UNIVERSIDAD CARLOS III DE MADRID

DEPARTAMENTO DE INGENIERÍA TELEMÁTICA

TESIS DOCTORAL

DEFINICIÓN DE UN MODELO PARA LA VERIFICACIÓN FORMAL DE PROCESOS DE NEGOCIO

Autor: **Jesús Arias Fisteus**
Ingeniero de Telecomunicación

Director: **Carlos Delgado Kloos**
Doctor Ingeniero de Telecomunicación

Leganés, Septiembre de 2005

Tribunal nombrado por el Mgfco. y Excmo. Sr. Rector de la Universidad Carlos III de Madrid, el día ___ de ________________ de _____.

Presidente

Vocal

Vocal

Vocal

Secretario

Realizado el acto de defensa y lectura de la Tesis el día ___ de ________________ de _____ en _____________________.

Calificación:

EL PRESIDENTE EL SECRETARIO LOS VOCALES

Agradecimientos

En primer lugar, me gustaría agradecer a Carlos Delgado Kloos por la dirección de esta tesis, sin cuya ayuda no habría sido posible su realización. Este agradecimiento es extensible a quienes han aportado ideas o sugerencias valiosas para esta tesis en distintos momentos de su desarrollo: Abelardo, Andrés, Carlos (cgr), Luis y Vicente.

Gracias a todos mis compañeros del Departamento de Ingeniería Telemática, especialmente a M. Carmen, Celeste, Julio, Norberto, Pablo y Raquel. Gracias muy especiales a Iria por todos estos años que llevamos estudiando y trabajando juntos.

Y gracias a mi familia, sobre todo a mis padres y hermanos.

Resumen

Las tecnologías de gestión de procesos de negocio supusieron, en los años 90, una revolución en la forma de coordinar las actividades internas de organizaciones, principalmente de tamaño grande o mediano. Posteriormente, a principios de esta década, el ámbito de utilización de estas tecnologías comenzó su expansión a entornos B2B, como un facilitador de las colaboraciones entre distintas organizaciones. Ahora, están comenzando a ser aplicadas para realizar composición de servicios en entornos diseñados conforme al paradigma de la *computación orientada a servicios*. Aunque un proceso de negocio no deja de ser un programa de ordenador, la principal característica que lo diferencia de los programas convencionales es su muy alto nivel de abstracción. Este nivel de abstracción permite que el desarrollo, implantación o modificación de procesos sea considerablemente más ágil que en el caso de programas convencionales. Por otra parte, aporta una gran flexibilidad durante la ejecución de los mismos. Sin embargo, a pesar de este nivel de abstracción, también las definiciones de procesos de negocio son susceptibles de tener errores, especialmente debido al alto grado de concurrencia, computación distribuida y dependencia de entidades externas que las caracterizan. Esto hace necesario el uso de técnicas de verificación que permitan comprobar que el comportamiento de los procesos sea conforme a sus especificaciones.

En esta tesis doctoral se realizan aportaciones en el ámbito de la verificación de requisitos funcionales de procesos de negocio. Por una parte, se propone una arquitectura abierta, modular y extensible para la verificación de procesos, que permite la integración de distintos lenguajes de definición de procesos y herramientas de verificación, y se basa en un sistema formal intermedio. Por otra parte, se define este sistema formal intermedio, llamado *Modelo Formal Común* (CFM), que se basa en sistemas de estado–transición etiquetados, pero con una notación y abstracciones orientadas a la representación de procesos de negocio. Con el objetivo de comprobar la expresividad y adecuación de este formalismo para la representación de los mismos, se realiza un análisis basado en *patrones de workflow*. Adicionalmente, se integra BPEL4WS, un lenguaje de definición de composiciones de servicios Web basado en procesos, en la arquitectura. Para ello, se define su semántica en términos del formalismo, así como una metodología de transformación de definiciones de procesos BPEL4WS a definiciones CFM. También se integran en la arquitectura dos herramientas de verificación: los *model checkers* Spin y NuSMV. Para ello, se define una transformación entre definiciones CFM y los lenguajes de entrada de estas herramientas.

Abstract

Business process management technologies led, in the 90s, to a revolution in the way large and medium–sized organisations coordinated their internal activities. Later, at the beginning of this decade, these technologies begun to be expanded to B2B environments, facilitating collaborations among organisations. Now, they are being adapted and applied for composing services in environments based on the *service oriented computing* paradigm. The main difference between business processes and conventional programs is the high level of abstraction of the former. Therefore, the development, installation and modification of business processes is simpler and more agile. Furthermore, business processes are more flexible at run–time.

However, business process definitions can contain errors, despite their high level of abstraction. This is mainly caused by the high level of concurrency, distributed computation and dependencies with external entities that characterise these definitions. Verification techniques can be applied to process definitions in order to check if they fulfil their specifications.

In this PhD. Thesis we make contributions in the field of verification of functional requirements of business processes. Firstly, we propose an open, modular and extensible architecture for verifying processes. It is based on an intermediate formalism and allows the integration of different process definition languages and verification tools. Secondly, we define this intermediate formalism, called *Common Formal Model* (CFM). It is based on labelled state–transition systems, with a notation and abstractions close to the concepts used in activity–based process modelling. In order to check its adequacy and expressive power for representing business processes, we analyse it using the *Workflow Patterns*. Thirdly, we integrate the *Business Process Execution Language for Web Services* (BPEL4WS) in the proposed architecture, by defining its semantics in terms of the CFM, and designing a mapping between BPEL4WS and this formalism. Finally, we integrate two verification tools in the architecture: the model checkers Spin and NuSMV. In order to do so, we design a mapping between the CFM and the input language of each of them.

Contenidos

1. Introducción y objetivos **1**

1.1. Motivación de la tesis . . . 1

1.1.1. Tecnologías de gestión de procesos de negocio . . . 2

1.1.2. Sinergias entre procesos de negocio y computación orientada a servicios . . . 2

1.1.3. El papel de los servicios Web . . . 4

1.1.4. Robustez en los procesos de negocio . . . 4

1.2. Objetivos . . . 6

1.3. Plan de trabajo . . . 6

1.4. Historia de la tesis . . . 8

1.5. Estructura . . . 9

2. Estado del arte **11**

2.1. Métodos formales y *model checking* . . . 11

2.1.1. Introducción . . . 11

2.1.2. Especificación de sistemas . . . 12

2.1.3. Verificación de requisitos funcionales . . . 13

2.1.4. Lógicas temporales . . . 13

2.1.5. Model checking . . . 15

2.1.6. Técnicas avanzadas de *model checking* . . . 16

2.1.7. Herramientas de *model checking* . . . 20

2.2. Gestión de procesos de negocio . . . 20

2.2.1. Definición de procesos de negocio . . . 22

2.2.2. Lenguajes de definición de procesos de negocio . . . 25

2.2.3. Patrones de *workflow* . . . 27

2.2.4. Análisis de procesos de negocio . . . 28

2.3. Composiciones y coordinaciones de servicios Web . . . 28

2.3.1. Composición de servicios Web . . . 30

2.3.2. Coordinación de servicios Web . . . 31

2.3.3. El lenguaje BPEL4WS . . . 32

2.3.4. Relación entre BPML y BPEL4WS . . . 32

2.4. Trabajos relacionados . . . 33

2.4.1. Woflan (Technische Universiteit Eindhoven) . . . 33

2.4.2. Tesis doctoral de Eshuis (Universiteit Twente) . . . 34

2.4.3. Web Service Engineering Project (Imperial College London) . . . 36

2.4.4. Otros trabajos relacionados 37
2.5. Conclusiones 37

3. Definición de una arquitectura de verificación 41
3.1. Una arquitectura para la verificación de requisitos funcionales 42
3.2. Justificación 43
3.2.1. Perspectiva de datos 43
3.2.2. Expresividad de las especificaciones 43
3.2.3. Independencia del lenguaje de definición y formalismo de verificación 43
3.2.4. Número de herramientas de transformación necesarias 44
3.3. El modelo formal común 44
3.4. Conclusiones 45

4. El Modelo Formal Común 47
4.1. Sistema formal básico 48
4.1.1. Atributos y estado 48
4.1.2. Transiciones 48
4.1.3. Caminos y ejecuciones 49
4.1.4. Proceso 49
4.1.5. Transiciones funcionales 50
4.2. Recubrimiento de alto nivel 50
4.2.1. Conceptos añadidos al formalismo básico 50
4.2.2. Definición de transiciones funcionales 51
4.3. Definición de una notación para el *Modelo Formal Común* 53
4.3.1. Sintaxis y gramática 54
4.3.2. Restricciones adicionales 56
4.3.3. Especificaciones 57
4.4. Líneas generales de representación de procesos de negocio 59
4.4.1. Casos 60
4.4.2. Objetos manipulados 60
4.4.3. Variables 61
4.4.4. Actividades 62
4.4.5. Control de flujo 64
4.4.6. Ejecución secuencial de actividades 66
4.4.7. Ejecución concurrente de actividades 66
4.4.8. Ejecución condicional de actividades 67
4.4.9. Ejecución iterativa de actividades 68
4.4.10. Composición jerárquica de actividades 69
4.4.11. No–determinismo 70
4.5. Limitaciones 72
4.5.1. Limitaciones en la representación de atributos 72
4.5.2. Ausencia de entidades de ámbito local 74
4.5.3. Limitaciones en la comunicación entre distintos casos 76
4.5.4. Limitaciones en la instanciación múltiple de actividades 76
4.6. Conclusiones 77

5. Análisis del Modelo Formal Común basado en Patrones de Workflow 79
5.1. Patrones de *workflow* en CFM . . . 80
5.1.1. Patrones de control de flujo básico . . . 80
5.1.2. Patrones avanzados de selección y sincronización . . . 85
5.1.3. Patrones estructurales . . . 88
5.1.4. Patrones con instancias múltiples . . . 91
5.1.5. Patrones basados en estado . . . 92
5.1.6. Patrones de cancelación . . . 94
5.1.7. Conclusión . . . 96
5.2. Comparación con otros lenguajes de definición de procesos . . . 96
5.3. Conclusiones . . . 98

6. Verificación 101
6.1. Definición de especificaciones en trabajos relacionados . . . 102
6.2. Definición de especificaciones en CFM . . . 103
6.2.1. Notación . . . 104
6.2.2. Estructura de una especificación . . . 105
6.2.3. Tipos de requisitos . . . 105
6.2.4. Requisitos generales . . . 106
6.3. Verificación de procesos mediante Spin . . . 108
6.3.1. Transformación de CFM a Promela . . . 109
6.3.2. Verificación de los distintos tipos de requisitos . . . 112
6.4. Verificación de procesos mediante NuSMV . . . 119
6.4.1. Transformación de CFM a SMV . . . 119
6.4.2. Verificación de distintos tipos de requisitos . . . 121
6.5. Conclusiones . . . 123

7. Integración de BPEL4WS en la arquitectura 125
7.1. Introducción a BPEL4WS . . . 126
7.1.1. Representación y manejo de datos . . . 126
7.1.2. El proceso . . . 126
7.1.3. Actividades . . . 127
7.1.4. Manejo de fallos y compensación . . . 130
7.2. Semántica de BPEL4WS basada en CFM . . . 132
7.2.1. Variables . . . 133
7.2.2. Modelado de entidades externas . . . 135
7.2.3. Actividades . . . 136
7.2.4. Expresiones XPath . . . 139
7.2.5. Procesos . . . 140
7.2.6. Actividad *assign* . . . 141
7.2.7. Actividad *sequence* . . . 142
7.2.8. Actividad *flow* . . . 143
7.2.9. Actividad *switch* . . . 144
7.2.10. Actividad *while* . . . 145
7.2.11. Actividad *receive* . . . 146

7.2.12. Actividad *pick* 148
7.2.13. Actividad *reply* 150
7.2.14. Actividad *empty* 151
7.2.15. Actividad *wait* 151
7.2.16. Actividad *scope* 151
7.2.17. Actividad *invoke* 152
7.2.18. Cancelación 153
7.2.19. Actividad *terminate* 154
7.2.20. Manejo de fallos 155
7.2.21. Actividad *throw* 159
7.2.22. Modelado de vínculos (*links*) 159
7.2.23. Compensación 162
7.2.24. Actividad *compensate* 165
7.2.25. Manejo de eventos 167
7.3. Especificaciones 169
7.3.1. Definición de requisitos 169
7.3.2. Funciones añadidas al lenguaje de expresiones 170
7.3.3. Requisitos específicos de esta metodología de transformación . . . 171
7.3.4. Ejemplos 171
7.4. Limitaciones 171
7.4.1. Limitaciones en la representación de variables 171
7.4.2. Limitaciones en la representación de manejadores de eventos 172
7.4.3. Limitaciones en el sistema de gestión de compensación 173
7.4.4. Limitaciones en la representación de *scopes* serializables 174
7.5. Conclusiones 174
7.6. Código de los ejemplos 175

8. Caso de estudio **193**
8.1. Planteamiento del caso de estudio 193
8.2. Diseño inicial del proceso 194
8.2.1. Interfaces WSDL y tipos de datos 194
8.2.2. El proceso BPEL4WS 194
8.2.3. Abstracción de datos 196
8.2.4. Modelado de las entidades externas 197
8.3. Análisis de requisitos generales 198
8.3.1. Planteamiento de los requisitos 198
8.3.2. Verificación de los requisitos generales en el ejemplo 199
8.3.3. Ciclos sin progreso 200
8.4. Análisis de requisitos específicos 201
8.4.1. Requisitos que no se cumplen 202
8.5. Análisis de requisitos de lógica temporal 202
8.5.1. Un requisito temporal que se cumple 203
8.5.2. Un requisito temporal que no se cumple 203
8.6. Conclusiones 204

9. Conclusiones y trabajos futuros **209**
9.1. Conclusiones . . . 209
9.2. Principales contribuciones . . . 210
9.2.1. Estudio de los trabajos más relevantes en el campo de la verificación de requisitos funcionales de procesos de negocio y composiciones de servicios Web . . . 211
9.2.2. Propuesta de una arquitectura de verificación abierta, modular y extensible para procesos de negocio y composiciones de servicios Web 211
9.2.3. Propuesta de un sistema formal para la representación de procesos de negocio y composiciones de servicios Web, desde el punto de vista de las perspectivas de control del flujo y datos . . . 212
9.2.4. Identificación de tipos de requisitos útiles en la definición de especificaciones de procesos de negocio y composiciones de servicios Web . . . 212
9.2.5. Aplicación de dos herramientas de *model checking* a la verificación de procesos de negocio y composiciones de servicios Web . . . 213
9.2.6. Propuesta de un mecanismo de transformación de procesos BPEL4WS a modelos verificables . . . 213
9.2.7. Propuesta de un mecanismo para la definición de especificaciones de procesos BPEL4WS . . . 213
9.3. Trabajos futuros . . . 214
9.3.1. Integración de otros lenguajes de definición y herramientas de verificación . . . 214
9.3.2. Verificación de procesos de negocio concurrentes que se comunican 214
9.3.3. Mejora de CFM para que admita instanciación múltiple de atributos 215
9.3.4. Análisis de rendimiento de las verificaciones . . . 215
9.3.5. Desarrollo de técnicas de reducción de complejidad de definiciones CFM . . . 216
9.3.6. Representación de *scopes* serializables de BPEL4WS mediante CFM 216
9.3.7. Desarrollo de un entorno de verificación completo y funcional . . . 217

Referencias **219**

A. Caso de estudio: código completo **229**
A.1. Proceso de pedido de aceite de oliva (diseño inicial) . . . 230
A.2. Proceso de pedido de aceite de oliva (traducción a CFM) . . . 233
A.3. Proceso de pedido de aceite de oliva (traducción a Promela) . . . 243

B. Descripción del prototipo **247**

Índice de figuras

2.1. Proceso de gestión de reclamaciones modelado con redes de Petri (extraído de [2]). Los círculos representan *lugares*, y los cuadrados, *transiciones*. En el formalismo Workflow Nets las actividades del proceso se representan como transiciones. 26

3.1. Arquitectura del entorno de verificación propuesto. 42

4.1. Ejemplo de un ciclo de vida sencillo de una actividad, con sólo cuatro estados posibles. 62

5.1. Ejemplo de diagrama de flujo de un ciclo arbitrario (patrón 10). 89

6.1. Estructura básica de la definición de un proceso y su especificación en Promela. 109
6.2. Ejemplo de transformación de entidades y tipos de entidades del formalismo CFM a lenguaje Promela. 111
6.3. Ejemplo de transformación de entidades CFM con valores iniciales alternativos a Promela. El tipo de entidad *OilOrder* se declara en el ejemplo de la figura 6.2. 112
6.4. Estructura básica del elemento *proctype* y de la transformación de transiciones funcionales. 113
6.5. Ejemplo de transformación de transiciones funcionales a Promela. 114
6.6. Verificación de invariantes con Spin. 115
6.7. Verificación de requisitos de tipo *objetivo* con Spin. 116
6.8. Verificación de requisitos de tipo *pre–requisito* y *post–requisito* con Spin. . 117
6.9. Transformación de una transificón funcional marcada mediante el atributo *progress*. 118
6.10. Estructura básica de la definición de un proceso y su especificación en SMV. 119
6.11. Ejemplo de transformación de entidades y tipos de entidades del formalismo CFM a lenguaje Promela. 121
6.12. Ejemplo de transformación de transiciones funcionales a SMV. 122

7.1. Modelado de mensajes alternativos en una actividad *receive*. 136
7.2. Ciclo de vida general (izda.) y simplificado (dcha.) de las actividades BPEL4WS. 137
7.3. Tipos de entidad que representan el ciclo de vida de actividades. 137
7.4. Evolución de la actividad *switch* 144

7.5. Ejemplo de transformación de variables BPEL4WS. 176
7.6. Ejemplo de transformación de una actividad *assign*. 177
7.7. Ejemplo de transformación de una actividad *sequence*. 178
7.8. Ejemplo de transformación de una actividad *flow*. 179
7.9. Ejemplo de transformación de una actividad *switch*. 180
7.10. Ejemplo de transformación de una actividad *while*. 181
7.11. Ejemplo de transformación de una actividad *receive*. 182
7.12. Fragmento BPEL4WS del ejemplo de transformación de una actividad *pick*. 183
7.13. Representación mediante CFM del ejemplo de transformación de una actividad *pick*. 184
7.14. Ejemplo de transformación de una actividad *invoke*. 185
7.15. Procedimiento de cancelación. 186
7.16. Representación de la actividad *terminate*. 187
7.17. Proceso BPEL4WS tomado como ejemplo para ilustrar la representación del mecanismo de manejo de fallos. 188
7.18. Ejemplo de manejo de fallos en CFM. 189
7.19. Representación de las cuatro actividades *throw* del proceso representado en la figura 7.17. 190
7.20. Representación de vínculos (*links*). 191
7.21. Representación de manejadores de eventos. 192
7.22. Ejemplo de transformación de requisitos BPEL4WS a requisitos CFM. . . . 192

8.1. Fragmento de la interfaz WSDL del proveedor de aceite. 195
8.2. Fragmento de la interfaz WSDL del proceso de gestión de pedidos de aceite de oliva. 196
8.3. Fragmento de la interfaz WSDL de control del proceso de gestión de pedidos de aceite de oliva. 197
8.4. Mensajes de entrada alternativos en el proceso de pedido de aceite de oliva. 206
8.5. Requisitos generales aplicados al proceso ejemplo. 207
8.6. Requisitos específicos para el proceso de pedidos de aceite de oliva. 207
8.7. Requisito de lógica temporal expresado mediante XPath, CFM y Promela. . 208

Índice de cuadros

2.1. Definiciones de los principales conceptos relacionados con procesos de negocio. Los cuatro primeros se definen en un estándar del WfMC [126]. El último, en [10]. 21
2.2. Definiciones de los principales conceptos aplicados en metodologías de modelado de procesos basadas en actividad. Se puede encontrar una definición formal de los mismos en [126]. 24
2.3. Resumen de trabajos relacionados. 40

5.1. Comparación de CFM con BPEL4WS, XLANG, WSFL y BPML. Fuentes: [125, 6]. 97
5.2. Comparación de CFM con XPDL, redes de Petri de alto nivel (HL-PN) y YAWL. Fuentes: [8]. 98
5.3. Comparación de CFM con otros lenguajes de definición de procesos de negocio presentados en la literatura. Fuentes: [8]. 99

6.1. Comparación de la capacidad de Spin y NuSMV para verificar de forma sencilla los distintos tipos de requisitos. 123

7.1. Actividades de BPEL4WS 1.1. Las actividades del primer grupo son básicas, mientras que las del segundo son estructuradas. 128

B.1. Estadística de número de líneas del prototipo *verbus-j*. 248

Capítulo 1

Introducción y objetivos

En este capítulo se realiza una introducción a esta tesis doctoral. En primer lugar, se plantean las motivaciones que han dado lugar a la elaboración de la tesis. A continuación, se presentan de forma concisa los objetivos de la misma. Posteriormente, se describe el plan de trabajo planteado para su elaboración. A continuación, se describe de forma breve la evolución seguida para el desarrollo de la tesis. Finalmente, se describe la organización del resto de capítulos de este documento.

