceso, en un sentido descriptivo son elementos pasivos que caracterizan estados del proceso. Estas condiciones definidas por los eventos pueden asumirse como precondiciones para las funciones subsiguientes o, como poscondiciones o resultados de funciones anteriores.

Dos eventos particulares, que necesariamente tienen lugar en cualquier modelo EPC, son los de *comienzo* y *fin*.

- *Operadores Lógicos*: se utilizan para modificar el flujo de control del proceso. EPC provee tres tipos de operadores: *AND* (paralelo), *XOR* (exclusivo), *OR* condicional.

  - AND: este operador puede utilizarse como sincronizador (*join*), esto es, cuando llegan al operador dos o más flujos de control (flechas), o como distribuidor (*split*) activando todos los flujos de salida del mismo.
  - XOR: en rol de distribuidor, activa exactamente uno de sus flujos de control salientes. De manera dual, como selector, sólo acepta uno de los posibles flujos que converjan en él.
  - OR: este operador tiene un comportamiento similar al XOR, pero sin la exclusividad, esto es, uno o más de sus flujos saliente o entrantes pueden ser activados.

A.3.2 Objetos Complementarios

Como mencionamos anteriormente EPC es el lenguaje para especificar uno de los enfoques (Control de Flujo) que provee ARIS para la descripción de procesos de negocios. Para los demás enfoques existen otros diagramas cuyos elementos pueden ser vinculados con los elementos de un diagrama EPC. Estos objetos complementarios aportan información adicional al flujo de control. Algunos de estos objetos complementarios son las *unidades de organización* (Recursos) y la descripción de la *información* (Datos) que las funciones manipulan.

- *Unidades de organización*: describen entidades del proceso (en una perspectiva jerárquica) responsables de ciertas funciones a las cuales son asignadas. Si bien no modifican, ni condicionan la ejecución (flujo de control) del proceso, aportan información relevante acerca de asignación de recursos para la realización de las tareas que intervienen en el proceso.

- *Información*: representan *información* que es necesaria (datos de entrada) o producida (datos de salida) de una función. Al igual que las organizaciones no alteran la ejecución del proceso pero aportan detalles sobre la información que manipula el proceso. Se vinculan con las funciones a través de flechas cuyo sentido ilustra si la información es de entrada o salida.

A.3.3 Conectores

Llamamos *conectores* a los elementos que provee el lenguaje como herramienta para relacionar tanto los objetos de control, definiendo el sentido del flujo entre los mismos, como así también para vincular objetos complementarios. Los conectores pueden ser:*conexiones (process path)*, *flujo de control*, *flujo de información* o *asignación de unidad de organización*.

- *Conexión (process path)*: asisten en la diagramación de un proceso, esto es, sirven como conectores o saltos entre dos procesos.
- *Flujo de Control o arcos*: se utilizan para relacionar *funciones*, *eventos*, *operadores lógicos* o *conexiones*. Los flujos de control definen la ocurrencia secuencial de los elementos que relaciona. Todos los objetos de control deben estar relacionados por algún flujo de control contemplando las siguientes restricciones:
  - Todo diagrama EPC comienza y termina con un evento, excepto en presencia de *conexiones*, donde éstas toman el lugar del evento final en en uno de los diagramas conectados y de comienzo en el subsiguiente.
  - Los eventos tienen como máximo un flujo entrante y a lo sumo uno saliente, pero al menos uno de ellos.
  - Las funciones tienen exactamente un flujo entrante y uno saliente.
  - Los operadores lógicos tienen dos configuraciones posibles, o bien un flujo entrante y múltiples salientes, o múltiples entrantes y uno saliente.
  - Las funciones y eventos deben estar intercalados, esto es, a cada evento le sigue necesariamente una función y cada a función un evento.
  - Los operadores lógicos pueden ser combinados con eventos y funciones pero respetando la alternación de los mismos. Además, dado que los eventos son elementos pasivos y no pueden intervenir en decisiones, una restricción semántica es que el único operador que puede seguir a un evento es un operador AND.
- *Flujo de Información*: vinculan la información que necesita o produce un función durante su ejecución. El sentido (de entrada o salida) se corresponde con el de la flecha que ilustra el flujo de información.
- *Asignación de Unidades de Organización*: se utilizan para asociar (especificar) recursos asignados a ciertas funciones.

A.4 Unified Modeling Language (UML)

Unified Modeling Language (UML) es un lenguaje gráfico (compuesto por diferentes diagramas) que fue desarrollado por Boock, Rumbaugh y Jacobson [17] para la organización, especificación, desarrollo y documentación de los diferentes componentes de un sistema. Gracias a su difusión y aceptación fue adoptado como un *estándar* para el modelado orientado a objetos por la *Objet Management Group* (OMG). Pese a su escasos fundamentos formales, la versatilidad del lenguaje puede apreciarse a través de sus numerosos diagramas que permiten abordar diferentes tipos y etapas en el desarrollo de sistemas. Estos diagramas pueden agruparse como *estructurales* o de *comportamiento*.