1.1. Motivación de la tesis

Los sistemas de información han sufrido una gran evolución desde los inicios de la informática [10]. En los años sesenta, consistían en aplicaciones hechas a medida ejecutándose sobre un sistema operativo de funcionalidad muy limitada. Desde entonces, estos sistemas han ido evolucionando de forma continua. Por una parte, los sistemas operativos ofrecen cada vez una mayor funcionalidad. Por otra parte, se han desarrollado aplicaciones genéricas y reutilizables que implementan funcionalidades que previamente residían por completo en aplicaciones finales hechas a medida. Estas nuevas aplicaciones pueden clasificarse, según su rango de reutilización, en dos grandes grupos: aplicaciones genéricas reutilizables en un rango amplio de dominios de aplicación (sistemas de gestión de bases de datos, editores de texto y documentos, hojas de cálculo, etc.), y aplicaciones reutilizables para dominios específicos (sistemas de gestión de recursos humanos, sistemas de gestión de relaciones con el cliente, etc.)

Como consecuencia de esta evolución, los sistemas de información actuales se pueden ver como un conjunto de diversas aplicaciones y subsistemas, que deben ser integrados y coordinados para resolver las necesidades de cada organización.

Por otra parte, esto ha facilitado que los sistemas de información hayan comenzado a transformarse desde los años noventa desde una orientación centrada en datos (basadas en el almacenamiento y recuperación de datos) hacia una orientación centrada en procesos (con modelado explícito de los procesos que controlan la coordinación de las distintas aplicaciones del sistema).

1.1.1. Tecnologías de gestión de procesos de negocio

En este contexto, se han desarrollado los sistemas de gestión de procesos de negocio (*Business Process Management Systems, BPMS*) y los sistemas de gestión de *workflow* (*WorkFlow Management Systems, WfMS*), como sistemas reutilizables en distintos dominios que evitan programar la lógica de un proceso directamente en aplicaciones a medida. Un proceso de negocio se define, en general, como un conjunto de actividades interrelacionadas con un objetivo común. El componente principal de estos sistemas es un *motor de workflow*, que no es más que un intérprete que ejecuta procesos de negocio definidos utilizando un lenguaje determinado. Por tanto, se puede ver estos sistemas como entornos programables en los que la programación se realiza a un nivel muy superior al de los lenguajes de programación convencionales. Estos entornos permiten disminuir de forma considerable el tiempo de diseño, desarrollo, instalación, mantenimiento y modificación de lógicas de negocio complejas.

La necesidad que dio lugar al nacimiento de las tecnologías de procesos de negocio fue, principalmente, la automatización y coordinación de tareas, así como el intercambio de documentos electrónicos, en entornos de oficina. Esta tecnología estaba muy orientada a combinar tanto tareas manuales realizadas por personas como tareas automatizables, llevadas a cabo por máquinas, normalmente aplicaciones de ordenador. Estas tecnologías han ido evolucionando en los últimos quince años, pero manteniendo el enfoque inicial. Se utilizan, sobre todo, en empresas grandes y medianas. Su mercado continúa creciendo en la actualidad. Tal y como se justifica en [94], representan una de las tecnologías clave, actualmente, para la realización de interacciones B2B. Las grandes empresas en desarrollo de aplicaciones empresariales integran tecnologías de gestión de procesos en sus productos, como por ejemplo SAP en *NetWeaver*, IBM en *Lotus Workflow* y *WebSphere MQ Workflow* y Fujitsu en *i-Flow*. Compiten también en este mercado algunas empresas especializadas, de menor entidad, con productos como, por ejemplo, *Staffware* de Tibco, *Reactor* de Oak Grove Systems o *COSA Workflow* de COSA Solutions.

El auge de la investigación en procesos de negocio se produjo en la década de los 90. Durante esos años eran objeto principal de investigación los sistemas de gestión de procesos de negocio centralizados. En ellos, un servidor central coordina todas las instancias de los procesos de negocio de una organización, sin perjuicio de que las actividades se ejecuten en otras máquinas comunicadas por red. Este paradigma comenzó a cambiar a finales de los 90, debido principalmente al crecimiento del comercio electrónico B2B (*Business to Business*) y al desarrollo de *middleware* de comunicaciones como CORBA. En este caso, la solución centralizada ya no resulta aceptable, dado que cada organización involucrada en la colaboración de negocio desea mantener, normalmente, el control de sus sistemas. La solución, entonces, pasa por que cada organización disponga al menos de un servidor, de tal forma que todos los servidores involucrados en un determinado proceso se coordinen entre sí como iguales.

1.1.2. Sinergias entre procesos de negocio y computación orientada a servicios

Actualmente se está consolidando la utilización del paradigma de *computación orientada a servicios* (*service oriented computing*, SOC) [74]. Este paradigma promueve un modelo de computación distribuida basado en la existencia de proveedores de servicios, consumidores de servicios y registros para la publicación y búsqueda de servicios. Su ámbito de

aplicación son tanto entornos internos a una organización como entornos de computación colaborativa entre distintas organizaciones. Las principales características que debe cumplir una arquitectura que dé soporte a este paradigma son, según se expone en [74]:

- Acoplamiento débil entre proveedores y consumidores de servicios.
- Independencia con respecto a la implementación (lenguajes de programación, bibliotecas de código, plataformas de ejecución, etc.)
- Configuración flexible y dinámica.
- Robustez y fiabilidad.
- Modelado de interacciones a un nivel de abstracción alto.
- Colaboración.

Dado el planteamiento de los *servicios* como los elementos básicos de este tipo de arquitecturas, una de las extensiones naturales al concepto es el de *composición de servicios*. Una entidad *software* puede proveer un servicio compuesto, basado en una combinación adecuada de otros servicios. Esto es, la entidad actúa al mismo tiempo como proveedora de un servicio y como consumidora de otros servicios.

Existe una clara relación de sinergia entre las tecnologías de procesos de negocios y la computación orientada a servicios. Por una parte, los problemas a los que se enfrenta la composición de servicios son similares a algunos de los problemas afrontados previamente en el desarrollo de las tecnologías de procesos de negocio. Las soluciones planteadas en este último campo a dichos problemas suponen una ayuda importante a la composición de servicios. Por otra parte, la propia computación orientada a servicios plantea un paradigma adecuado para el desarrollo e implantación de sistemas de gestión de procesos de negocio. Se profundiza en ambos aspectos a continuación.

La composición de servicios y los procesos de negocio tienen claramente algunos aspectos en común. Los procesos de negocio son, en esencia, una composición de actividades que, de forma conjunta, colaboran para alcanzar un objetivo común. Si se sustituye el concepto de *actividad* por el de *servicio*, se obtiene la propia definición del concepto de composición de servicios. Debido a ello, los formalismos y lenguajes desarrollados para definir la perspectiva de control del flujo de procesos de negocio pueden ser aplicados, con algunas modificaciones, a la composición de servicios. A pesar de esta similitud, cabe destacar que hay aspectos fundamentales de las tecnologías de procesos de negocio, como la participación de personas en los procesos, que no son aplicables a las composiciones de servicios. Por otra parte, también hay problemas específicos de la composición de servicios, como el manejo de datos, que las tecnologías de procesos de negocio no resuelven adecuadamente.

Las arquitecturas orientadas a servicios son un mecanismo adecuado para la implementación de sistemas de gestión de procesos de negocio, tanto en entornos internos a una organización, como en entornos colaborativos. En los primeros, facilitan la integración de las distintas aplicaciones de la organización, que pueden presentar altos grados de heterogeneidad, en el propio sistema de gestión de procesos. En los últimos, facilitan la ejecución descentralizada de procesos entre varias organizaciones, realizando todas las interacciones como invocaciones a servicios.

1.1.3. El papel de los servicios Web

Los *servicios Web* [13] son una tecnología de soporte a la computación distribuida nacida a principios de esta década. Se basan en la descripción de los servicios mediante el estándar WSDL, publicación de los mismos en repositorios UDDI y acceso a los servicios mediante el protocolo SOAP, aunque actualmente no existe consenso acerca de hasta qué punto son estas tecnologías concretas las que definen el concepto de servicio Web. Dadas sus características, resultan una tecnología adecuada, aunque no ideal, por el momento, para la implementación de arquitecturas orientadas a servicios.

Dada esta íntima relación con el paradigma de computación orientada a servicios, comparten con este las sinergias explicadas anteriormente con respecto a las tecnologías de gestión de procesos de negocio.

Más concretamente, existen varias soluciones para el problema de la composición de servicios basadas en la tecnología de servicios Web. De entre ellas, el lenguaje BPEL4WS (*Business Process Execution Language for Web Services*) destaca por el consenso alcanzado entre las principales empresas del sector. Este lenguaje es un claro exponente de la relación existente entre procesos de negocio, servicios Web y computación orientada a servicios.

Por otra parte, el éxito actual de los servicios Web, unido al éxito de BPEL4WS como primera aproximación a la composición de servicios en arquitecturas orientadas a servicios, están fomentando la adopción de tecnologías basadas en procesos, aunque con unos objetivos distintos a los originales. En este caso, el modelo no contempla de forma explícita la participación de personas en la ejecución de los procesos, sino que son los servicios los que ejecutan las tareas del proceso, fundamentalmente de forma automática.

1.1.4. Robustez en los procesos de negocio

La robustez del sistema es uno de los factores clave que pueden influir en que una organización decida o no automatizar tareas, que hasta entonces hayan sido ejecutadas de forma manual, mediante la implantación de sistemas de gestión de procesos de negocio o de composición de servicios. El concepto de robustez es amplio. Se refiere tanto a aspectos de seguridad (confidencialidad, integridad y disponibilidad, principalmente), como a aspectos funcionales, esto es, que el sistema se comporte tal y como se espera. Con respecto a los aspectos funcionales, es especialmente importante que no se produzcan errores (entendidos como un comportamiento distinto del que se espera según su especificación) y que el sistema sea capaz de reaccionar de forma correcta ante cualquier situación anómala (situaciones no previstas por el diseñador de los procesos o composiciones, ausencia de conectividad en uno o más equipos, parámetros erróneos en la invocación de servicios, caída de uno o más equipos, retardos excesivos en la respuesta de algunos equipos, etc.)

Uno de los factores que influyen determinantemente en la robustez de un sistema es la propia robustez de las definiciones de los procesos de negocio y composiciones de servicios. Al igual que un error de programación en un programa convencional suele provocar que el comportamiento de la máquina que lo ejecuta no sea el esperado, lo mismo ocurre con los errores en las definiciones de los procesos: un error puede provocar un comportamiento no deseado en el sistema de gestión de procesos.

Los procesos de negocio se caracterizan por un alto grado de concurrencia, computación distribuida y dependencia de entidades externas. Esto contribuye a que sean susceptibles de

tener errores, tales como, entre otros, posibilidad de abrazo mortal, comportamiento incorrecto en presencia de errores en el envío de mensajes, condiciones de carrera, terminación indebida del proceso, actividades que en ningún caso pueden ser ejecutadas, etc. Este tipo de errores puede ser especialmente crítico cuando los procesos de negocio se apliquen en entornos de comercio electrónico que supongan un movimiento importante de dinero.

Asimismo, los lenguajes de definición de procesos son cada vez más complejos y, por tanto, propensos a la ocurrencia de errores. Esta complejidad creciente viene dada, por ejemplo, por la posibilidad de definir variables con tipos de datos complejos o por la introducción de mecanismos de gestión de fallos y compensación. Por otra parte, la definición explícita en el proceso de operaciones ejecutadas remotamente da lugar a otro tipo de situaciones que un proceso debe gestionar adecuadamente, como por ejemplo la ocurrencia de errores de conectividad de red, ausencia de respuesta por parte de sistemas remotos, ocurrencia de errores en sistemas remotos, operaciones remotas que pueden lanzar *fallos*, etc. El crecimiento de la complejidad provoca que el número de comportamientos posibles de un proceso sea considerablemente mayor, y hace más difícil la detección de errores por inspección visual o por mecanismos manuales como simulación guiada y pruebas.

Esto justifica la necesidad de disponer de mecanismos automatizados que permitan comprobar la corrección de las definiciones de procesos de forma sistemática.

En otros dominios, como la ingeniería del software, electrónica digital o telemática se utilizan desde hace décadas diversas técnicas para detectar errores en programas *software*, circuitos electrónicos digitales o protocolos de comunicaciones, entre otros. Muchas de estas técnicas pueden ser aplicadas también en el dominio de los procesos de negocio, con el objetivo de detectar errores en etapas tempranas de diseño, antes de su implantación en el sistema de producción. Una de estas técnicas es la verificación de requisitos funcionales, que consiste en la aplicación de métodos formales para demostrar, mediante artilugios matemáticos, la presencia o ausencia de determinados tipos de errores.

En el ámbito de los procesos de negocio, se han llevado a cabo pocos trabajos relativos al análisis y, más en concreto, la verificación de definiciones de procesos, sobre todo si se compara con el esfuerzo dedicado a otros aspectos como el modelado de procesos o el diseño de arquitecturas centralizadas y distribuidas de motores de ejecución de procesos. Tampoco los productos de gestión de procesos de negocio comerciales suelen incorporar herramientas que permitan llevar a cabo verificaciones sobre los procesos definidos. La verificación de procesos de negocio es un tema que todavía permanece abierto y prueba de ello es, por ejemplo, que el tema principal de la primera edición del congreso *International Conference on Business Process Management (BPM 2003)*[1] haya sido *sobre la aplicación de métodos formales a sistemas de información basados en procesos*[2]. Por otra parte, el problema de la verificación de requisitos funcionales de procesos de negocio está muy relacionado con la verificación de requisitos funcionales de coordinaciones y composiciones de servicios Web, y por tanto las soluciones propuestas son, en general, aplicables en ambos entornos.

En esta tesis pretendemos realizar aportaciones en los ámbitos de la verificación de requisitos funcionales de procesos de negocio y composiciones de servicios Web. En concreto, proponemos una arquitectura de verificación más abierta, modular y extensible que las desarrolladas hasta ahora en otros trabajos de investigación relacionados. En esta arquitectura

[1] http://tmitwww.tm.tue.nl/bpm2003/

[2] On the Application of Formal Methods to Process–Aware Information Systems.

pretendemos adaptar las definiciones de procesos de negocio con el objetivo de que sean verificables mediante distintas herramientas de verificación desarrolladas previamente en el ámbito de los métodos formales.

1.2. Objetivos

El objetivo de esta tesis es proponer un entorno de verificación de procesos de negocio y composiciones de servicios Web que solucione los problemas identificados en otros trabajos relacionados, analizados en el capítulo 2: ausencia de soporte para la perspectiva de datos de los procesos de negocio; flexibilidad reducida para expresar requisitos específicos, a medida de cada proceso de negocio; soluciones ligadas a lenguajes de definición de procesos y formalismos de verificación específicos. Más concretamente, se pretende realizar las siguientes contribuciones:

- Una arquitectura abierta, modular y extensible para la verificación de procesos de negocio y composiciones de servicios Web. Su diseño debe estar orientado a solucionar los problemas identificados en los trabajos relacionados. El resto de las contribuciones de la tesis deben estar basadas en la arquitectura.

- Un sistema formal de representación de procesos de negocio y composiciones de servicios Web, que conforme el núcleo de la arquitectura. Debe actuar como formalismo intermedio entre lenguajes de definición de procesos y herramientas de verificación. Más concretamente, debe cumplir los siguientes requisitos:

  - Su definición debe ser formalmente sólida.
  - Debe tener la expresividad suficiente para representar procesos modelados mediante los principales lenguajes de definición de procesos de negocio.
  - Debe utilizar, en su notación, conceptos lo más cercanos posible a los empleados en modelado de procesos basado en actividad.
  - Debe ser representable mediante los sistemas formales de las principales herramientas de verificación.

- Integración de lenguajes de definición de procesos de negocio en la arquitectura. Dada su importancia actualmente, se dará prioridad al lenguaje BPEL4WS.

- Integración de herramientas de verificación en la arquitectura. Se dará prioridad a las herramientas más habitualmente utilizadas en el ámbito académico: NuSMV y Spin.

1.3. Plan de trabajo

Para llevar a cabo los objetivos propuestos se ha seguido el siguiente plan de trabajo.

1. Realizar un estudio del estado del arte, con especial énfasis en trabajos relacionados con la verificación de requisitos funcionales de procesos de negocio y composiciones de servicios Web. Analizar los problemas de las soluciones existentes.

2. Estudiar los lenguajes de definición de procesos de negocio más utilizados, así como las necesidades que se pueden plantear desde el punto de vista de su verificación.

3. Estudiar técnicas y herramientas de verificación, y determinar cuáles resultan más adecuadas para las necesidades planteadas en el ámbito de los procesos de negocio.

4. Proponer una arquitectura abierta, modular y extensible para la verificación de procesos de negocio y composiciones de servicios Web, orientando el diseño a la resolución de los problemas identificados en los trabajos relacionados.

5. Definir un sistema formal intermedio para la arquitectura:

   a) Estudiar los distintos tipos de sistemas formales.

   b) Seleccionar el sistema formal más apropiado, y adaptarlo a las necesidades planteadas en el entorno de los procesos de negocio.

   c) Definir matemáticamente el sistema formal.

   d) Definir una notación para el sistema formal, adaptada a la notaciones habituales de modelado basado en actividad.

   e) Integrar un formalismo y una notación para la definición de especificaciones (esto es, propiedades a verificar).

   f) Establecer unas líneas generales de representación de procesos de negocio mediante el sistema formal.

6. Integrar lenguajes de definición de procesos de negocio en la arquitectura. Se seleccionará como referencia el lenguaje BPEL4WS:

   a) Definir una transformación de procesos definidos mediante BPEL4WS a procesos definidos mediante el sistema formal, como una concreción de las líneas generales de representación.

   b) Definir una notación para la definición de especificaciones de procesos BPEL4WS, y una transformación de las mismas a especificaciones del sistema formal.

7. Integrar herramientas de verificación en el sistema:

   a) Seleccionar herramientas de verificación adecuadas para ser integradas en el entorno.

   b) Definir una transformación de definiciones de procesos basadas en el formalismo a definiciones basadas en cada una de las herramientas seleccionadas.

   c) Definir una transformación de especificaciones expresadas mediante el sistema formal a especificaciones expresadas mediante el formalismo de cada una de las herramientas seleccionadas.

8. Validar el sistema propuesto:

   a) Desarrollar un prototipo del sistema.

b) Evaluar la expresividad del sistema formal desde el punto de vista de los *patrones de workflow* (véase el apartado 2.2.3), con el objetivo de comprobar su capacidad para representar distintos lenguajes de definición de procesos.

9. Redactar la tesis.

1.4. Historia de la tesis

En este apartado se presentan brevemente las distintas etapas en la evolución de esta tesis. Se comentan también las publicaciones realizadas en cada momento de resultados directamente relacionados con la tesis.

Este trabajo comienza a principios de 2002 como un estudio de los distintos lenguajes de modelado de procesos de negocio, así como de tecnologías de sistemas de gestión de *workflow*. En esta línea se elaboran dos trabajos de estudio de estas tecnologías, en el contexto de las asignaturas *Comercio Electrónico* y *Arquitecturas de Sistemas Distribuidos*, del programa de Doctorado en Tecnologías de las Comunicaciones. En el primero se analizan distintos formalismos de modelado, en especial redes de Petri de alto nivel de Aalst, diagramas de actividad de UML o los diagramas rol–actividad de Ould. En el segundo se plantea un estudio de las principales arquitecturas empleadas para desarrollar sistemas de gestión de *workflow*. Como resultado de este primer estudio, se plantea el modelado mediante *Streams* [81]. Sin embargo, se abandona pronto esta vía de trabajo. Al profundizar en su aplicación al modelado de procesos de negocio, se decide que el formalismo no resulta adecuado para ello.

Aunque ya estaba planteado como objetivo anteriormente, se comienza en septiembre de 2002 a centrar la tesis en la verificación de requisitos en procesos de negocio. Se comienza aplicando el método formal Z [120], como uno de los trabajos de doctorado del segundo curso. Para ello, se desarrolla una metodología de representación de procesos basada en dicho formalismo. Sin embargo, tras varios meses de trabajo, se decide que no resulta adecuado, dada la limitación actual de las herramientas existentes para trabajar con Z. Los motores de demostración de teoremas son bastante limitados y requieren un gran esfuerzo por parte del usuario. Se consigue demostrar propiedades triviales, pero no resulta útil para la demostración de propiedades más generales en los procesos. Finalmente, se abandona también esta línea de investigación.

Sin embargo, se conserva el modelo, basado en estados y transiciones, desarrollado mediante Z. Se formaliza este modelo independientemente de Z, dando lugar al formalismo que en esta tesis se conoce como *Modelo Formal Común* (CFM). Se estudian posibles alternativas para realizar las verificaciones. En concreto, se realiza un primer estudio modelando los procesos en un entorno CLIPS (*C Language Integrated Production System* [3]), una tecnología de sistemas expertos desarrollada a mediados de los 80 en *Johnson Space Center* de la NASA. Se desarrollan las verificaciones con uno de sus clones más potentes, el motor de inferencia JESS 6.0 [4]. Se propone una metodología para representar procesos de negocio, basados en el modelo desarrollado, mediante el lenguaje CLIPS.

En paralelo, se evalúa la aplicación de técnicas de verificación basadas en *model checking*, en otro de los trabajos de segundo curso del programa de doctorado. Se profundiza en

[3]http://www.ghg.net/clips/CLIPS.html

[4]http://herzberg.ca.sandia.gov/jess/

el empleo de verificadores a medida y herramientas de *model checking* existentes (en concreto, Spin). Por otra parte, se trabaja en una extensión del modelo básico para el modelado de procesos de negocio concurrentes que se comunican. Se desarrolla un primer prototipo de conversión de los modelos al lenguaje Promela.

En base a los resultados estos trabajos, se realiza una primera publicación: en [49] se presenta el modelo formal precursor de CFM y se plantean distintas alternativas para realizar la verificación de requisitos funcionales: motor de inferencia, *model checking* y algoritmos para un verificador a medida.

A partir del momento de redacción del artículo anterior, se comienza a asentar, formalizar y analizar en más detalle el trabajo realizado hasta entonces. En octubre de 2003 se escriben estos resultados y se publican en [50]. En paralelo con esta última publicación, se comienza a explorar la aplicación de estos resultados en otros dominios. En [51] se plantea el desarrollo de aplicaciones Web como procesos, y la verificación de requisitos funcionales en ellos.

En noviembre de 2003 comienza una nueva etapa en el desarrollo de la tesis, la transformación de procesos BPEL4WS al modelo CFM. En paralelo, se desarrolla en lenguaje Java un prototipo que realice esta transformación de forma automática: *verbus–j*. Por otra parte, se formaliza la arquitectura propuesta en esta tesis y se establece el nombre de CFM para su capa intermedia. Los resultados de esta etapa, orientados a entornos de comercio electrónico *business to business* colaborativos, se publican en [45].

Posteriormente, en julio de 2004, se trabaja en la integración del *model checker* SMV en el entorno. Se publican los resultados en [47] y [46]. La primera de estas publicaciones es seleccionada por el comité de programa del congreso y los editores de *IEEE América Latina* para ser publicada posteriormente en dicha revista [48].

Finalmente, a principios de 2005, se realiza un estudio más formal de la adecuación del modelo CFM para el modelado de procesos de negocio, aplicando *patrones de workflow*. Al mismo tiempo, se comienza la redacción de esta tesis.

1.5. Estructura

En este apartado se describe brevemente la organización del contenido de la tesis y su distribución en capítulos.

En el capítulo 2 se presenta un estudio del estado del arte en el ámbito de las tecnologías más relacionadas con esta tesis. En primer lugar, se estudian los métodos formales, profundizando en *model checking*, por ser la tecnología de verificación seleccionada para aplicar en esta tesis. A continuación, se estudia el campo de la gestión de procesos de negocio. Posteriormente, se analizan las tecnologías de composición y coordinación de servicios Web. Finalmente, se estudian los trabajos realizados hasta el momento en el ámbito de la verificación de procesos de negocio y composiciones y coordinaciones de servicios Web.

En el capítulo 3 se propone una arquitectura abierta, modular y extensible para la verificación de requisitos funcionales en procesos de negocio y composiciones de servicios Web. Con esta arquitectura se pretende solucionar los principales problemas detectados en los trabajos relacionados. Se toma como marco de referencia para el desarrollo del resto de la tesis.

En el capítulo 4 se define la capa intermedia de la arquitectura mediante un nuevo sistema formal: el *Modelo Formal Común*. Se definen formalmente el modelo y su notación. Posteriormente, se plantean unas líneas generales acerca de cómo representar procesos con este modelo. Finalmente, se analizan las principales limitaciones del modelo, y se justifica su importancia desde el punto de vista de la representación de procesos.

En el capítulo 5 se analiza el *Modelo Formal Común* aplicando *patrones de workflow*, con el objetivo de comprobar su grado de adecuación para formar parte de la capa intermedia de la arquitectura. Se comparan los resultados con los correspondientes a aplicar los mismos patrones a otros lenguajes y formalismos de modelado de procesos.

En el capítulo 6 se discute cómo se pueden verificar requisitos funcionales de procesos de negocio representados mediante el *Modelo Formal Común*. En primer lugar, se plantean los distintos tipos de requisitos a verificar. Para ello, se analizan trabajos de otros autores, tanto en el ámbito de la verificación de procesos como de otros tipos de sistemas. A continuación, se plantea cómo verificar estos tipos de requisitos en dos herramientas de *model checking* distintas: Spin y NuSMV.

En el capítulo 7 se analiza la integración en la arquitectura del lenguaje BPEL4WS. El objetivo de este capítulo es mostrar la viabilidad de la arquitectura y del modelo formal propuestos, además de servir de referencia para futuras integraciones de otros lenguajes de definición de procesos. Por una parte, se presenta una metodología de representación de procesos BPEL4WS mediante el *Modelo Formal Común*. Esta metodología permite realizar la transformación de definiciones de procesos entre ambos lenguajes de forma automática. Por otra, se propone cómo definir requisitos para procesos BPEL4WS y cómo representarlos mediante requisitos en el ámbito del *Modelo Formal Común*.

En el capítulo 8 se propone un caso de estudio para ilustrar los conceptos presentados anteriormente. Se propone un proceso BPEL4WS a modo de ejemplo, con un conjunto de requisitos a ser verificados. En primer lugar, se expone cómo se representan el proceso y sus requisitos mediante el *Modelo Formal Común*. A continuación se plantea cómo se transforman al lenguaje de entrada de Spin. Finalmente, se muestra cómo se realiza la verificación.

En el capítulo 9 se concluye presentando los principales resultados de este trabajo, y planteando posibles líneas de continuación del mismo.

Capítulo 2

Estado del arte

En este capítulo se realiza un estudio sobre el estado actual de las tecnologías más importantes relacionadas con las contribuciones propuestas en esta tesis doctoral. En el apartado 2.1 se presenta el estado del arte en el uso de métodos formales para la verificación de requisitos funcionales de sistemas. En el apartado 2.2 se presentan las tecnologías de *gestión de procesos de negocio*, centrando la descripción en dos aspectos concretos: definición y análisis de procesos de negocio. En el apartado 2.3 se expone el estado del arte en tecnologías de composición y coordinación de servicios Web, dada la estrecha relación existente entre estas tecnologías y las tecnologías de procesos de negocio. En el apartado 2.4 se analizan los principales trabajos relacionados con la verificación de requisitos funcionales de procesos de negocio y composiciones de servicios Web. En el apartado 2.5 se concluye con un análisis de las principales limitaciones encontradas en estos trabajos relacionados.

2.1. Métodos formales y *model checking*

En este apartado se presenta brevemente el estado del arte en la aplicación de métodos formales a sistemas *hardware* y *software*. En concreto, se centra en técnicas de *model checking*, dado que actualmente son las técnicas de verificación de requisitos funcionales más automatizables y potentes.

2.1.1. Introducción

Las ingenierías tradicionales están basadas en conocimientos científicos y matemáticos, que permiten conocer con una precisión elevada las características y propiedades de diseños y modelos. Sin embargo, la ingeniería del *software* apenas cuenta con este tipo de artilugios para el diseño y desarrollo de sistemas, lo cual provoca a menudo incumplimientos de plazos, existencia de demasiados errores en los sistemas desarrollados o estimaciones incorrectas de costes. Los métodos formales pretenden dotar a la ingeniería del *software* de estos artilugios matemáticos.

Se han planteado distintas definiciones de *métodos formales*. En [116] se definen como la utilización de modelado matemático, cálculo y predicción en la especificación, diseño, análisis, construcción y prueba de sistemas de computación y *software*. En [68] se define de forma más concreta un método formal como un conjunto de herramientas y notaciones

(con semánticas formales) utilizadas para especificar de forma no ambigua los requisitos de un sistema *software*, que permite la demostración de propiedades de dicha especificación, así como el desarrollo de demostraciones acerca de la corrección de una implementación con respecto a su especificación.

En [87] se describe brevemente la historia de los métodos formales. Comienza a finales de los años 60, cuando Floyd, Hoare y Naur proponen técnicas axiomáticas para comprobar que programas secuenciales son compatibles con su especificación formal [53, 69, 105]. A mediados de los 70 se comienza a trabajar en la especificación de programas concurrentes, y Pnueli aplica por primera vez lógicas temporales a este tipo de sistemas [112]. Las demostraciones de propiedades se hacían a mano, por lo que no resultaban demasiado prácticas, pero a principios de los 80 tanto Clarke y Emerson como Quielle y Sifakis proponen algoritmos de *model checking* para lógicas temporales CTL, que permitieron la automatización de estas demostraciones [31, 39, 113]. Desde entonces se han desarrollado numerosos y diversos métodos formales.

Desde el punto de vista de este trabajo, los dos aspectos más importantes en la aplicación de métodos formales al diseño de sistemas son la *especificación* y la *verificación de requisitos funcionales*. A continuación se desarrolla cada uno de ellos.

2.1.2. Especificación de sistemas

La especificación de sistemas es un aspecto fundamental de la aplicación de métodos formales. Consiste en realizar un modelo o abstracción del sistema que contenga sus propiedades más relevantes. El uso que se pretenda dar a dicha especificación determinará cuáles son las propiedades relevantes.

En [87] se clasifican los distintos paradigmas de especificación formal de sistemas *hardware* y *software* en los siguientes tipos:

- Basados en *historias*: se especifica un sistema caracterizando el conjunto máximo de historias (comportamientos) permitidos, utilizando propiedades expresadas mediante lógicas temporales.

- Basados en estados: se caracteriza el sistema especificando los estados admisibles (invariantes) y pre/post–condiciones de cada operación. Corresponden a este tipo, entre otros, Z [120], VDM [77] y B [11].

- Basados en transiciones: se especifican las transiciones entre estados. Algunos de los lenguajes y formalismos de este tipo son Statecharts [66], Promela [72, 73], STep-SPL [91], RSML [88] o SCR [109].

- Especificación funcional: se especifica el sistema como una colección estructurada de funciones matemáticas. Se pueden clasificar en este tipo lenguajes como OBJ [56], ASL [17], PLUSS [59], LARCH [64], HOL [61] o PVS [108].

- Especificación operacional: el sistema se caracteriza como un conjunto de procesos ejecutados por una máquina abstracta. Se pueden incluir en esta categoría Paisley, GIST, redes de Petri y álgebras de procesos [70, 98].

2.1.3. Verificación de requisitos funcionales

Una aplicación habitual de los métodos formales es la *verificación de requisitos funcionales*, esto es, la detección de errores en diseños e implementaciones de sistemas *hardware* y *software*, que provoquen que la funcionalidad de los mismos no sea la deseada por sus diseñadores. En el resto de este trabajo se denominará este concepto de forma abreviada como *verificación*. De acuerdo con [32], los principales métodos de verificación aplicables a este tipo de sistemas son simulación guiada, pruebas, verificación deductiva y *model checking*.

Simulación guiada y pruebas consisten en realizar experimentos con los sistemas antes de su implantación. La simulación guiada se realiza sobre una abstracción o modelo del sistema, mientras que las pruebas se realizan sobre el propio sistema desarrollado. Ambas técnicas permiten detectar errores en los sistemas, pero no permiten comprobar todas las situaciones posibles.

La verificación deductiva consiste en utilizar axiomas y reglas para demostrar matemáticamente la corrección de los sistemas. Esta técnica permite realizar razonamientos incluso sobre sistemas con un número infinito de estados, pero en general las demostraciones requieren mucho tiempo y la participación de un experto en matemáticas y lógica. Por otra parte, es fácil que se cometan errores en las demostraciones, dada su complejidad.

El concepto de *model checking* engloba a un conjunto de algoritmos eficientes que permiten verificar propiedades de modelos concurrentes con un número finito de estados, normalmente expresadas mediante lógicas temporales. Se desarrollan en profundidad los conceptos de lógicas temporales y *model checking* en los próximos apartados.

2.1.4. Lógicas temporales

Las lógicas temporales son el formalismo más frecuentemente utilizado para expresar las propiedades a verificar en modelos basados en estados y transiciones. La lógica de proposiciones permite expresar propiedades relativas a estados del modelo, mediante proposiciones atómicas y conectores lógicos (negación, conjunción, disyunción, condicional y bicondicional). Esta lógica resulta suficiente para verificar programas desde el punto de vista de su entrada y salida. Sin embargo, en sistemas reactivos o concurrentes resulta relevante también poder describir las transiciones entre estados, esto es, detalles internos acerca de cómo evoluciona la computación. Esto es especialmente importante en sistemas de ejecución continua, dado que nunca finalizan.

Las *lógicas temporales* son formalismos que permiten describir secuencias de transiciones entre estados, complementando la lógica de proposiciones con operadores temporales. De esta forma se pueden establecer relaciones de orden entre eventos sin necesidad de introducir explícitamente el concepto de *tiempo*. Estas lógicas fueron desarrolladas inicialmente por filósofos, y Pnueli [112] fue el primero en aplicarlas para la verificación de sistemas concurrentes. A partir de entonces, las lógicas temporales se han convertido en un formalismo comúnmente utilizado para enunciar propiedades en modelos basados en estados y transiciones.

Existen distintas lógicas temporales, que fundamentalmente difieren en los operadores temporales aplicables, de entre las cuales destacan CTL* y sus sublógicas CTL y LTL, así como el cálculo proposicional μ*-calculus*.

Lógica temporal CTL*

CTL* (*Computation Tree Logic*) describe propiedades relativas a árboles de computación. Un árbol de computación representa todas las posibles secuencias de estados que se pueden producir en el sistema. El nodo raíz del árbol representa el estado inicial del sistema. Los nodos hijo de un determinado nodo del árbol representan todos los estados alcanzables desde este a través de una transición. Esta estructura podría tener incluso tamaño infinito. Las fórmulas CTL* se construyen mediante lógica de proposiciones, cuantificadores de camino y operadores temporales.

Los cuantificadores de camino describen bifurcaciones en el árbol, esto es, propiedades que deben ser ciertas para todos o alguno de los caminos partiendo de un estado dado. Son cuantificadores de camino los siguientes:

- Operador **A**: la fórmula $\mathbf{A}f$, aplicada a un estado, indica que la fórmula f debe ser cierta para todos los caminos de computación que partan de dicho estado.
- Operador **E**: la fórmula $\mathbf{E}f$, aplicada a un estado, indica que la fórmula f debe ser cierta para al menos un camino de computación que parta de dicho estado.

Los operadores temporales describen propiedades que deben ser ciertas para un determinado camino de computación. Son operadores temporales los siguientes:

- Operador **X**: la fórmula $\mathbf{X}f$, aplicada a un camino de computación, indica que la fórmula f debe ser cierta en el siguiente estado de dicho camino.
- Operador **F**: la fórmula $\mathbf{F}f$, aplicada a un camino de computación, indica que debe existir al menos un estado en dicho camino para el cual la fórmula f sea cierta.
- Operador **G**: la fórmula $\mathbf{G}f$, aplicada a un camino de computación, indica que la fórmula f debe ser cierta en todos los estados del camino.
- Operador **U**: la fórmula $f\mathbf{U}g$, aplicada a un camino de computación, indica que existe al menos un estado del camino en el cual la fórmula g es cierta, y en todos los estados anteriores a este la fórmula f es cierta.
- Operador **R**: la fórmula $f\mathbf{R}g$, aplicada a un camino de computación, indica que la fórmula g es cierta para todos los estados del camino hasta, inclusive, el primero para el cual la fórmula f sea cierta. No exige que exista algún estado en el camino en el cual la fórmula f sea cierta.

Por ejemplo, la fórmula $s \models \mathbf{EF}(\mathbf{AG}f)$, siendo s un estado y f una proposición atómica, significa que existe algún camino de computación partiendo de s, en el cual existe algún estado s' tal que la propiedad f es cierta en todos los estados de todos los caminos de computación que parten de s'.

Las lógicas temporales CTL y LTL son sublógicas de CTL*, construidas estableciendo determinadas restricciones a la combinación de operadores para la construcción de fórmulas. LTL sólo permite fórmulas del tipo $\mathbf{A}f$ donde f es una fórmula de camino que no puede contener cuantificadores de camino. CTL exige que los operadores temporales deben estar siempre precedidos de un cuantificador de camino.

Las lógicas temporales ACTL* y ACTL son sublógicas de CTL* y CTL, que sólo permiten utilizar cuantificadores de camino **A**. Como consecuencia, sólo se pueden aplicar negaciones lógicas a fórmulas atómicas, dado que una negación aplicada a un cuantificador **A** puede dar lugar implícitamente a un cuantificador **E**. Estas restricciones permiten optimizar los algoritmos de verificación con técnicas como la *abstracción* y el *razonamiento por composición*.

En [32] se explican en mayor profundidad estas lógicas temporales y se presentan descripciones precisas de su sintaxis y semántica.

Cálculo proposicional μ-calculus

El cálculo proposicional μ-calculus es un lenguaje potente que permite expresar propiedades de sistemas de transiciones utilizando los operadores de punto fijo mínimo y máximo. Su importancia para verificación de modelos se debe a que muchas lógicas temporales se pueden expresar utilizando este lenguaje, a la existencia de algoritmos de *model checking* eficientes para μ-calculus, y a la importancia de los algoritmos de punto fijo en *model checking* simbólico. Se han desarrollado varias versiones de μ-calculus, como por ejemplo la versión de Kozen [83].

2.1.5. Model checking

El concepto de *model checking* [32] engloba a un conjunto de técnicas de verificación de sistemas concurrentes con número finito de estados, basadas realizar una exploración exhaustiva del espacio de estados de los sistemas. Las ventajas de *model checking* con respecto a otras técnicas de verificación de requisitos funcionales son dos, principalmente. Por una parte, la verificación es totalmente automática y no requiere ser un experto en matemáticas para realizarla. Por otra parte, cuando se detecta que una determinada propiedad es falsa, el sistema proporciona un contraejemplo que demuestra su falsedad, el cual es una inestimable ayuda para encontrar y comprender las causas del error.

El principal problema del *model checking* está en que realiza una exploración exhaustiva del espacio de estados, y por tanto su aplicabilidad está condicionada por el número de estados alcanzables por el sistema, que crece exponencialmente con el número de variables con que se modele el mismo. Se conoce a este problema como *explosión de estados*. Con el objetivo de minimizar el efecto de la explosión de estados, se han desarrollado distintos algoritmos de *model checking* en las dos últimas décadas, cada vez más eficientes, así como técnicas de simplificación que transforman modelos en otros modelos equivalentes desde el punto de vista de las propiedades a verificar, pero con un tamaño del espacio de estados considerablemente menor.

Dado un modelo de un sistema, expresado como un sistema de estados y transiciones, y una especificación, que contiene propiedades que deberían ser ciertas en el modelo, el problema de *model checking* consiste en comprobar si todas las propiedades de la especificación son ciertas en dicho modelo. Las propiedades de la especificación se suelen expresar utilizando lógicas temporales.

Un problema equivalente al anterior es el de *comprobación de equivalencia*, que consiste en comprobar si dos modelos se comportan de la misma forma. Esto es útil, por ejemplo,

para comprobar que el modelo y su especificación se comportan de la misma forma. También es útil cuando se optimiza el diseño de un sistema, ya que permite comprobar que el sistema original y el optimizado son equivalentes desde un punto de vista funcional. Una posible forma de resolver el problema consiste en construir un modelo como la composición en paralelo de los modelos de los cuales se pretende hacer la comprobación de equivalencia, estableciendo como especificación una propiedad que asegure que la salida de ambos submodelos sea siempre igual.

Inicialmente la verificación de propiedades expresadas mediante lógicas temporales se realizaba a mano. A principios de los 80, Clarke y Emerson proponen un algoritmo de *model checking* para la verificación de fórmulas CTL, implementable de forma razonablemente eficiente. Demostraron que su complejidad era polinómica con respecto al tamaño del modelo y al tamaño de la fórmula CTL [31, 39]. De forma simultánea Quielle y Sifakis [113] proponen otro algoritmo de *model checking* aplicable a un subconjunto de CTL.