Los diagramas estructurales aportan una descripción de los aspectos estáticos del sistema. Entre sus diagramas podemos mencionar: Diagrama de Clases, Diagrama de Objetos, Diagrama de Componentes y Diagrama de Despliegue. Entre los diagramas que comportamiento, que describen aspectos dinámicos de del sistema, UML provee los siguientes diagramas: Diagrama de Casos de Uso, Diagrama de Secuencia, Diagrama de Colaboración, Diagrama de Estados y Diagrama de Actividades.

A.4.1 Diagrama de Actividades

Los diagramas de actividad describen el flujo de control a través de las diferentes actividades que conforman el sistema. Los elementos de que componen un diagrama de actividades son:

Objetos de Control

- *Actividades*: Las actividades representas los elementos del sistema que, mediante su ejecución, contribuyen a la realización del mismo. En los diagramas de Actividades de UML existe una separación conceptual entre *actividad* y *acción*, la primera hace referencia a un conjunto de operaciones o acciones cuya ejecución cambia el estado del sistema. Las actividades pueden estar representadas (detalladas) mediante otro diagrama de actividades. A diferencia de las actividades, las *acciones* representan la ejecución de una tarea *atómica* del sistema.

  Dos estados destacables de los sistemas en los diagramas de actividades son el estado *inicial* y *final* del sistema.

- *Decisión*: Los puntos de decisión permiten condicionar el flujo de control, es decir, dado el flujo de control entrante, éste puede trasladarse a *uno* de varios flujos salientes dependiendo del cumplimiento de las condiciones (expresión

lógica) que los identifica. Como restricción, las expresiones lógicas deben ser mutuamente excluyentes para lidiar con la ambigüedad y deben ser exhaustivas (al menos uno debe poder evaluarse como verdadero) para garantizar el progreso del sistema. Para asistir y garantizar este último punto, se puede utilizar la expresión *else* (otros) para hacer referencia a cualquier otra condición no expresada explícitamente en alguno de los flujos salientes. Para concluir este tipo de bifurcaciones del flujo de control, se utiliza el mismo operador de manera dual, es decir, confluyen en el todos los posibles flujos en uno

- *Bifurcación y Sincronización*: En pos de describir ciertas partes del sistema que puedan realizarse en paralelo, los diagramas de actividades proveen el concepto de barra de bifurcación (*fork, split*) o sincronización (*join*). La *barra de bifurcación* permite, dado un flujo de control entrante, crear o trasladar dicho control a *cada uno* (todos) los flujos salientes posibilitando así, la realización concurrente de actividades que pertenezcan diferentes flujos.

  De manera dual, existen las *barras de sincronización* que se utilizan para *unificar* varios flujos de control concurrentes en uno. Una barra de sincronización espera a que *todos* los flujos entrantes la alcancen para propagar el control al flujo saliente asociada a la misma.

- *Transiciones*: se utilizan para especificar el flujo entre las actividades y operadores de control definidos anteriormente de manera secuencial, esto es, cuando una actividad concluye su ejecución el control del proceso pasa (habilita) a su próximo objeto conectado mediante una *transición*.

Artefactos

- *Canales*: los canales se utilizan para representar las entidades o recursos que tienen la responsabilidad de llevar a cabo una o más actividades. Si bien no influyen en el flujo de control, aporta información útil a la hora de agrupar o distribuir recursos en el sistema. Como es de esperar, los nombres de los canales deber ser únicos y misma actividad no puede pertenecer a más de uno de ellos.

- *Objetos*: especifican los objetos (información) que las actividades manipulan en su ejecución. Las actividades pueden crean, modificar o eliminar objetos de flujos asociados a las mismas mediante flujos de objetos. Estos flujos, independientes de los de control, ilustran la secuencia de creación, modificación y/o eliminación de los objetos por parte de las actividades a las que están

asociadas. Los objetos especifican sus características mediante atributos que establecen su estado y cuyos valores pueden modificarse.

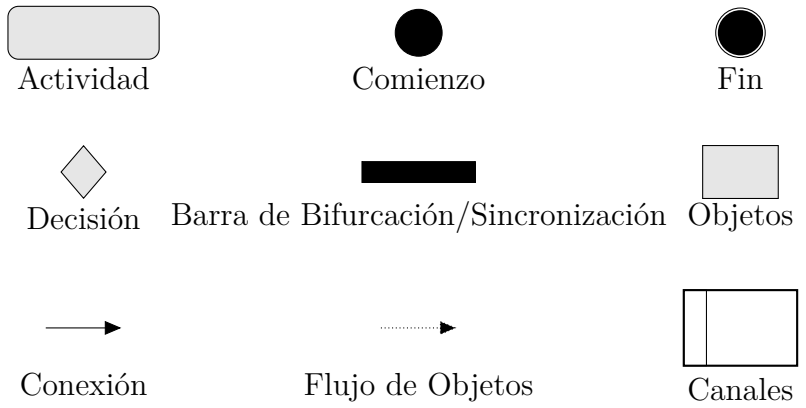


Figura A.5: Notación gráfica de Diagrama de Actividades UML