Estructuras de Kripke

Los principales algoritmos de *model checking* se basan en el modelado de los sistemas concurrentes como *grafos estado-transición etiquetados*, también conocidos como *estructuras de Kripke*. Una estructura de Kripke se denota como $M = (S, S_0, R, L)$, donde S es el conjunto de todos los posibles estados del sistema, S_0 el conjunto de estados iniciales, R una relación tal que $R(s, s')$ es *cierto* si y sólo si existe una transición desde el estado s hacia el estado s', y L es una función tal que $L(s)$ es el conjunto de todas las proposiciones atómicas ciertas en el estado s.

2.1.6. Técnicas avanzadas de *model checking*

Los algoritmos originales de *model checking* trabajaban de forma explícita con todas las transiciones y estados de la estructura de Kripke. Esto es, construían un grafo representando explícitamente todos los estados del sistema. Dado que el número de estados de un sistema crece exponencialmente con el número de variables, estos algoritmos podían ser aplicados a modelos pequeños, pero requerían un tamaño de memoria excesivamente grande para sistemas concurrentes con un mayor número de estados.

En este apartado se presentan distintas técnicas que se aplican actualmente para resolver el problema de *model checking* de forma más eficiente.

Algunas de estas técnicas consisten en nuevos algoritmos de *model checking*. Es el caso de *model checking simbólico* y *model checking acotado*. Otras consisten en algoritmos que generan modelos de tamaño considerablemente menor que los modelos originales. De esta forma reducen el espacio de búsqueda y, por tanto, mejoran el rendimiento de los algoritmos de *model checking*. Es el caso de técnicas como la *reducción parcial de orden*, *abstracción*, *simetría* y *verificación modular*, entre otros. Otras técnicas se orientan a utilizar entornos de computación distribuidos o multiprocesador para acelerar la verificación (*model checking distribuido*).

Las técnicas descritas en este apartado se exponen en profundidad, incluyendo descripción de algoritmos y su justificación matemática, en [32].

Model checking simbólico

El desarrollo en 1987 de *model checking simbólico* por McMillan [25, 93] supuso una revolución en la eficiencia de los algoritmos de *model checking*, permitiendo la verificación de modelos mayores en varios órdenes de magnitud. La principal característica de estas técnicas es que no representan explícitamente cada estado alcanzable por el modelo, sino mediante fórmulas booleanas que implícitamente definen un conjunto potencialmente muy grande de estados.

Se ha comprobado que una estructura de datos muy eficiente para representar y operar con este tipo de fórmulas booleanas es el *diagrama ordenado de decisión binaria* (*ordered binary decision diagram*, OBDD), que representa cada fórmula mediante un árbol. Bryant [23] desarrolló esta estructura en los años 80, aunque fue McMillan quien la aplicó a *model checking*. En *model checking* simbólico se representan mediante OBDDs tanto las transiciones del modelo como los conjuntos de estados sobre los que operan los algoritmos de punto fijo durante el proceso de verificación.

Los algoritmos de *model checking* basados en OBDDs permiten verificar de forma razonable modelos con cientos de variables y, a lo sumo, unos 10^{20} estados. Su mayor limitación está en el tamaño en memoria de los OBDDs necesarios para llevar a cabo la verificación, que pueden alcanzar fácilmente cifras del orden de *gigabytes*. Por otra parte, este tamaño es muy dependiente de la ordenación de las variables del modelo, pero el problema de averiguar el orden óptimo que minimiza el tamaño tiene una complejidad que impide que su resolución resulte rentable. Sin embargo, sí se han desarrollado posteriormente algoritmos de agrupación de variables, así como algoritmos de reordenación dinámica de variables durante la construcción de OBDDs que, aunque en general no encuentran el orden óptimo, son capaces de aproximarse razonablemente a este.

En la misma época se desarrollaron otras técnicas de *model checking* simbólico, como las mencionadas en [24].

Procedimientos de decisión procedural (SAT): *model checking* acotado

El *problema de satisfacción binaria* (*boolean satisfiability problem*, SAT) consiste en, dada una expresión *booleana* con k variables, averiguar si existe alguna combinación de valores de sus variables para la cual la expresión se evalúe con valor *cierto* [33]. Se ha demostrado que este problema (k-SAT con $k \geq 3$) es *NP-completo* si no se permite el uso de cuantificadores ($\forall$, $\exists$), y que cualquier problema *NP-completo* se puede reducir a un problema SAT en tiempo polinómico.

Tal y como se expone en [24], se han desarrollado distintos algoritmos de búsqueda y estructuras de representación de expresiones *booleanas* para resolver este problema, cada vez con un nivel de eficiencia mayor, y se han aplicado a numerosos problemas en distintas áreas de investigación, incluida la verificación de requisitos funcionales de modelos.

Desde este punto de vista, resultan especialmente destacables las técnicas de *model checking acotado* (*bounded model checking*), que vieron la luz en 1999 de la mano de Biere, Cimatti, Clarke y Zhu [19, 30], con la intención de proponer una alternativa al uso de OBDDs y aumentar así el tamaño de los modelos verificables. Actualmente suponen una línea muy activa de investigación en *model checking*.

Las técnicas de *model checking* acotado permiten resolver problemas de *model checking*

estableciendo una longitud máxima en los caminos a considerar en la verificación. Para cada longitud k de caminos menor o igual que dicha longitud máxima, se genera un fórmula proposicional tal que es cierta si y sólo si existe un contraejemplo de longitud k para la especificación. Mediante algoritmos SAT se buscan combinaciones de valores que hagan que dicha fórmula se evalúe como *cierta*. Si existe alguna, entonces se concluye que la especificación es *falsa* para dicho modelo. Aunque no permiten garantizar la veracidad de una especificación en caminos mayores que la cota, estas técnicas resultan muy útiles, dado que son capaces de generar contraejemplos mucho más rápidamente que los algoritmos basados en OBDDs, y permiten en muchos casos trabajar con modelos de mayor tamaño que dichos algoritmos. En [19] se presenta una comparativa de eficiencia entre ambas técnicas.

Reducción parcial de orden

La *reducción parcial de orden* (*partial order reduction*) es una técnica para reducir el número de estados del espacio de búsqueda de los algoritmos de *model checking*. Esta técnica aprovecha el hecho de que, en modelos con alto grado de concurrencia, existen normalmente transiciones que, independientemente del orden relativo en que sean ejecutadas, conducen al mismo resultado.

Por ejemplo, supongamos que, partiendo de un determinado estado s_0, se recorre el camino $< s_0, s_1, s_2 >$ como consecuencia de ejecutar en orden las transiciones t_1 y t_2. Supongamos también que si se ejecutan las mismas transiciones en orden contrario el camino es $< s_0, s_1', s_2 >$. Se observa que el estado final es independiente del orden de las transiciones y que, por tanto, se puede reducir el espacio de búsqueda si el algoritmo de *model checking* sólo tiene en cuenta una de las dos alternativas.

Explicándolo de otra forma, esta técnica permite crear, a partir del grafo completo del modelo, un grafo reducido tal que todos los comportamientos del grafo completo estén representados en el grafo reducido por un comportamiento equivalente. También se puede denominar a esta técnica *model checking con representantes*, dado que cada comportamiento en el grafo reducido representa a uno o más comportamientos equivalentes en el grafo completo. Forma parte del propio algoritmo determinar qué dependencias existen entre las distintas transiciones del modelo, dado que estas determinarán qué comportamientos son o no equivalentes. La reducción del espacio de búsqueda gracias al empleo de esta técnica puede alcanzar varios órdenes de magnitud, dependiendo del modelo. En [32] se presentan los resultados de una comparación experimental, en que se aprecia para un modelo concreto una reducción de 522255 a 8475 estados. También se observan en esos experimentos reducciones considerables en el tiempo de computación y cantidad de memoria utilizada. En [73] se explica cómo se ha implementado esta técnica en el *model checker* Spin.

Abstracción

La abstracción es considerada en [32] la técnica más importante de reducción del espacio de estados. Esta técnica se aplica antes de la construcción del propio modelo del sistema, y consiste en generar un modelo abstracto más sencillo que el original.

Dado un modelo M, se genera un modelo abstracto M_h que contenga, al menos, todos los posibles comportamientos del modelo original, pero que sea más sencillo. Para ello, se crea M_h sobre un conjunto de estados abstractos, donde cada estado abstracto representa

a uno o más estados de M. Toda transición entre dos estados de M se representa en M_h como una transición entre sus correspondientes estados abstractos. En consecuencia, M_h contendrá todos los comportamientos de M, pero puede también contener comportamientos no presentes en M. A pesar de ello, se puede demostrar que toda fórmula ACTL, ACTL* o LTL que sea cierta en M_h lo será también en M.

Sin embargo, del hecho de que una fórmula no sea cierta en M_h no se puede concluir que tampoco lo sea en M. Dado un contraejemplo a una fórmula en M_h, es necesario comprobar si dicho contraejemplo es posible en M. Si lo es, se concluye que la fórmula es también falsa en M. Si no, habría que *refinar* el modelo abstracto para excluir el comportamiento dado por el contraejemplo. Este paradigma de verificación en modelos abstractos se conoce como *refinado de abstracción basado en contraejemplos (CounterExample–Guided Abstraction–Refinement, CEGAR)*. En [27] se analiza en detalle la interpretación de contraejemplos en modelos abstractos.

Algunas de las técnicas de abstracción más utilizadas son el *cono de reducción de influencia*, la *abstracción de datos* y la *abstracción de predicados*.

La técnica del *cono de reducción de influencia* reduce el tamaño del grafo de transiciones eliminando aquellas variables que no formen parte de la especificación ni influyan en ninguna de las variables presentes en la especificación. Con esta técnica se verifican las mismas propiedades, pero en modelos cuyo tamaño puede ser considerablemente menor.

La técnica de *abstracción de datos* consiste en mapear el conjunto de posibles valores de las variables del sistema a un conjunto de valores abstractos más pequeño. De esta forma se crea un modelo abstracto que simula al modelo original, pero normalmente más pequeño. La verificación se realiza sobre el modelo abstracto. Por ejemplo, supongamos un modelo que declara una variable con valores enteros en el intervalo $[-128, 127]$. Supongamos también que sólo afecta al comportamiento del sistema el hecho de que su valor sea cero, mayor que cero o menor que cero. En este caso, se puede abstraer su rango de valores a un conjunto de sólo tres elementos: $\{v_-, 0, v_+\}$.

La técnica de *abstracción de predicados* consiste en agrupar estados concretos en estados abstractos de acuerdo al conjunto de predicados que satisfagan.

En [57] se realizan contribuciones en la aplicación de técnicas de abstracción en el *model checker* Spin.

Simetría

Numerosos tipos de sistemas concurrentes de estados finitos presentan relaciones de simetría. Las técnicas de *simetría* permiten construir modelos reducidos aprovechando la existencia de simetrías en los modelos originales, de tal forma que la verificación en el modelo reducido sea equivalente a la verificación en el modelo original. En [40] se profundiza en esta técnica.

Verificación modular

Una de las fuentes de complejidad en los modelos es la composición de distintos módulos, dado que el espacio de estados total del modelo es el producto del espacio de estados de cada uno de los módulos. La verificación modular consiste en verificar cada módulo por separado, en combinación con un modelo abstracto del resto de los módulos, que represente

únicamente el comportamiento visible externamente de los mismos desde el punto de vista del módulo a verificar. Dado que el espacio de estados del modelo abstracto de cada módulo será, en general, considerablemente menor, se reduce el espacio de estados a verificar.

Model checking distribuido

Esta técnica consiste en paralelizar algoritmos de *model checking*, con el objetivo de aprovechar los recursos de máquinas multiprocesador y entornos de computación distribuidos.

2.1.7. Herramientas de *model checking*

Actualmente existen múltiples herramientas de *model checking*, tanto comerciales como de dominio público. En el ámbito académico, destacan principalmente dos: SMV (y sus variantes) y Spin.

Spin[1] [72, 73] es un *model checker* LTL que utiliza enumeración explícita de estados y reducción parcial de orden. Holzmann y Peled iniciaron su desarrollo en los años 80 en Bell Labs. Desde 1991 se encuentra públicamente disponible, incluido su código fuente. Los modelos y especificaciones se definen mediante el lenguaje Promela, desarrollado por Holzmann. Una de las principales optimizaciones de Spin consiste en generar dinámicamente, a partir del modelo y especificación, un programa C que contiene el algoritmo de verificación particularizado para el propio modelo.

SMV[2] (*Symbolic Model Verifier*) [93] es una herramienta de *model checking* desarrollada en *Carnegie Mellon University*, capaz de verificar especificaciones CTL en modelos de sistemas concurrentes de estados finitos. Utiliza el algoritmo de *model checking* simbólico con OBDDs desarrollado originalmente por McMillan. Los modelos y especificaciones se definen mediante un lenguaje específico desarrollado también por McMillan [93]. La herramienta se encuentra disponible públicamente, incluyendo su código fuente. McMillan desarrolló versiones comerciales posteriores de este código conocidas como *Cadence SMV*, incluyendo optimizaciones con respecto a la versión inicial de SMV.

NuSMV[3] [29] es una reimplementación y extensión de SMV desarrollada en colaboración por varios centros de investigación italianos y *Carnegie Mellon University*. Su versión actual se distribuye con licencia de código libre (LGPL). Esta herramienta integra algoritmos de verificación basados en OBDDs y algoritmos SAT.

2.2. Gestión de procesos de negocio

La gestión de procesos de negocio comprende un conjunto de tecnologías orientadas a dar soporte a la ejecución de la lógica de negocio de organizaciones. Por ejemplo, la recepción de un pedido por parte de un distribuidor de libros implica la recogida de los datos del cliente y del pedido, comprobación de existencias en almacén, pedido a otro distribuidor o editorial si fuese necesario, o incluso su importación desde el extranjero, cobro del pedido

[1] http://spinroot.com
[2] http://www-2.cs.cmu.edu/ modelcheck/smv.html
[3] http://nusmv.irst.itc.it/

proceso de negocio	Un conjunto de uno o más procedimientos o actividades que colectivamente realizan un objetivo de negocio o una meta, normalmente en el contexto de una estructura organizativa que define unos papeles funcionales y relaciones.
workflow	La automatización de un proceso de negocio, completo o en parte, durante la cual se transfieren documentos, información y tareas de un participante a otro para que actúen sobre ellos, de acuerdo a un conjunto de reglas procedurales.
sistema de gestión de *workflow*	Un sistema que define, crea y gestiona la ejecución de *workflows* mediante la utilización de *software*, ejecutado en uno o más motores de *workflow*, que es capaz de interpretar la definición del proceso, interactuar con los participantes del *workflow* y, cuando es necesario, invocar a otros sistemas de información y aplicaciones.
definición de un proceso	Representación de un proceso de negocio de forma que soporte manipulación automática, como modelado o ejecución por un sistema de gestión de *workflow*. La definición de un proceso consiste en una red de actividades y sus relaciones, criterios para indicar el inicio y finalización del proceso, e información acerca de las actividades individuales, tales como sus participantes, aplicaciones y datos asociados, etc.
gestión de procesos de negocio	Técnicas de soporte para procesos de negocio mediante la utilización de métodos, técnicas y *software* para diseñar, llevar a cabo, controlar y analizar procesos operativos involucrando a personas, organizaciones, aplicaciones, documentos y otras fuentes de información.

Cuadro 2.1: Definiciones de los principales conceptos relacionados con procesos de negocio. Los cuatro primeros se definen en un estándar del WfMC [126]. El último, en [10].

y envío del mismo al cliente. Estas tareas deben coordinarse de acuerdo a una determinada lógica, que se representa como un *proceso de negocio*.

En el cuadro 2.1 se muestran definiciones de los principales conceptos relacionados con el ámbito de la gestión de procesos de negocio.

Los procesos de negocio definen la forma en que las organizaciones llevan a cabo sus propósitos. Fundamentalmente, un proceso de negocio se caracteriza por: contener actividades que tienen un propósito común; ser llevado a cabo colaborativamente por un grupo; sobrepasar, a menudo, las fronteras de una organización; ser iniciado como reacción a eventos originados en el exterior de la organización. En una organización se puede identificar tres tipos básicos de procesos de negocio [107]: procesos centrales, procesos de soporte y procesos de gestión. Los primeros son los que directamente intentan aportar valor a los clientes. Los segundos dan soporte a las necesidades de entidades internas a la organización para, indirectamente, aportar valor a los clientes. Los últimos gestionan los tipos de procesos anteriores y se encargan de la planificación al nivel de negocio.

Un *workflow* es un proceso de negocio operativo, esto es, un proceso de negocio que puede ser ejecutado automáticamente. Un sistema de gestión de *workflow* (WFMS) es el sistema de *software* que proporciona el soporte necesario para ejecutar y controlar *workflows*. El concepto de gestión de *workflow* (WFM) está, como se puede apreciar en las definiciones, muy centrado en la ejecución de los procesos. En los últimos tres o cuatro años se ha encontrado esta aproximación demasiado restrictiva, y se acuñó un nuevo término: *Gestión de Procesos de Negocio* (*Business Process Management*, BPM). BPM es un concepto más amplio que WFM, y su principal diferencia, según se recoge en [10], está en que hace especial énfasis en las fases de diagnóstico de procesos (análisis de procesos para identificar problemas o aspectos que pueden ser mejorados) y definición de procesos (diseño o rediseño

de procesos), aspectos tradicionalmente desatendidos en WFM.

Dado que esta tesis se centra en las tecnologías para la definición y análisis de procesos de negocio, en los siguientes apartados se describen estas facetas.

2.2.1. Definición de procesos de negocio

En el cuadro 2.1 se muestra una definición formal del concepto de *definición de un proceso de negocio*. Una definición de un proceso es, esencialmente, un modelo del mismo inteligible y procesable por aplicaciones *software*. Estos modelos se representan mediante *lenguajes de definición de procesos de negocio*. Desde principios de los noventa se han definido y utilizado una gran cantidad de lenguajes de definición de procesos de negocio. Aunque ha habido intentos para establecer algunos lenguajes como estándar, ninguno de ellos ha cuajado por el momento.

Dependiendo del uso que se le pretenda dar a la definición de un proceso, el modelo debe recoger distintas características del mismo. Esto influye en gran manera en la elección de una metodología de modelado y lenguaje de definición. En [107] se clasifican en tres categorías los motivos por los cuales se modela un proceso de negocio:

- Para describirlo: definirlo, comunicarlo, negociarlo, etc. Por ejemplo, para comunicar a un empleado cómo debe realizar su trabajo, y cómo este se relaciona con el trabajo de sus compañeros y los objetivos de la empresa.
- Para analizarlo: implica un análisis, cuantitativo o cualitativo, de un proceso para, por ejemplo, decidir cambios en la planificación de actividades, cambios de asignación de tareas a empleados, incremento o reducción del grado de paralelismo, etc.
- Para ejecutarlo: a partir de un modelo (o especificación) formal del proceso, conseguir que un sistema *software* (por ejemplo, un WFMS) coordine su ejecución, esto es, gestione automáticamente las tareas necesarias para que los procesos puedan ser llevados a cabo.

Requisitos de un lenguaje de modelado

A pesar de que los requisitos exigibles a una técnica de modelado de procesos depende de la utilización que se pretenda dar a las especificaciones, en [107] se recogen una serie de requisitos deseables en la mayoría de los casos:

- Debe permitir el modelado de situaciones complejas. Los procesos de negocio no se pueden modelar, en general, sólo con estructuras jerárquicas (árboles).
- Los modelos no pueden ser ambiguos. Para ello, resulta conveniente que posean una semántica y sintaxis formalmente definidas.
- La notación empleada debe ser fácilmente asimilable por personas. Desde este punto de vista, los lenguajes gráficos resultan más adecuados.
- Debe permitir capturar tanto los aspectos funcionales del proceso como los no funcionales.

- Debe permitir la incorporación fácil de los patrones que más frecuentemente aparecen en los procesos, tales como planificación, delegación de responsabilidades, contratos, actividades periódicas, etc. Por ello, debe también permitir la reutilización de modelos o partes de modelos.

- Se debe centrar en el proceso, no en los datos.

- Debe proporcionar la posibilidad de ser visualizado desde distintos puntos de vista o niveles de abstracción.

Metodologías de modelado

Cada lenguaje de definición de procesos está basado en un modelo. Existen, principalmente, dos metodologías de modelado de procesos [60]: basadas en comunicación y basadas en acción.

Las *metodologías basadas en comunicación* surgen del trabajo de Winograd y Flores "Conversation for Action Model" [124]. Estas metodologías asumen que el objetivo de la reingeniería de un proceso es incrementar la satisfacción del cliente. Representa cada acción como un ciclo con cuatro fases de comunicación entre un cliente y un intérprete: preparación, negociación, realización y aceptación. En la fase de preparación el cliente solicita la realización de una acción, o el intérprete se ofrece para realizar una acción. En la fase de negociación ambos acuerdan los términos relativos a la acción y definen los términos de satisfacción. En la fase de realización se realiza la acción de acuerdo a los términos anteriores. En la fase de aceptación el cliente expresa su satisfacción (o insatisfacción) con la acción realizada. Cada una de las fases de un ciclo se puede unir a otros ciclos, de tal forma que un intérprete de un ciclo puede actuar como cliente de otros. De esta forma se puede ver un proceso de negocio como una red social en la que un grupo de personas, realizando distintos papeles, alcanzan el objetivo del proceso. En [60] se identifican las principales limitaciones de estas metodologías. Por una parte, dado que están totalmente centradas en la satisfacción del cliente, no resultan apropiadas para situaciones cuyo objetivo de modelado sea otro, como, por ejemplo, minimizar el gasto de material de un proceso de producción. Por otra parte, las definiciones de procesos no permiten la ejecución automática en WFMSs.

Las *metodologías basadas en actividad* se centran en el modelado del trabajo. En lugar de modelar compromisos entre personas, se centran en modelar el conjunto de tareas necesarios para llevar a cabo el proceso. Una gran mayoría de los modelos utilizados tanto comercialmente como a nivel de investigación siguen estas metodologías. En el cuadro 2.2 se muestran los conceptos más relevantes utilizados habitualmente para el modelado basado en actividad.

Perspectivas de un proceso de negocio

Habitualmente, un proceso de negocio está compuesto por la unión de aspectos de distintos ámbitos o perspectivas. Es común identificar las siguientes cinco perspectivas [75, 121]: control de flujo (o proceso), recursos (u organización), datos (o información), actividades (o tareas o funciones) y operación (o aplicación):

caso	Por cada cliente que realiza una petición, se crea una nueva instancia del proceso para atender a dicho cliente. Cada una de estas instancias recibe del nombre de *caso*.
tarea	Representa una unidad de trabajo. Se describe como una ordenación total o parcial de operaciones, descripciones de actividades humanas y otras tareas. También a menudo se emplea el término *actividad* como sinónimo de tarea.
actividad	Realización de una tarea para un caso concreto. A menudo se utiliza el concepto de actividad como sinónimo de *tarea*.
actor	Entidad que realiza tareas, e interactúa con otros actores durante la ejecución del proceso. Los actores pueden ser personas o no serlo (máquinas, programas de *software*).
papel (*rol*)	Conjunto de actores cuyas características los capacita para realizar una determinada tarea o conjunto de tareas. Cada actor está adscrito a uno o más roles, y cada tarea especifica a qué rol debe pertenecer el actor que la realiza.
objeto manipulado	Documento, datos, objeto material, etc. sobre el cual se actúa en las tareas, y que intercambian los actores durante la ejecución de un proceso. También se le conoce como *entidad*.
interacción	Modela cómo las distintas tareas interactúan entre sí (ordenación en la ejecución de tareas, sincronización, intercambio de objetos, etc.)
evento	Suceso, normalmente externo al proceso, que puede forzar en éste un cambio de estado (por ejemplo, la llegada de un mensaje, el vencimiento de una temporización o una determinada acción de un actor).

Cuadro 2.2: Definiciones de los principales conceptos aplicados en metodologías de modelado de procesos basadas en actividad. Se puede encontrar una definición formal de los mismos en [126].

- En la *perspectiva de control de flujo* se define la coordinación de actividades, esto es, qué actividades deben ser ejecutadas y en qué orden. Se suelen utilizar primitivas de ejecución secuencial o en paralelo, bifurcación condicional, unión de ramas alternativas, iteración y sincronización, entre otras.

- En la *perspectiva de recursos* se definen los actores (máquinas o humanos) necesarios para la ejecución del proceso. Los actores humanos, también llamados participantes, suelen estar agrupados según su papel o papeles en la organización.

- En la *perspectiva de datos* se gestionan datos de control y de producción. Los datos de control se introducen para controlar la ejecución del proceso (por ejemplo, para tomar decisiones en puntos de bifurcación condicional). Los datos de producción se corresponden con las entidades del proceso (por ejemplo, documentos, formularios, tablas, etc.)

- En la *perspectiva de actividades* se describen las características de cada actividad del proceso (descripción, duración estimada, plazos, prioridad, clases de actores necesarias, etc.)

- Cada actividad puede estar compuesta por más de una acción elemental. En la *perspectiva de operación* se describen las acciones elementales, que normalmente consisten en la ejecución de una aplicación con unos determinados parámetros.

2.2.2. Lenguajes de definición de procesos de negocio

Desde los inicios de la tecnología de *workflow* y procesos de negocio, se han desarrollado una gran cantidad de lenguajes de definición de procesos y, a pesar de algunos intentos, no se ha conseguido establecer ningún lenguaje como estándar comúnmente aceptado. En este apartado se describen brevemente algunos de los más utilizados.

Desde el punto de vista del análisis de procesos de negocio, es importante que el modelo utilizado dé lugar a definiciones de procesos no ambiguas. Para ello, muchos de estos modelos se construyen sobre una semántica formal, que relaciona una definición de un proceso con una estructura matemática. Por tanto, resultan especialmente relevantes para esta tesis los lenguajes de definición que cuentan con semánticas formales.

Modelos basados en redes de Petri

Las redes de Petri, y algunas de sus extensiones, son formalismos muy utilizados para el modelado de procesos de negocio. En [111, 101, 115] se describe y analiza en detalle este formalismo, sus aplicaciones y distintas variantes.

Algunas de las variantes de redes de Petri utilizadas para el modelado de procesos de negocio son: *Workflow Nets* [2], *Information Control Nets* [38], INCOME/WF [106], *FunSoft Nets* [34] y MILANO WFMS [12]. En [117] se presentan con detalle otras aplicaciones de redes de Petri para el modelado de procesos de negocio.

En [2] se justifica la utilización de redes de Petri para modelado de procesos de negocio:

- Semántica formal: las definiciones de procesos son claras y precisas porque tanto las redes de Petri como sus principales extensiones están matemáticamente formalizadas.
- Representación gráfica: son intuitivas y fáciles de leer.
- Expresividad: permite modelar las primitivas más utilizas en procesos de negocio.
- Propiedades bien conocidas: se han estudiado en profundidad sus propiedades.
- Análisis: se han desarrollado una gran cantidad de algoritmos de análisis para redes de Petri, que permiten comprobar propiedades de viveza, seguridad, medidas de prestaciones, etc.

Los trabajos más profundos y destacables de aplicación de redes de Petri a procesos de negocio han sido desarrollados por Aalst. Entre mediados y finales de los 90, este autor desarrolla el formalismo *Workflow Nets*, basado en redes de Petri de *alto nivel* (formalismo que añade *color*, *tiempo* y *jerarquía* al modelo clásico de redes de Petri). En la figura 2.1 se muestra un ejemplo sencillo de un proceso modelado mediante redes de Petri. En la herramienta Woflan [121] implementa algoritmos de verificación de procesos de negocio implementados como *Workflow Nets*. En trabajos más recientes del mismo autor [8] se justifica, mediante un análisis basado en patrones de *workflow* [9], que los lenguajes basados en redes de Petri modelan bien patrones basados en estado, pero no determinados patrones más complejos (instancias múltiples, sincronizaciones complejas, retrocesos no locales, etc.) Por ello, Aalst ha propuesto un nuevo lenguaje, llamado YAWL (*Yet Another Workflow Language*) [8], diseñado a partir del formalismo de redes de Petri de alto nivel, pero capaz de modelar también dichos patrones complejos.

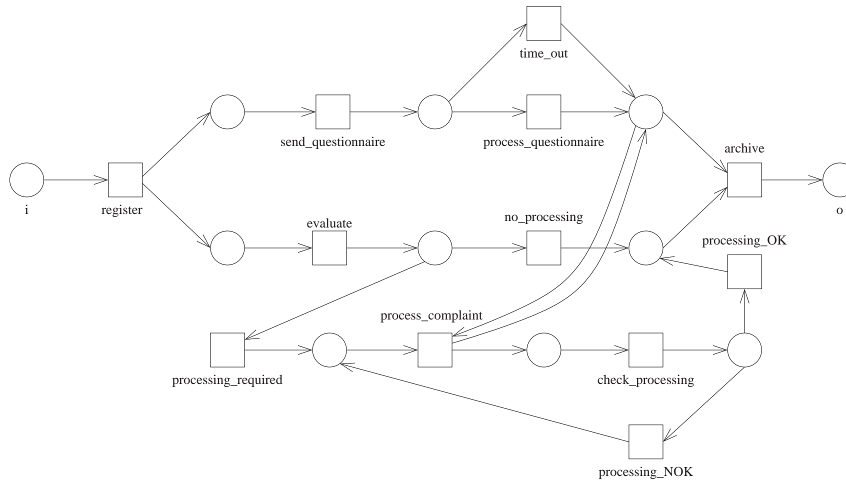

Figura 2.1: Proceso de gestión de reclamaciones modelado con redes de Petri (extraído de [2]). Los círculos representan *lugares*, y los cuadrados, *transiciones*. En el formalismo Workflow Nets las actividades del proceso se representan como transiciones.

Modelos basados en UML

En numerosos trabajos se ha investigado la utilización de UML [63] para el modelado de procesos de negocio. Los diagramas de actividad son el componente de UML más empleado con esta finalidad. Es más, en [41] se justifica que el paradigma de orientación a objetos no resulta adecuado para el modelado de procesos de negocio, y que por tanto UML tampoco, exceptuando los diagramas de actividad, que no se basan en este paradigma.

El trabajo más profundo utilizando diagramas de actividad es la tesis doctoral de Eshuis [41]. En este trabajo se analizan distintas semánticas formales propuestas en trabajos previos para diagramas de actividad, se justifica que ninguna de ellas resulta adecuada para el modelado de procesos de negocio, se proponen dos semánticas formales [42, 43] y se presenta una herramienta de verificación funcional, aplicando *model checking* a dichas semánticas [44].

En [36] se analiza la capacidad de los diagramas de actividad de UML para representar distintos patrones de *workflow* [9].

Otros lenguajes y formalismos

Además de los lenguajes basados en redes de Petri y diagramas de actividad de UML, se han desarrollado otros lenguajes y formalismos para el modelado de procesos de negocio. En este apartado se mencionan los más destacados:

- XPDL [127] especifica un lenguaje de intercambio de procesos de *workflow*, con el objetivo de ser un lenguaje común que se pueda exportar o importar en cualquier herramienta de *workflow*. Se define como una aplicación de XML. Forma parte de la interfaz 1 (*intercambio de procesos de negocio*) de la arquitectura de referencia del WfMC [71].

- Ould [107] define un lenguaje gráfico, centrado en el modelado de los papeles de los distintos tipos de actores y en la colaboración entre ellos. No se define formalmente su semántica.

- EPC (*Event–driven Process Chains*) forman parte de ARIS [118]. No tienen una semántica formal, aunque están inspiradas en redes de Petri.

- AMBER [37] (*Architectural Modelling Box for Enterprise Redesign*) es un lenguaje de modelado de procesos de negocio. En [76] se define una semántica formal en Promela que permite verificar procesos de negocio con Spin.

- En el proyecto Mentor [103] se utiliza STATEMATE [67], una entorno de modelado de sistemas sistemas de tiempo real, para modelar procesos de negocio. STATEMATE está basado en *statecharts* [66], un formalismo relativamente semejante a los diagramas de actividad de UML, pero con distinta semántica y capacidad expresiva.

- Recientemente se propone la utilización de modelos basados en π*–calculus* [99] para el modelado de procesos de negocio. Aalst defiende justificadamente en [4] la utilización de redes de Petri frente a π-calculus.

- BPEL4WS y BPML, dos lenguajes de definición de procesos de negocio con notación XML, orientados a la utilización de servicios Web como mecanismo de comunicación. Se presentan en el apartado 2.3.

2.2.3. Patrones de *workflow*

Dada la variedad de lenguajes y formalismos de modelado de procesos de negocio existentes, un grupo de trabajo, liderado por Aalst en *Technische Universiteit Eindhoven* y *Queensland University of Technology*, comenzó en 1999 a trabajar con el objetivo de proporcionar un marco en el cual poder comparar la expresividad de distintos lenguajes de modelado de procesos de negocio.

Para ello, han definido un conjunto de 20 patrones, llamados *patrones de workflow*. Cada patrón propone una funcionalidad que resulta útil en el modelado de procesos de negocio. La referencia básica que describe estos patrones es [9], aunque los autores han publicado anteriormente varios trabajos con resultados previos. Clasifican en seis categorías los *patrones de workflow*:

- Patrones de control básico: modelan estructuras básicas de control en procesos de negocio, como la ejecución en secuencia, ejecución concurrente, sincronización de actividades concurrentes, selección de un camino de ejecución entre múltiples alternativas y unión simple de ramas de ejecución alternativas.

- Patrones de bifurcación y sincronización avanzados: forman parte de esta categoría el patrón de selección de varias ramas de ejecución de entre múltiples alternativas, así como varios patrones de mezcla de ramas de ejecución.

- Patrones estructurales: se engloban en esta categoría el patrón de ejecución de bucles arbitrarios y de finalización implícita del proceso cuando no hay actividades pendientes de ejecución.

- Patrones de instancias múltiples: patrones relacionados con la posibilidad de instanciación múltiple de actividades.
- Patrones basados en estado: ejecución de actividades en orden aleatorio (pero no en paralelo), selección implícita de una rama de ejecución entre varias alternativas, y la posibilidad de ejecución de una actividad dada sólo hasta que el proceso alcance un determinado hito.
- Patrones de cancelación: patrones de cancelación de actividades y del proceso.

Estos patrones han sido aplicados a diversos lenguajes de modelado de procesos de negocio definidos en herramientas comerciales, estándares y diversos trabajos de investigación [125, 7, 6, 3, 80].

2.2.4. Análisis de procesos de negocio

Se puede definir *análisis de procesos de negocio* como un conjunto de técnicas cuyo objetivo es obtener propiedades relevantes acerca de definiciones de procesos de negocio para realizar razonamientos acerca de los mismos.

Las técnicas de análisis de procesos de negocio se pueden clasificar, según su objetivo, en dos grupos principalmente: verificación funcional y análisis de prestaciones. La verificación funcional consiste en comprobar si una definición de un proceso es compatible con su especificación (o requisitos), con el objetivo de detectar errores de funcionamiento en los procesos durante la fase de diseño de los mismos. El objetivo del análisis de prestaciones es obtener estadísticas acerca de las prestaciones del proceso, desde el punto del cliente (tiempos de respuesta, principalmente), o de la propia organización que ejecuta el proceso (utilización de recursos). El análisis de prestaciones es importante para realizar *reingeniería de procesos de negocio* (BPR), que consiste en rediseñar las definiciones de procesos para mejorar sus prestaciones.

Para realizar verificación funcional de procesos se están utilizando técnicas previamente aplicadas en otros campos, y presentadas en el apartado 2.1. Las principales son simulación guiada, pruebas, verificación deductiva y *model checking*. Para realizar análisis de prestaciones, se emplean también técnicas utilizadas en otros campos, para el diseño de redes de telecomunicaciones, sistemas operativos o sistemas de producción, entre otros. Las principales técnicas se basan en teoría de colas [62] y simulación [21].

Dado que esta tesis doctoral se centra en la verificación funcional de procesos de negocio, se profundizará en este tipo técnicas en el apartado 2.4, en el que se exponen los principales trabajos relacionados.

2.3. Composiciones y coordinaciones de servicios Web

Según el *World Wide Web Consortium* (W3C), se define un servicio Web como *un sistema software diseñado para dar soporte a interacciones de máquina a máquina interoperables sobre la red. Posee una interfaz descrita en un formato procesable por una máquina (WSDL específicamente). Otros sistemas interactúan con el servicio Web tal y como se especifica en su descripción utilizando mensajes SOAP, que típicamente se transmiten utilizando*

HTTP con serialización XML y en conjunción con otros estándares relacionados con la Web [130].

En general, los servicios Web son una tecnología de *middleware* muy adecuada para el desarrollo de *arquitecturas orientadas a servicio* (*Service Oriented Architecture*, SOA). En aplicaciones reales basadas en SOA, las invocaciones a servicios no suelen ser independientes, sino que es necesario habitualmente realizar invocaciones coordinadas a más de una operación de un servicio Web e incluso a operaciones de distintos servicios Web. Esta coordinación suele estar regida por restricciones en el orden en el cual las operaciones pueden ser invocadas. Esto da lugar a necesidades en dos perspectivas distintas: externa (interacción) e interna (implementación). Desde el punto de vista de la perspectiva externa, es necesario algún tipo de abstracción que permita definir las restricciones que rigen una interacción compleja entre varias entidades. Desde el punto de vista de la perspectiva interna a una entidad, es necesario proporcionar herramientas que faciliten a cada entidad la implementación de la lógica necesaria para participar en la interacción, manteniendo el contexto entre las distintas invocaciones y estableciendo qué operaciones se deben realizar en cada momento. La *coordinación de servicios Web* se refiere a la perspectiva externa, mientras que la *composición de servicios Web* se refiere a la perspectiva interna. En los próximos apartados se desarrollan estos conceptos.

En [94] se destaca la importancia de los servicios Web y las composiciones de servicios Web en el desarrollo de plataformas para la interacción B2B.

Desde el punto de vista de la gestión de procesos de negocio, los servicios Web y SOA prometen también ser una tecnología clave. Por una parte, suponen un mecanismo de comunicación idóneo para la integración de aplicaciones heterogéneas en los procesos de negocio internos de una organización. Por otra, permiten implementar procesos de negocio llevados a cabo entre distintas organizaciones.

En el entorno interno de una organización, los servicios Web suponen un mecanismo adecuado, tal y como se justifica en [13], para *Integración de Aplicaciones Empresariales* (*Enterprise Application Integration*, EAI). En este caso la organización dispone de diversas aplicaciones, que proporcionan acceso mediante mecanismos heterogéneos, y deben ser integradas en el sistema BPM. Para ello, se implementan *envoltorios*[4] que dan acceso a la funcionalidad de la aplicación mediante servicios Web, homogeneizando de esta forma el acceso a estas aplicaciones desde BPMSs y otras aplicaciones cliente.

Los procesos de negocio entre organizaciones se caracterizan por ser llevados a cabo como una colaboración entre varias organizaciones. Su principal característica reside en el hecho de que no existe un BPMS centralizado, sino varios BPMSs heterogéneos y distribuidos en las distintas organizaciones. En este entorno cada organización da acceso a sus servicios y a su BPMS a otras organizaciones mediante servicios Web. Típicamente, el orden en que se deben invocar las distintas operaciones de los servicios que componen estas colaboraciones está sujeto a determinadas restricciones.

En ambas situaciones, el proceso de negocio, ya sea interno a una organización o implementado entre varias organizaciones, puede ser visto como una composición o una coordinación de servicios Web.

[4] En inglés, *wrappers*.

2.3.1. Composición de servicios Web

La *composición de servicios Web* consiste en el desarrollo de un servicio complejo mediante combinación de la funcionalidad ofrecida por otros servicios Web. Dado que la implementación de estos servicios compuestos no es, en absoluto, trivial [13], es necesario contar con *middleware* de composición de servicios, que provea abstracciones e infraestructura para la definición y ejecución de servicios compuestos.

Con el auge de los servicios Web, comenzó a ser patente la necesidad de desarrollar un *middleware* de composición. Uno de los aspectos clave de este *middleware* es un lenguaje de modelado de definiciones de servicios compuestos. Inicialmente distintas organizaciones propusieron lenguajes como XL [52], XLANG [95], WSFL [89] o BPML [15]. Posteriormente, IBM, BEA Systems y Microsoft aunaron esfuerzos y definieron el lenguaje BPEL4WS (*Business Process Execution Language for Web Services*) [14] partiendo de XLANG y WSFL. Actualmente, BPEL4WS es el lenguaje dominante para composición de servicios Web, y está en proceso de estandarización dentro de OASIS.

El objetivo de un *middleware* de composición de servicios es facilitar la implementación de un servicio compuesto, permitiendo a los desarrolladores trabajar a nivel de lógica de negocio sin necesidad de descender a detalles de bajo nivel. Los componentes principales de este *middleware* son: un modelo y lenguaje de definición de servicios compuestos, un entorno de desarrollo y un entorno de ejecución. El modelo de definición permite especificar los servicios compuestos (servicios que deben ser invocados y en qué orden, cómo se construyen los mensajes de entrada de cada invocación, etc.) El entorno de desarrollo proporciona una interfaz gráfica para facilitar el diseño de procesos. El entorno de ejecución ejecuta la lógica de negocio de procesos compuestos.

Un servicio compuesto se define desde el punto de vista de varias dimensiones. En [13] se identifican seis dimensiones:

- Modelo de componentes: define la naturaleza de los elementos a ser compuestos de acuerdo con las suposiciones que el modelo hace acerca de ellos.
- Modelo de orquestación: define abstracciones y lenguajes para especificar el orden en que los servicios deben ser ejecutados. Se fundamenta en los mismos formalismos que los lenguajes de definición de procesos de negocio: diagramas de actividad, redes de Petri, *statecharts* y álgebras de procesos, principalmente.
- Modelo de datos y acceso a datos: define cómo se especifican los datos utilizados en la composición y cómo son intercambiados entre los distintos componentes.
- Modelo de selección de servicio: describe cómo se seleccionan, estática o dinámicamente, los servicios a ser utilizados como componentes.
- Transacciones: define las semánticas transaccionales a aplicar en la composición.
- Gestión de excepciones: define cómo se gestionan los errores y situaciones excepcionales ocurridas durante la ejecución de la composición.

Como se ha apuntado anteriormente, la composición de servicios Web supone una tecnología adecuada para la implementación de BPMSs y EAI, e incluso, tal y como se expone

en [13], puede llegar a sustituir a BPMSs en algunos entornos gracias a su mayor sencillez y facilidad de uso. La mayor desventaja de la tecnología de composición de servicios Web radica en que se basa en que todos los componentes sean servicios Web. Para integrar en las composiciones aplicaciones que no implementen interfaces mediante servicios Web, es necesario programar aplicaciones adaptadoras (por ejemplo, *envoltorios*).

Dado que BPEL4WS es, a día de hoy, la principal iniciativa de estandarización en este campo, se describe en el apartado 2.3.3. En [110] se justifica la importancia de BPEL4WS en las arquitecturas orientadas a servicios y en el desarrollo de aplicaciones basadas en servicios Web. En [96] se exponen y analizan otras alternativas para la composición de servicios Web.

2.3.2. Coordinación de servicios Web

Las definiciones de WSDL permiten describir interacciones simples en las cuales sólo se invoque una operación de forma aislada, o varias operaciones de forma independiente. Sin embargo, cuando son necesarias interacciones más complejas, WSDL no proporciona los mecanismos adecuados para garantizar la corrección y consistencia de las operaciones. Por ejemplo, supongamos que se define un servicio Web de venta que proporciona una operación para recibir pedidos, pero confirma los pedidos varios días más tarde mediante la invocación de una operación de *call–back* proporcionada por el cliente. WSDL resulta insuficiente para la situación planteada en el ejemplo, dado que no permite especificar que el cliente debe implementar una determinada operación de *call-back* para recibir la respuesta.

El término de *coordinación de servicios Web* engloba a los protocolos, abstracciones e infraestructuras necesarias para dar soporte a interacciones realizadas mediante múltiples invocaciones a servicios Web que presenten algún tipo de interdependencia.

La diferencia entre coordinación y composición radica en que la composición define el comportamiento interno que debe implementar una entidad para participar en una interacción compleja, mientras que la coordinación define cómo las distintas entidades involucradas en la interacción deben interactuar, ocultando cualquier detalle acerca de la implementación interna de las entidades participantes. Por ello, es habitual referirse a coordinaciones como *procesos públicos* o *protocolos de negocio* y a las composiciones como *procesos privados*.

WS–Coordination, definido por IBM, Microsoft y BEA Systems en Agosto de 2002, cuya última especificación se presenta en [86], es una meta–especificación, en el sentido de que crea un marco genérico en el cual se pueden definir protocolos de coordinación específicos. Por ejemplo, el soporte de transacciones para servicios Web, definido originalmente en *WS–Transaction* en 2002, se especifica en el marco proporcionado por *WS–Coordination*. La especificación de *WS–Transaction* has sido dividida posteriormente en dos especificaciones: *WS–AtomicTransaction* [84] (transacciones de corta duración con propiedades ACID) y *WS–BusinessActivity* [85] (protocolos de negocio de larga duración).

Actualmente existen otras iniciativas y propuestas de protocolos de coordinación de servicios Web. El W3C, a través de su grupo de trabajo *Web Services Choreography*, está definiendo un estándar de coordinación llamado *Web Services Choreography Description Language* (WSCDL) [79], que en el momento de escribir este documento es todavía un borrador de trabajo. *Web Services Choreography Interface* (WSCI) [16] es un lenguaje de coordinación de servicios Web definido por Intalio, SAP, BEA y Sun en 2002. *Web Services Conversation Language* (WSCL) [18] es un lenguaje de coordinación propuesto en 2002 por HP.

Por otra parte, como se ha mencionado anteriormente, BPEL4WS también permite definir coordinaciones de servicios Web mediante el concepto de *procesos abstractos*.

Además de las propuestas anteriores, también se considera en [13] que RosettaNet, xCBL y ebXML definen protocolos de coordinación para ámbitos de uso más específicos.

2.3.3. El lenguaje BPEL4WS

Business Process Execution Language for Web Services (BPEL4WS) es una notación para especificar procesos de negocio basados en servicios Web, esto es, que sólo pueden importar y exportar funcionalidad mediante servicios Web. La especificación inicial (BPEL4WS 1.0) fue desarrollada por IBM, Microsoft y BEA, tomando como referencia XLANG, de Microsoft, y WSFL, de IBM, y publicada en Julio de 2002. Su especificación actual (BPEL4WS 1.1), de mayo de 2003, introduce algunos cambios en la notación con respecto a la primera versión, y a ella se han unido SAP y Siebel Systems como coautores.

BPEL4WS permite especificar tanto *procesos abstractos* (protocolos de negocio) como *procesos ejecutables*. Los procesos abstractos especifican la secuencia de mensajes que deben ser intercambiados por varias entidades que participan en un protocolo de negocio, pero sin revelar aspectos internos de implementación en cada entidad. Un proceso ejecutable, por el contrario, especifica el comportamiento interno de una entidad, de tal forma que pueda ser ejecutado automáticamente por un motor de ejecución. Por tanto, los procesos abstractos especifican coordinaciones de servicios Web, mientras que los procesos ejecutables especifican composiciones de servicios Web. Ambos tipos de procesos se especifican utilizando la misma notación básica, común para ambos, y extensiones específicas para cada uno de ellos. Las extensiones son pocas comparadas con la notación básica, de tal forma que la mayor parte de la notación es común para ambos tipos.

En el capítulo 7 se presenta una introducción al lenguaje BPEL4WS.

2.3.4. Relación entre BPML y BPEL4WS

El consorcio BPMI.org (*Business Process Management Initiative*) tiene como objetivo potenciar la utilización de tecnologías basadas en procesos de negocio a nivel empresarial. La principal contribución de este organismo es *Business Process Modeling Language* (BPML) [15]. Este lenguaje permite modelar los procesos de negocio internos de una organización. Su objetivo, inicialmente, era convertirse en un estándar.

BPML tuvo su origen en el mundo de la gestión de procesos de negocio, sin presuponer comunicaciones basadas en servicios Web. Sin embargo, BPEL4WS ha sido desarrollado desde un principio centrado en la tecnología servicios Web. A pesar de ello, BPML y BPEL4WS resultaron ser muy semejantes tanto en objetivos como en características técnicas. BPML es ligeramente más expresivo, y proporciona alguna característica ausente en BPEL4WS, como la posibilidad de anidar subprocesos. Sin embargo, BPEL4WS ha recibido finalmente más apoyos en el mundo de la industria, y el desarrollo de BPML se encuentra paralizado desde hace dos años. BPMI.org ha publicado recientemente la última versión de BPMN (*Business Process Modeling Notation*) [123], que define una notación gráfica para procesos de negocio. Esta versión de la especificación ha sido editada por White, de IBM (empresa que ha participado desde un principio en el desarrollo de BPEL4WS), y en ella se

define cómo generar a partir de dicha notación procesos BPEL4WS, pero no se hace referencia a BPML. Esto demuestra que el propio consorcio BPMI.org ha reconocido la derrota de BPML.

2.4. Trabajos relacionados

En el apartado anterior se han presentado distintas tecnologías relacionadas con los objetivos de esta tesis. En este apartado se realiza un análisis de distintos trabajos cuya temática está relacionada con el ámbito de la misma: verificación funcional de procesos de negocio y composiciones de servicios Web.

Dada la gran variedad de lenguajes de definición de procesos de negocio existentes, cada uno de estos trabajos está orientado a un lenguaje de definición de procesos distinto. Por otra parte, abordan el problema utilizando distintos métodos formales, como redes de Petri, álgebras de procesos, o sistemas de transiciones etiquetados en combinación con herramientas de *model checking*. Como se verá, estos proyectos difieren también en el tipo de propiedades que se permiten verificar en los procesos. El cuadro 2.3 presenta un resumen con las características principales de los mismos.

En los próximos apartados se analizan estos trabajos. Finalmente, como conclusiones de este análisis, se expone en el apartado 2.5 cuáles son los principales problemas que permanecen sin resolver en este ámbito, o cuya solución podría ser mejorada.

2.4.1. Woflan (Technische Universiteit Eindhoven)

Woflan [121] es una herramienta, cuyo desarrollo de inició a finales de 1996, que permite detectar errores en definiciones de procesos de negocio mediante la utilización de técnicas de análisis de redes de Petri. La implementación es compatible con los lenguajes de definición de procesos de varias herramientas de gestión de *workflow* comerciales: *COSA*, *Staffware*, *METEOR* y *Protos*.

Un objetivo clave de Woflan es demostrar si un determinado proceso es o no *correcto*[5]. Para ello, los autores definen un proceso *correcto* como aquel que cumple las siguientes reglas de corrección: es siempre posible finalizar las instancias del proceso (esto es, el proceso está libre de *abrazo mortal*), finalización adecuada (no puede notificar la finalización de una instancia del proceso si hay trabajo sin concluir relativo a dicha instancia) y ausencia de actividades muertas (dada una definición de un proceso, no debe existir ninguna actividad cuya ejecución sea imposible). Aunque el hecho de que un proceso sea *correcto* no garantiza que se comporte tal y como el diseñador pretende, este tipo de verificación consigue detectar errores habituales de diseño.

La verificación está basada en una semántica desarrollada en trabajos previos de los mismos autores, que permite expresar procesos de negocio en forma de redes de Petri, llamada *Workflow Nets* [1] (*WF nets*). Una *WF net* es una subclase de las redes de Petri, especialmente diseñada para la definición de procesos de negocio desde el punto de vista de la perspectiva de *control de flujo* (véase la definición de perspectivas en el apartado 2.2.1). En dicho trabajo, la subclase de las *WF nets* se define formalmente mediante la aplicación de restricciones en la estructura a la definición de red de Petri.

[5] *Sound*, en inglés.

Análisis

Los principales contribuciones de este proyecto son los siguientes:

- Supone el primer trabajo en que se aborda el problema de la verificación de procesos de negocio. Justifica la necesidad de verificar formalmente procesos de negocio.
- Se ha llevado a cabo con un profundo rigor matemático. Todos los conceptos se definen formalmente, y los algoritmos de verificación se apoyan en proposiciones y teoremas demostrados.
- Se muestra que las redes de Petri son un formalismo adecuado para modelar procesos de negocio desde la perspectiva de control de flujo.
- Se ha desarrollado una herramienta que aplica los resultados del proyecto y es compatible con cuatro lenguajes de definición de procesos de herramientas de gestión de *workflow* comerciales.

Los puntos débiles identificados en este proyecto son los siguientes:

- El formalismo *WF nets* sólo permite modelar y verificar procesos de negocio desde la perspectiva de control de flujo, abstrayendo el resto de perspectivas. Por tanto, no es posible modelar la perspectiva de datos y, en consecuencia, no se pueden realizar verificaciones con respecto al estado de las entidades u objetos manejados por el proceso.
- El formalismo *WF nets* presenta algunas limitaciones desde el punto de vista de los *patrones de workflow*. En [7] se identifican varios patrones relacionados con sincronización avanzada, cancelación y múltiples instancias que nos son fácilmente modelables con este formalismo. Algunos de estos patrones pueden aparecer en procesos definidos en lenguajes como BPEL4WS. Comparando los análisis de [7] y [125], se observa que los patrones número 7 (*mezcla con sincronización*), 11 (*terminación implícita*), 19 (*cancelación de actividad*) y 20 (*cancelación de caso*) pueden ser modelados con BPEL4WS, pero no con formalismos basados en redes de Petri.
- La verificación está orientada al concepto de *corrección*. No se contempla la verificación de otros tipos de propiedades, como las expresadas mediante lógicas temporales, por ejemplo.

2.4.2. Tesis doctoral de Eshuis (Universiteit Twente)

El objetivo de la tesis doctoral de Eshuis [41], presentada a finales de 2002, es definir una semántica formal para diagramas de actividad de UML que resulte adecuada para la definición de procesos de negocio. Por otra parte, es un requisito fundamental para dicha semántica permitir la verificación funcional de los procesos.

El autor justifica que los diagramas de actividad de UML carecen de una semántica formal, por lo cual no resultan adecuados para la definición o verificación de procesos de negocio. Por otra parte, sostiene justificadamente que las semánticas comúnmente utilizadas en redes de Petri no siempre resultan adecuadas para la definición de procesos de negocio.

Por ello, en la tesis define semánticas formales para diagramas de actividad orientadas a la definición de procesos de negocio. Para demostrar la aplicabilidad de dichas semánticas, desarrolla una herramienta capaz de generar modelos verificables en el *model checker* SMV a partir de procesos definidos conforme a dichas semánticas.

En este trabajo se justifica que un proceso de negocio es un sistema reactivo, esto es, un sistema que evoluciona como respuesta a la ocurrencia de estímulos o eventos originados en su entorno. Teniendo en cuenta esto, se justifica que las semánticas utilizadas habitualmente en redes de Petri no resultan adecuadas para sistemas reactivos. La justificación se basa en dos motivos, principalmente. Por una parte, la forma de modelar un evento externo mediante redes de Petri no es sencilla. Por otra, el hecho de que una transición cuente con marcas en todos sus lugares de entrada implica que dicha transición se puede activar, pero no que se tenga que activar necesariamente (por tanto, aunque un sistema reactivo debe responder a la ocurrencia de un evento, con esta semántica no se puede garantizar que esto sea así). Debido a ello, las semánticas desarrolladas en el trabajo para el modelado de procesos de negocio se basan en el formalismo *statecharts*.

En lugar de desarrollar una única semántica, se desarrollan dos: una a nivel de requisitos y otra a nivel de implementación.

Análisis

Las principales aportaciones de este trabajo son las siguientes:

- Pese a que en los últimos años se han realizado varias propuestas para utilizar UML, y más concretamente los diagramas de actividad, para la definición de procesos de negocio, este trabajo supone la aproximación más rigurosa al problema.
- Se definen dos semánticas formales para diagramas de actividad de UML, orientadas al modelado de procesos de negocio.
- Se justifica que los formalismos basados en *statecharts* resultan más adecuados para el modelado de procesos de negocio que los basados en redes de Petri.
- Para demostrar que el formalismo resulta adecuado para realizar verificaciones, se ha desarrollado una herramienta que transforma procesos de negocio expresados mediante diagramas de actividad a SMV.

Los puntos débiles identificados en este proyecto son los siguientes:

- Las semánticas desarrolladas permiten modelar y verificar procesos de negocio desde la perspectiva de control de flujo únicamente. Al igual que en otros trabajos, no es posible modelar la perspectiva de datos.
- No se ha estudiado todavía la expresividad de las semánticas desarrolladas con respecto a los patrones de *workflow*.
- Es posible definir especificaciones utilizando lógicas temporales, pero para hacerlo es necesario conocer los símbolos generados en la transformación del modelo a SMV.

2.4.3. Web Service Engineering Project (Imperial College London)

En este proyecto, iniciado en 2003 y todavía en progreso, los autores proponen un entorno para la verificación de composiciones de servicios Web definidas mediante BPEL4WS. Según los resultados presentados hasta el momento [54, 55], se ha implementado una herramienta capaz realizar verificaciones sobre un proceso BPEL4WS y sobre la ejecución concurrente de varios procesos BPEL4WS que se comunican entre sí.

Para ello, se transforman los procesos BPEL4WS a modelos expresados mediante el formalismo *Finite State Process* (FSP). Estos modelos se verifican con la herramienta *Labeled Transition System Analyzer* (LTSA) [90]. La herramienta es capaz de detectar situaciones de *abrazo mortal*, así como demostrar si el proceso es equivalente o no a un *Message Sequence Chart* (MSC) dado. Por otra parte, el sistema puede ser aplicado a otros lenguajes de definición de procesos de negocio, si se implementa una herramienta de transformación desde dichos lenguajes a FSP.

Análisis

Las principales aportaciones de este trabajo son las siguientes:

- Es el primer trabajo desarrollado que permite verificar procesos BPEL4WS, lo cual resulta de gran importancia dado el auge actual de esta tecnología para la composición y coordinación de servicios Web.

- La mayoría de los trabajos de verificación de procesos de negocio permiten verificar únicamente un proceso de forma aislada. El entorno de dicho proceso debe ser abstraído. Sin embargo, este trabajo permite verificar la ejecución concurrente de varios procesos de negocio que se comunican entre sí. Esto permite detectar errores que no se podrían detectar si cada uno de los procesos de verifica de forma aislada.

- Se ha desarrollado un prototipo de la herramienta en forma de *plugin* para una herramienta de verificación ya existente (LTSA).

En el análisis de este proyecto hay que tener en cuenta que es un trabajo todavía en curso, y que por tanto es de esperar que algunas de las limitaciones se resuelvan en futuras versiones. Las principales limitaciones de este proyecto son las siguientes:

- No da soporte a la especificación completa de BPEL4WS 1.1. Presenta bastantes limitaciones para la verificación de gestores de fallos y compensación dentro de *entornos* anidados.

- Las propiedades a verificar han de expresarse utilizando el formalismo FSP. Por tanto, es necesario que la persona que realiza la verificación conozca este formalismo. Por otra parte, esto conlleva también que sólo se puedan enunciar las propiedades a verificar una vez se ha realizado la traducción del proceso a FSP, dado que es necesario conocer los nombres de los símbolos (nombres de variables, actividades, etc.) del modelo FSP del proceso.

2.4.4. Otros trabajos relacionados

En [82] se aborda el problema de la verificación funcional de BPEL4WS utilizando álgebras de procesos. Para ello, se define un formalismo intermedio llamado BPE-Calculus, y se formaliza en términos de álgebras de procesos. Modelos expresados mediante este formalismo pueden ser analizados utilizando la herramienta CWB. Los procesos BPEL4WS son automáticamente compilados a BPE-Calculus, y posteriormente analizados en CWB. Este trabajo permite verificar sólo un subconjunto de BPEL4WS y no tiene soporte, por ejemplo, para los mecanismos de gestión de fallos ni de compensación. Por otra parte, ha sido validado únicamente con procesos de muy pequeño tamaño.

En [104] el autor propone la utilización del *model checker* Spin para verificar composiciones de servicios Web expresadas mediante el lenguaje WSFL, predecesor de BPEL4WS. Para ello, traduce las composiciones WSFL directamente al lenguaje Promela utilizado por Spin.

En [22] se presenta una formalización de coordinaciones de servicios Web expresadas mediante WSCI (*Web Service Choreography Interface*) [16]. La principal aplicación que proponen para esta formalización es comprobar si dos servicios dados son compatibles para interactuar en un protocolo de negocio. La formalización se hace mediante CCS [98].

2.5. Conclusiones

Hasta ahora son varios los trabajos en que se ha abordado el problema de la verificación funcional de procesos de negocio. Como se ha visto, cada uno de ellos está centrado en un formalismo distinto, y utiliza en consecuencia distintas técnicas y herramientas de verificación. Tras realizar un análisis de los más relevantes, se han identificado varias carencias en estas propuestas, que se describen en mayor profundidad en los siguientes apartados:

- No se tiene en cuenta la perspectiva de datos.
- No es sencillo expresar propiedades de verificación demasiado específicas.
- Las soluciones están ligadas a lenguajes de definición de procesos y a herramientas y formalismos de verificación concretos.

No se tiene en cuenta la perspectiva de datos

Todos los trabajos presentados están orientados a la perspectiva de control de flujo. Los principales formalismos empleados hasta ahora, como redes de Petri, *statechart* o diagramas de actividad no modelan, al menos de forma sencilla, el concepto de *variable*. Sin embargo, nuevos lenguajes como BPEL4WS y BPML sí modelan la perspectiva de datos, incorporando variables como parte fundamental del lenguaje. Por tanto, aunque la perspectiva de control de flujo es, sin duda, la más importante en un proceso de negocio desde el punto de vista de funcionalidad, es necesario tener en cuenta también la perspectiva de datos en los formalismos utilizados para realizar la verificación. De entre los trabajos presentados anteriormente, sólo la solución propuesta en *Imperial College* permite modelar variables, aunque es necesario expresar propiedades en su formalismo intermedio, FSP, para aprovechar esta propiedad.

Dificultad para expresar especificaciones

Desde el punto de vista de la definición de las especificaciones, esto es, el conjunto de propiedades que el diseñador del proceso espera que sean ciertas en el proceso diseñado, identificamos dos carencias fundamentalmente: rigidez para definir especificaciones y necesidad de utilizar el formalismo final en lugar del formalismo de modelado.

La rigidez se da en aquellos sistemas que sólo permiten verificar un conjunto de propiedades genéricas, y no permiten que el diseñador defina propiedades específicas para un determinado proceso. Es el caso, por ejemplo, de Woflan, que se centra en la propiedad genérica de *corrección*.

En otros trabajos, como los de Twente o Imperial College, el diseñador sí puede definir propiedades específicas. Sin embargo, en el primero han de ser definidas directamente en lenguaje SMV y, en el segundo, mediante FSP. Esto tiene varios inconvenientes. Por una parte, el diseñador debe conocer tanto los lenguajes de definición de procesos (diagramas de actividad y BPEL4WS, respectivamente) como los formalismos utilizados en la verificación (SMV y FSP, respectivamente). Por otra, no se pueden definir estas propiedades *a priori*: es necesario realizar antes la traducción desde el lenguaje de definición al formalismo de verificación para poder expresar las propiedades en términos de los símbolos generados en esta traducción.

Idealmente, debería ser posible definir el proceso y su especificación en el mismo dominio, de tal forma que ambas sean traducidas en el mismo momento al formalismo empleado para la verificación. Por ejemplo, en el caso de procesos definidos mediante BPEL4WS se emplea XPath como lenguaje de expresiones. Por tanto, sería interesante poder definir la especificación mediante XPath, pudiendo referenciar actividades y variables con el nombre mediante el cual se declaran en el propio documento BPEL4WS.

Lenguajes de definición y formalismos de verificación específicos

Los trabajos analizados presentan soluciones al problema de la verificación de procesos de negocio que son específicas para lenguajes de definición de procesos concretos y que emplean formalismos y herramientas de verificación concretos. Desde nuestro punto de vista, supondría una aportación interesante proporcionar un entorno no ligado a lenguajes y formalismos específicos, sino que fuese lo suficientemente modular y extensible como para integrar diversos lenguajes de definición y formalismos de verificación. Para justificar nuestro punto de vista, nos basamos en los siguientes hechos:

- Como se ha presentado en el estudio del estado del arte, en la actualidad se están utilizando, tanto a nivel de investigación como en sistemas comerciales, muy diversos lenguajes de definición de procesos de negocio distintos. Aunque hay varias iniciativas que intentan estandarizar un lenguaje de definición de procesos, no parece nada probable a corto o medio plazo que se imponga alguna de estas iniciativas.

- Especialmente en el entorno académico, se han desarrollado una gran cantidad de formalismos y herramientas de verificación. Es difícil comparar varios formalismos o herramientas y afirmar que uno de ellos es mejor, dado que habitualmente depende en gran medida del tipo de modelo y propiedades a verificar.

Teniendo en cuenta estos hechos, consideramos que las principales ventajas de un entorno genérico, modular y extensible serían las siguientes:

- Una solución genérica tendría un rango de aplicación considerablemente más amplio que una solución ligada a un lenguaje de definición de procesos concreto.
- Dado un lenguaje de definición concreto, sería considerablemente más sencillo integrarlo en un entorno genérico ya existente que desarrollar un entorno de verificación completo para el mismo.
- Un entorno de verificación genérico, que integre varios formalismos y herramientas de verificación, permite seleccionar en cada momento el que resulte más adecuado para el tipo de proceso o de propiedades a verificar.

Proyecto	Lenguaje	Formalismo	Objetivos
Woflan (Technische Universiteit Eindhoven)	*(varios)*	redes de Petri	Demostrar si un proceso es o no *correcto*, según la definición que los autores dan a este término, desde la perspectiva de control de flujo. Utiliza determinadas técnicas de análisis de redes de Petri para demostrar propiedades que resultan relevantes en procesos de negocio. No permite verificar procesos desde la perspectiva de datos.
Tesis de H. Eshuis (Universiteit Twente)	*Diag. act. UML*	*SMV*	Definir una semántica a diagramas de actividad de UML para definir procesos de negocio. Verificar estos procesos utilizando *model checking*. No se contempla el modelado ni la verificación de la perspectiva de datos.
Web Services Engineering Project (Imperial College London)	*BPEL4WS*	*LTSA*	Verificar procesos BPEL4WS utilizando la herramienta LTSA, mediante una transformación al formalismo FSP. Permite comprobar la equivalencia de un proceso con una especificación MSC. Permite la verificación sólo de un subconjunto de la especificación de BPEL4WS. Permite la verificación desde la perspectiva de datos, si se especifican las propiedades en FSP.
BPE-Calculus (University of York, Canada)	*BPEL4WS*	álgebras de procesos	Definición del formalismo BPE-Calculus, que formaliza BPEL4WS en términos de álgebras de procesos. Verificar estos procesos utilizando la herramienta CWB. Permite la verificación sólo de un subconjunto muy reducido de la especificación de BPEL4WS.
Trabajos de N. Nakajima (Hosei University, Japón)	WSFL	Spin	Definición de reglas de transformación de composiciones de servicios Web WSFL a lenguaje Promela, verificables con Spin. La especificación se expresa en LTL, siendo necesario conocer el modelo generado en Promela para generarla. La verificación se centra en la perspectiva de control de flujo.

Cuadro 2.3: Resumen de trabajos relacionados.

Capítulo 3

Definición de una arquitectura de verificación

The first step toward the management of disease was replacement
of demon theories and humours theories by the germ theory.
That very step, the beginning of hope, in itself dashed
all hopes of magical solutions. It told workers that progress
would be made stepwise, at great effort, and that a persistent,
unremitting care would have to be paid to a discipline of cleanliness.
So it is with software engineering today.

(Fred Brooks, "No Silver Bullet: Essence and Accidents of Software Engineering",
IEEE Computer Magazine 20(3):10–19, 1987)

El objetivo principal de esta tesis doctoral es definir un nuevo entorno de verificación de requisitos funcionales de procesos de negocio, que solucione los problemas mostrados durante el estudio del estado del arte, en el apartado 2.5. Una de las contribuciones principales de esta tesis consiste en definir este entorno como una arquitectura *abierta*, *modular* y *extensible*.

Planteamos esta arquitectura por primera vez como tal en [45], a pesar de que la idea subyace en trabajos nuestros publicados con anterioridad.

En este apartado se presenta, de forma razonada, el diseño de esta arquitectura de verificación. En primer lugar, en el apartado 3.1, se describe la arquitectura. En el apartado 3.2 se justifica su adecuación a las necesidades planteadas. En el apartado 3.3 se analiza la capa clave de esta arquitectura, llamada capa del *Modelo Formal Común*. Finalmente, en el apartado 3.4, se plantean las principales conclusiones de este capítulo.

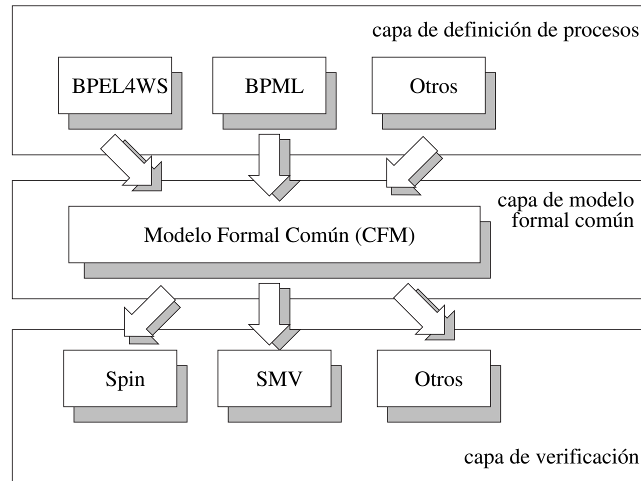

Figura 3.1: Arquitectura del entorno de verificación propuesto.

3.1. Una arquitectura para la verificación de requisitos funcionales

En esta tesis doctoral se propone una arquitectura para la verificación de requisitos funcionales de procesos de negocio basada en tres capas, tal y como se muestra en la figura 3.1. A continuación se describe brevemente cada una de estas capas:

- Capa de *definición de procesos* (capa 3): especificación de procesos de negocio y composiciones de servicios Web, mediante distintos lenguajes de alto nivel manejados por herramientas de gestión de procesos de negocio y de composición y coordinación de servicios Web. Es el caso, por ejemplo, de los lenguajes BPEL4WS o BPML.

- Capa de *modelo formal común* (capa 2): compuesta por un sistema formal y una notación que permiten definir procesos de negocio, orientados exclusivamente a la representación de aquellos aspectos que afecten a la verificación de requisitos funcionales, desde el punto de vista de las perspectivas de *datos* y *control de flujo*. Es un lenguaje intermedio entre los lenguajes de la capa 3 y las herramientas de verificación de la capa 1.

- Capa de *verificación* (capa 1): especificación de modelos verificables directamente mediante herramientas de verificación, como Spin o SMV.

La clave de esta arquitectura reside en el formalismo introducido en la capa 2, que permite desconectar los lenguajes de definición de procesos de los lenguajes empleados para la verificación de requisitos funcionales. Esto evita definir compiladores directos entre cada par de lenguaje de definición y lenguaje de verificación. En su lugar, se crea, para cada lenguaje de definición de procesos de negocio, un compilador al formalismo de la capa 2. A su vez, se crea un compilador desde el formalismo de la capa 2 hacia cada uno de los lenguajes empleados por las herramientas de verificación.

La idea subyacente en esta arquitectura es relativamente similar a la propuesta en el proyecto IF [20], aunque aplicada a un entorno distinto. En este proyecto, se propone un nuevo formalismo intermedio capaz de representar modelos expresados mediante otros lenguajes, como por ejemplo SDL y LOTOS. Por otra parte, integran distintas herramientas para dar soporte al análisis de modelos IF, como, por ejemplo, *model checkers*, simuladores interactivos de modelos, etc. Esta idea también subyace en Veritech [78].

3.2. Justificación

La arquitectura planteada en el apartado anterior resulta adecuada para resolver los problemas planteados en el estudio de los trabajos relacionados. En este apartado se justifica esta afirmación.

Por otra parte, se muestra cómo esta arquitectura de verificación, gracias al uso del formalismo intermedio, permite reducir considerablemente el número de herramientas de transformación necesarias.

3.2.1. Perspectiva de datos

Uno de los problemas planteados durante el estudio de los trabajos relacionados es la carencia de estos trabajos de un sistema adecuado de representación de la perspectiva de datos de los procesos de negocio.

Esta arquitectura, en sí misma, no soluciona este problema, sino que la capacidad para solucionarlo está en el sistema formal que se emplee en la capa 2. Por tanto, con el objetivo de que la arquitectura propuesta solucione este problema, se plantea como requisito para el diseño de este sistema formal que sea capaz de representar de forma adecuada la perspectiva de datos.

3.2.2. Expresividad de las especificaciones

Los trabajos relacionados analizados presentan, en general, limitaciones en la flexibilidad del diseñador para especificar requisitos a verificar en los procesos. Por otra parte, es habitual también que estos deban ser expresados directamente mediante el formalismo de verificación.

Por una parte, la arquitectura propuesta permite que un proceso pueda ser verificado mediante una o más de las herramientas integradas con la arquitectura. Esta variedad de herramientas contribuye a que se puedan especificar más tipos de requisitos.

Por otra parte, la arquitectura permite que se definan los requisitos a verificar utilizando los propios lenguajes de la capa 3. Estas propiedades son compiladas automáticamente, junto a la propia definición de los procesos, al sistema formal de la capa 2 y, finalmente, adaptadas a las herramientas de verificación de la capa 1.

3.2.3. Independencia del lenguaje de definición y formalismo de verificación

Todos los trabajos analizados están ligados a lenguajes de definición de procesos de negocio concretos, así como a formalismos y herramientas de verificación concretos.

Sin embargo, la arquitectura propuesta no presenta esta dependencia y es, por el contrario, *abierta*, *modular* y *extensible*:

- Es una *arquitectura abierta*: en principio, es posible integrar, hasta donde lo permita el sistema formal de la capa 2, cualquier lenguaje de definición de procesos de negocio y cualquier herramienta de verificación.

- En una *arquitectura modular*: cada uno de los lenguajes de la capa 3 integradas en la arquitectura, así como cada una de las herramientas de verificación integradas, constituye, junto con su compilador correspondiente, un módulo, independiente de los demás, y capaz de interactuar con el resto de módulos a través del formalismo de la capa 2.

- Es una *arquitectura extensible*: se puede añadir en cualquier momento módulos para nuevos lenguajes de definición de procesos o herramientas de verificación, sin afectar al resto de módulos previamente instalados en el sistema.

3.2.4. Número de herramientas de transformación necesarias

En un entorno como el actual, caracterizado por el uso de una amplia variedad de lenguajes de definición de procesos de negocio y de herramientas de verificación, la aplicación de las herramientas de verificación a lenguajes de definición debe ser lo más sencilla posible, y una clave para ello es la *reutilización*.

En un entorno con N lenguajes de definición de procesos de negocio y M herramientas de verificación, si se desease que cada una de las herramientas pudiese ser aplicada a cada uno de los lenguajes, serían necesarias $N \times M$ herramientas de transformación, una por cada par lenguaje–herramienta. Sin embargo, en una arquitectura como la propuesta, basada en la transformación a un formalismo intermedio, bastaría con $N + M$ herramientas de transformación, una para cada lenguaje, y una para cada herramienta.

Claramente, esta arquitectura permite *reutilizar* estas herramientas de transformación y así reducir considerablemente el número de herramientas de transformación necesarias.

3.3. El modelo formal común

El formalismo de la capa 2 es la pieza clave de la arquitectura propuesta, y es el objeto de los siguientes capítulos de esta tesis. Los principales requisitos que debe cumplir este formalismo para la que arquitectura planteada sea viable son los siguientes:

- Debe estar basado en una definición formalmente sólida, esto es, el formalismo debe ser definido matemáticamente. No debe haber ambigüedades ni inconsistencias en su definición.

- Debe proporcionar una expresividad suficiente para representar procesos definidos mediante los principales lenguajes de definición de procesos de negocio. Este requisito es fundamental para ampliar el rango de lenguajes integrables en la arquitectura.

- Debe utilizar, en su notación, conceptos lo más cercanos posible a los empleados en modelado de procesos basado en actividad. De esta forma, la transformación de procesos desde la capa 3 a la capa 2 será más natural y sencilla.
- Debe ser capaz de representar tanto la perspectiva de control de flujo como la perspectiva de datos de los procesos de negocio. Como se ha explicado anteriormente, este requisito es fundamental para solucionar uno de los problemas detectados en el análisis de trabajos relacionados.
- Debe ser representable mediante los sistemas formales de las principales herramientas de verificación de requisitos funcionales. Este requisito amplía el rango de herramientas integrables en la arquitectura.

Una posibilidad consiste en seleccionar un formalismo existente, conforme a los requisitos anteriores, para constituir la capa 2 de la arquitectura. Del estudio del estado del arte se desprende que los lenguajes más adecuados, teniendo en cuenta su solidez, aceptación por la comunidad investigadora, expresividad y adecuación a verificación de requisitos funcionales, son dos formalismos basados en redes de Petri: Wf–nets [2] y YAWL [8], en especial este último. Sin embargo, no resultan adecuados para formar parte de la capa 2 debido a sus limitaciones para el modelado de la perspectiva de datos de los procesos de negocio.

Por ello, hemos decidido diseñar un nuevo sistema formal para formar parte de esta capa, llamado *Common Formal Model* (CFM). En el capítulo 4 se define este formalismo. En el capítulo 5 se analiza su expresividad, desde el punto de vista de los *patrones de workflow* [9], y se compara con la de otros lenguajes y formalismos. El objetivo es mostrar que su expresividad es suficiente para formar parte de la arquitectura. A continuación, en el capítulo 6, se muestra cómo definiciones de procesos CFM pueden ser verificados mediante distintas herramientas de *model checking*. Con el objetivo de mostrar el formalismo CFM desde un punto de vista más práctico, y demostrar su viabilidad, en el capítulo 7 se analiza la representación de procesos BPEL4WS mediante este formalismo.

3.4. Conclusiones

En este capítulo se ha planteado una de las contribuciones de esta tesis doctoral: una arquitectura abierta, modular y extensible para la verificación de procesos de negocio, basada en desconectar los lenguajes de definición de procesos de negocio de las herramientas de verificación mediante la incorporación de un sistema formal intermedio. Esta arquitectura resuelve los problemas encontrados en el análisis realizado a otros trabajos relacionados, aunque condicionado a que el formalismo intermedio cumpla una serie de requisitos que se plantean. En los próximos capítulos se diseña, desarrolla y analiza este formalismo, tomando como referencia tanto la arquitectura planteada en este capítulo como los requisitos que esta impone al formalismo escogido.

Capítulo 4

El Modelo Formal Común

Many nonformalists seem to believe that formal methods are merely
an academic exercise – a form of mental masturbation
that has no relation to real–world problems.

(J. P. Bowen y M. G. Hinchey,
"Seven More Myths of Formal Methods",
IEEE Software, 12(4), pág. 34–41, Julio de 1995)

El componente principal de la arquitectura presentada en el capítulo 3 es el *Modelo Formal Común* de la capa 2. El objetivo de este formalismo es ejercer como formalismo intermedio para la aplicación de herramientas de verificación a definiciones de procesos de negocio y composiciones de servicios Web. En el capítulo 3 se establecen los requisitos que debe cumplir dicho formalismo, pero no se propone ninguno. En este capítulo se propone un sistema formal concreto, llamado CFM (*Modelo Formal Común*), para la capa 2 de la arquitectura.

La definición del sistema formal se realiza en tres capas. Cada una de estas capas se desarrolla por separado en los apartados 4.1, 4.2 y 4.3 de este capítulo:

1. *Sistema formal básico*: se define el formalismo básico sobre el cual se construye el *Modelo Formal Común*, basado en el formalismo de diagramas de estado–transición etiquetados.

2. *Recubrimiento de alto nivel*: se introduce un recubrimiento al sistema formal básico con conceptos de más alto nivel, cuya semántica está más cercana a la de los procesos de negocio. Cada uno de estos conceptos se define en términos del sistema formal básico.

3. *Notación*: se define una notación para especificar procesos mediante los conceptos definidos en las dos capas anteriores.

La definición del sistema formal presentada en estos apartados es considerablemente abstracta, con escasas referencias a conceptos del dominio de los procesos de negocio. En el apartado 4.4 se concreta la aplicación de este formalismo a la definición de procesos de negocio. Para ello, se proponen unas líneas generales acerca de cómo aplicar el formalismo a la definición de procesos de negocio mediante un enfoque basado en actividad. Estas líneas generales también permiten mostrar, aunque informalmente, la capacidad expresiva del formalismo y su adecuación al dominio de los procesos de negocio.

El sistema formal debe ser capaz de representar no sólo la definición de un proceso de negocio, sino también su especificación, entendida como el conjunto de propiedades que se desea demostrar que son ciertas en la definición. Se profundiza en este aspecto del sistema en el capítulo 6.

En el apartado 4.5 se analiza el formalismo CFM desde el punto de vista de sus limitaciones. El objetivo de este análisis es comprender qué limitaciones presenta el formalismo y cómo influyen en su capacidad para el modelado de procesos de negocio, así como facilitar futuras mejoras del mismo.

Finalmente, se exponen las principales conclusiones acerca del formalismo presentado en este capítulo.

4.1. Sistema formal básico

En este apartado se define formalmente la base del formalismo CFM. Se toma como partida el formalismo de los diagramas de estado–transición etiquetados, por ser este un formalismo utilizado habitualmente por técnicas de verificación, como es el caso de *model checking*. Los conceptos básicos que definen este sistema formal son el de atributo, estado, transición, camino y proceso.

4.1.1. Atributos y estado

Se define un *atributo* como un elemento que, en cada instante de tiempo, toma un valor dentro del conjunto de valores que determinan su rango. El rango de valores de un atributo a, que debe ser finito, se denota como V_a. Para un proceso dado, el conjunto finito de sus atributos se denota como A. Se define $V = \bigcup_{a \in A} V_a$ como la unión de los conjuntos de valores posibles para todos los atributos del proceso.

El *estado* del proceso en un instante dado viene dado por el valor de todos sus atributos en dicho instante.

Definición 1. *Se define* estado *como una aplicación* $s : A \rightarrow V$ *que verifique para todo* $a \in A$ *que* $s(a) \in V_a$.

Esto es, el estado es una aplicación que relaciona cada uno de los atributos con su valor. Dado un conjunto de atributos A, el conjunto de todos los estados se denota como S. Es trivial demostrar que, dado que A y V son finitos, el conjunto S es también finito.

4.1.2. Transiciones

El proceso evoluciona de un estado al siguiente a través de transiciones. Una *transición* es una relación entre un estado origen y un estado destino alcanzable directamente a partir

de este.

Definición 2. *Se define una* transición *como un par* (s_1, s_2) *perteneciente al producto cartesiano* $S \times S$.

Por tanto, el conjunto de todas las transiciones de un determinado proceso, denotado como T, está contenido en $\mathbb{P}(S \times S)$. Esto no es más que la propia definición de *relación binaria*. Entonces se puede representar el conjunto T de todas las transiciones como una relación binaria R sobre el dominio S. Para cada par de estados $s_1, s_2 \in S$ tenemos que $(s_1, s_2) \in T$ es equivalente a $s_1 R s_2$.

El concepto de transición permite definir el comportamiento dinámico del proceso. Dado un proceso en el estado $s \in S$ y el conjunto $S' = \{s' \in S \ / \ sRs'\}$, entonces para todo $s' \in S'$ el proceso puede evolucionar desde el estado s al estado s'. Dado un estado s, si no existe ningún estado $s' \in S$ tal que sRs', entonces se dice que s es un *estado final* del proceso. $S_f \subseteq S$ denota el conjunto de todos los estados finales.

4.1.3. Caminos y ejecuciones

Un *camino* define una posible evolución de los estados de un proceso o, lo que es lo mismo, una posible secuencia de transiciones activadas en el proceso.

Definición 3. *Se define un* camino *como una secuencia de estados* $e = \langle s_0, s_1, \ldots, s_{n-1} \rangle$ *tal que para todo* $0 \leq i < n - 1$ *se verifica* $s_i R s_{i+1}$.

Dado un camino $e = \langle s_0, s_1, \ldots, s_{n-1} \rangle$, se denota como e^i al estado i–ésimo de e. Se denota al estado final de e como $\text{final}(e) = e^{n-1}$. Se denota el número de estados del camino como $\#e = n$. Cometiendo un pequeño abuso de notación, el hecho de que un estado s aparezca en un camino e se denota como $s \in e$.

Definición 4. *Dado un conjunto de estados iniciales* S_0 *se define una* ejecución del proceso *como un camino* e *tal que* $e^0 \in S_0$ *y* $\text{final}(e) \in S_f$.

Por tanto, una ejecución es una posible evolución del proceso, desde un estado inicial hasta un estado final, de entre todas las posibles evoluciones conformes con la especificación del mismo.

4.1.4. Proceso

En función de las definiciones anteriores, se puede definir el concepto de proceso como el conjunto de todas las posibles ejecuciones conformes a su especificación.

Definición 5. *Dados un conjunto de estados* S*, una relación binaria* R *en el dominio de* S *y un conjunto de estados iniciales* S_0*, se define un* proceso P *como el conjunto de todas las posibles ejecuciones.*

Por tanto, el hecho de que una determinada ejecución e pertenezca a un proceso P se puede denotar mediante la notación de pertenencia a un conjunto: $e \in P$.

4.1.5. Transiciones funcionales

La definición explícita de las transiciones es asequible para procesos pequeños, pero no resulta práctica para procesos de un tamaño más realista. Esto se debe a que obliga a definir explícitamente todos los estados del proceso. Salvo en procesos extremadamente sencillos, esto no resulta práctico. En este formalismo se propone una representación implícita de estados y transiciones.

En lugar de representar los estados aisladamente, se pueden representar como conjuntos de estados que verifican una determinada propiedad. Por ejemplo, si en un proceso dado es necesario representar todos los estados tales que determinado documento haya sido revisado, resulta más adecuado definir este conjunto enunciando la propiedad que enumerando, uno a uno, todos sus estados. Por otra parte, en lugar de representar las transiciones aisladamente, pueden ser representadas también de forma simbólica en forma de *transiciones funcionales*.

Definición 6. *Se define una* transición funcional *como una función* $f : S_1 \rightarrow S$ *tal que* $S_1 \subseteq S$ *y para todo* $s \in S_1$ *se verifica* $sRf(s)$.

La función f define un conjunto de transiciones, tantas como elementos contenga S_1. Cada transición se define como el par formado por un estado $s \in S_1$ y el estado $f(s)$. Una transición funcional es, por tanto, una forma abreviada de representar múltiples transiciones entre estados.

Para representar todas las transiciones del proceso no es suficiente, en general, con una única transición funcional. El conjunto de todas las transiciones funcionales que definen un proceso determinado se denota como F. Es importante reseñar que la representación de las transiciones implícitamente, mediante un conjunto de transiciones funcionales, es equivalente a una representación explícita, mediante la relación binaria R. Esto es, dado el conjunto F se puede obtener una relación binaria R que define exactamente las mismas transiciones. Por tanto, todas las definiciones anteriores son válidas igualmente si se definen las transiciones mediante transiciones funcionales.

4.2. Recubrimiento de alto nivel

El sistema formal básico introducido en el apartado anterior permite definir procesos mediante conceptos de muy bajo nivel. Con el fin de facilitar la definición de procesos, el recubrimiento de alto nivel enriquece el sistema con los siguientes conceptos: *tipo de datos básico*, *entidad*, *tipo de entidad* y *actividad*. Por otra parte, el recubrimiento concreta cómo se puede definir una transición funcional utilizando únicamente dos predicados.

4.2.1. Conceptos añadidos al formalismo básico

El sistema formal definido anteriormente especifica que un atributo almacena un valor, que puede cambiar en el tiempo. El valor debe pertenecer al conjunto finito de posibles valores para dicho atributo. Con el fin de facilitar la especificación del conjunto de valores válidos de cada atributo, se definen los siguientes tipos de datos simples:

- *Booleano*: toma un valor entre dos posibles, *cierto* o *falso*.

- *Entero*: número entero de 16 bits. El conjunto de valores que puede tomar un atributo de este tipo es el intervalo $[-2^{15}, 2^{15}-1]$.
- *Enumerado*: se define el conjunto de valores posibles para el atributo como una enumeración finita de símbolos.
- *Abstracto*: atributos cuyo valor puede ser abstraído, esto es, ignorado a efectos de verificación. A pesar de que no sea relevante para la verificación, se introduce para resaltar, cuando resulte conveniente, la existencia de un atributo que se ignora en la representación CFM.

Los tipos de datos básicos anteriores se pueden agrupar lógicamente, dando lugar a tipos de datos compuestos llamados *tipos de entidad*. Un tipo de entidad se define como un conjunto de uno o más campos, donde cada campo se define mediante un identificador y un tipo de datos básico. Si se hace una analogía con el lenguaje de programación C, un tipo de entidad es como una estructura (*struct*), con la restricción de que en el tipo de entidad todos los campos deben tener un tipo de datos simple.

Una *entidad* se define como una instancia de un tipo de entidad. Siguiendo la analogía anterior, es como una variable en C cuyo tipo de datos se define mediante una estructura. Cuando se declara una entidad, en el sistema formal subyacente se están introduciendo tantos atributos como campos tenga su tipo de entidad. Cada uno de estos atributos es identificable unívocamente con el par nombre de entidad–nombre de campo.

En el momento de declarar una entidad, se le pueden asignar uno o más valores iniciales. Cada valor inicial consiste en un conjunto de valores para todos sus campos. Si se especifica más de un valor inicial, entonces se selecciona uno de ellos de forma no determinista en el momento de instanciación de dicha entidad. La herramienta de verificación debe, en este caso, desarrollar los caminos de ejecución correspondientes a todos los valores posibles. Si no se especifica ningún valor inicial, se toma, para cada campo, el valor inicial por defecto del tipo de datos correspondiente. Estos valores por defecto son *falso* para *boolean*, 0 para *integer* y el primer valor declarado en campos de rango enumerado.

El estado de un proceso de negocio evoluciona como resultado de la ejecución de sus actividades. Dado que en el sistema formal básico las transiciones son los elementos capaces de provocar cambios en el estado del sistema, resulta lógico modelar el comportamiento de una instancia de una actividad como una secuencia de transiciones que modelan los cambios que esta genera en el sistema como consecuencia de su ejecución. Por tanto, se define una *actividad* como un conjunto de una o más transiciones funcionales. Estas transiciones funcionales modelan el comportamiento de la actividad. En el apartado 4.4 se muestra cómo un conjunto de transiciones funcionales pueden modelar la ejecución de una actividad de un proceso de negocio.

4.2.2. Definición de transiciones funcionales

Como se ha expuesto anteriormente, una transición funcional representa a un conjunto de transiciones. En general, las transiciones de una misma transición funcional modelan un mismo comportamiento, donde cada transición representa dicho comportamiento aplicado a un estado inicial distinto. En la definición 6 se introduce el concepto de transición funcional, y se especifica que viene determinado por un subconjunto de estados S_1 y una función f.

En este apartado se define cómo el subconjunto S_1 se representa mediante una expresión de *dominio* y la función f se define mediante una expresión de *acción*. Con ello, se presente obtener una representación compacta de las transiciones del sistema como alternativa a la definición explícita de cada uno de ellos.

El *dominio* de una transición funcional (S_1) es el conjunto de estados a los cuales dicha transición funcional es aplicable. Se especifica dicho dominio mediante una función *Booleana* $d : S \rightarrow \mathbb{B}$ tal que se evalúa como 1 lógico para los estados pertenecientes a S_1 y como 0 lógico para el resto de estados.

La *acción* es un predicado que define cómo se construye el estado destino de cada una de las transiciones en función de su estado origen. Este predicado es único para todas las transiciones pertenecientes a la misma transición funcional, dado que se pretende modelar un comportamiento común, desde un punto de vista de alto nivel. Este predicado especifica la función f de la definición 6.

Para simplificar la notación, se denota el valor de un atributo a en el estado inicial s de la transición como a, y su valor en su estado final s' como a'. Esto supone un pequeño abuso de notación, dado que, estrictamente, deberían denotarse como $s(a)$ y $s'(a)$. Por otra parte, se entiende que todo atributo cuyo valor en el estado destino no se especifique en el predicado de acción conserva el mismo valor que en el estado origen.

Ejemplo

Para aclarar cómo se especifica una transición funcional, se propone el siguiente ejemplo. Supongamos un sistema en que se modela la transferencia de dinero entre dos cuentas como una transición funcional. El sistema define, para cada cuenta, un atributo entero indicando su saldo en euros, c_1 y c_2. Adicionalmente, el proceso define otros dos atributos, también enteros, indicando la cantidad a transferir (t) y el identificador de la transferencia (n). La transferencia sólo se puede realizar si hay fondos suficientes en la cuenta origen. Nótese que se están haciendo dos simplificaciones para reducir la complejidad del ejemplo: sólo se permite operar con múltiplos enteros de euro y no se tiene en cuenta un posible desbordamiento en la cuenta destino de la transferencia.

Si se asocia el atributo c_1 a la cuenta origen y el atributo c_2 a la cuenta destino, el dominio de esta transición funcional es el conjunto de estados D en los cuales la cuenta origen dispone de dinero suficiente para realizar la transferencia, y el valor de la transferencia no es negativo. Este conjunto de estados se define como:

$$D = \{s \in S \ / \ s(c_1) \geqslant s(t) \ \ y \ \ s(t) \geqslant 0\} \tag{4.1}$$

La función *Booleana* que especifica este dominio sería la siguiente:

$$d(s) = \begin{cases} 1, & s(c_1) \geqslant s(t) \ \ y \ \ s(t) \geqslant 0 \\ 0, & s(c_1) < s(t) \ \ o \ \ s(t) < 0 \end{cases} \tag{4.2}$$

Desde el punto de vista del sistema formal presentado en este capítulo, se especificará de forma abreviada el dominio de la transición funcional mediante el siguiente predicado *Booleano*:

$$(c_1 \geqslant t) \wedge (t \geqslant 0) \tag{4.3}$$

La función f que define la acción de la transición funcional sería la siguiente:

$$\begin{array}{rccc} f: & D & \rightarrow & S \\ & s & \rightarrow & s' \end{array} \tag{4.4}$$

$$\begin{cases} s'(c_1) & = & s(c_1) - s(t) \\ s'(c_2) & = & s(c_2) + s(t) \\ s'(t) & = & 0 \\ s'(n) & = & s(n) \end{cases} \tag{4.5}$$

Utilizando la notación abreviada, se denota la acción de esta transición funcional mediante el siguiente predicado (si no se especifica un atributo, en este caso n, se supone que no cambia su valor):

$$(c_1' = c_1 - t) \wedge (c_2' = c_2 + t) \wedge (t' = 0) \tag{4.6}$$

Si se analiza la proposición que define el dominio de la transición funcional, se puede deducir el número de elementos de que consta dicho dominio y, por tanto, el número de transiciones de bajo nivel definidas por esta transición funcional. Si se supone que los números enteros pueden tomar un valor en el intervalo $[-N, N)$, el número de transiciones vendría dado por el producto del número de posibles valores de c_2 y n, que es $2N \cdot 2N$, por el número de posibles pares de c_1 y t tales que $t \geqslant 0$ y $c_1 \geqslant t$:

$$2N \cdot 2N \sum_{i=0}^{N-1} (N - i) = 4N^2 \left(N^2 - \frac{N(N-1)}{2} \right) = 2N^4 - 2N^3 \tag{4.7}$$

Aunque esto no es muy realista, supongamos que los enteros se representasen con sólo 16 bits. Entonces $N = 2^{15}$ y el número total de transiciones definidas por esta transición funcional es de $2^{61} - 2^{46} \approx 2,31 \cdot 10^{18}$. Por otra parte, si además de n hubiese más atributos cuyo valor no influyese en esta transición funcional concreta, se estaría multiplicando este número total de transiciones por el número de combinaciones posibles en los valores del resto de los atributos.

Este ejemplo justifica por qué es conveniente, y en este caso necesario, recurrir a la definición simbólica de transiciones. Sería inviable definir explícitamente los más de 2 millones de billones de transiciones, que por otra parte se pueden definir mediante dos predicados muy simples:

- Dominio: $(c_1 \geqslant t) \wedge (t \geqslant 0)$
- Acción: $(c_1' = c_1 - t) \wedge (c_2' = c_2 + t) \wedge (t' = 0)$

4.3. Definición de una notación para el *Modelo Formal Común*

Una vez definido el sistema formal en sí mismo, en este apartado se presenta la notación desarrollada para representar procesos de negocio mediante el *Modelo Formal Común*. Se define la notación mediante la variante de EBNF (*Extended Backus Naus Form*) definida en la sección 6 de la especificación de XML [129].

4.3.1. Sintaxis y gramática

En primer lugar se definen las reglas auxiliares utilizadas en el resto de las definiciones (caracteres alfabéticos, caracteres numéricos, espacios, identificadores y referencias a atributos).

```
Alpha       ::= [#x41-#x5A] | [#x61-#x7A]   ; A-Z / a-z
Digit       ::= [#x30-#x39]                 ; 0 - 9
S           ::= (#x20 | #x9 | #xD | #xA)+   ; space
Id          ::= (Alpha | Digit | '_')+      ; identifier
VarRef      ::= EntityId "." FieldId
```

Un proceso es el elemento principal de un modelo CFM. El proceso tiene asociado un identificador. Internamente contiene definiciones de tipos de entidades, declaraciones de entidades, definiciones de actividades y la especificación.

```
Process     ::= S? "process" S ProcessId S? "{" S?
                EntType* Entity* Activity*
                Spec* "}" S?
ProcessId   ::= Id
```

Una definición de un tipo de entidad se compone de un identificador y uno o más campos. Cada campo se define mediante un identificador y un tipo de datos. El tipo de datos puede ser de rango enumerado, entero, *Booleano* o abstracto.

```
EntType     ::= "enttype" S EntTypeId S? "{" S?
                Field+ "}" S?
Field       ::= FieldId S? ":" S? FieldType S? ";" S?
FieldType   ::= "enum" S? "(" S? Id (S Id)* S? ")"
                | "integer" | "boolean" | "abstract"
EntTypeId   ::= Id
FieldId     ::= Id
```

Una entidad es una instancia de un tipo de entidad. Su declaración consiste en su propio identificador y el identificador del tipo de entidad.

```
Entity      ::= "entity" S EntityId S? ":" S?
                EntTypeId S? Initials? ";" S?
EntityId    ::= Id
```

Los valores iniciales para una entidad, que son opcionales, se especifican a continuación de la declaración de la propia entidad. Se puede proporcionar más de un valor, y cada uno de ellos se especifica como una combinación de valores iniciales para todos los campos de la entidad.

```
Value       ::= Integer | Boolean | Id
Initials    ::= "=" S? "{" S? Initial+ "}" S?
Initial     ::= "{" S? Value S? ("," S? Value S?)* "}" S?
```

Una actividad tiene asociado un identificador. Se define como un conjunto de una o más transiciones funcionales.

```
Activity   ::= "activity" S ActivityId S? "{" S?
               Transition+ "}" S?
ActivityId ::= Id
```

Un transición funcional se define mediante un identificador, su dominio (expresión que define el conjunto de estados a los cuales la transición es aplicable) y su acción (expresión que define cómo la transición modifica su estado origen).

```
Transition ::= "transition" S TransitId S? Progress?
               S? "{" S? Domain S? Action S? "}" S?
Domain     ::= "domain" S? ":" S? "{" S?
               BooleanExp S? "}"
Action     ::= "action" S? ":" S? "{" S?
               ActionExp S? "}"
Progress   ::= "[" S? "progress" S? "]"
TransitId  ::= Id
```

En las reglas anteriores es necesario definir una notación para las expresiones. En primer lugar, se definen las expresiones con números enteros, mediante los operadores de aritmética entera *suma* (+), *resta* (-), *producto* (*), *división* (/) y *módulo* (%).

```
Integer    ::= ("-" S?)? Digit+
IntegerExp ::= Integer
               | VarRef
               | (IntegerExp S? "+" S? IntegerExp)
               | (IntegerExp S? "-" S? IntegerExp)
               | (IntegerExp S? "*" S? IntegerExp)
               | (IntegerExp S? "/" S? IntegerExp)
               | (IntegerExp S? "%" S? IntegerExp)
               | ("(" S? IntegerExp S? ")")
```

Una expresión enumerada se construye conforme a la siguiente sintaxis. Se dice que dos expresiones enumeradas son compatibles si ambas son referencias a variables del mismo tipo enumerado o una es referencia a variable y la otra un identificador correspondiente a uno de los valores enumerados declarados para dicha variable.

```
EnumExp    ::= VarRef        ; enum variable
               | Id          ; value of enum variable
```

Una expresión *Booleana* se define mediante los operadores lógicos *y* (&), *o* (|) y *no* (!), y los operadores de comparación *menor–que* (<), *menor–o–igual–que* (<=), *igual–que* (=), *mayor–o–igual–que* (>=), *mayor–que* (>) y *distinto-que* (!=).

```
Boolean    ::= "true" | "false"
BooleanExp ::= Boolean
             | VarRef
             | (BooleanExp S? "&" S? BooleanExp)
             | (BooleanExp S? "|" S? BooleanExp)
             | (! S? BooleanExp)
             | (BooleanExp S? "=" S? BooleanExp)
             | (BooleanExp S? "!=" S? BooleanExp)
             | (EnumExp S? "=" S? EnumExp)
             | (EnumExp S? "!=" S? EnumExp)
             | (IntegerExp S? "=" S? IntegerExp)
             | (IntegerExp S? "!=" S? IntegerExp)
             | (IntegerExp S? "<" S? IntegerExp)
             | (IntegerExp S? "<=" S? IntegerExp)
             | (IntegerExp S? ">" S? IntegerExp)
             | (IntegerExp S? ">=" S? IntegerExp)
             | ("(" S? BooleanExp S? ")")
```

Las expresiones de acción definen cambios en el estado del proceso, esto es, en el valor de sus variables. Se considera que aquellas variables no referenciadas en la expresión de acción tienen el mismo valor en el estado inicial y final de la transición.

```
ActionExp  ::= Assign S? ("&" S? Assign S?)*
Assign     ::= (VarRef S? "=" S? BooleanExp)
             | (VarRef S? "=" S? IntegerExp)
             | (VarRef S? "=" S? EnumExp)
             | VarRef
             | ("!" S? VarRef)
```

En una asignación, una referencia a una variable *Booleana* equivale a una asignación de valor *true* a dicha variable y una referencia a una variable precedida por el operador de negación equivale a una asignación de valor *false*.

4.3.2. Restricciones adicionales

Las reglas anteriores establecen la sintaxis y gramática de la notación de CFM. Se dice que una definición de un proceso CFM es válida si verifica las reglas anteriores y además es conforme a las siguientes restricciones:

1. En un mismo proceso no puede haber dos o más declaraciones de tipos de entidades con un mismo identificador.

2. En un mismo proceso no puede haber dos o más declaraciones de entidades con un mismo identificador.

3. En un mismo proceso no puede haber dos o más declaraciones de actividades con un mismo identificador.

4. En una misma actividad no puede haber dos o más declaraciones de transiciones funcionales con un mismo identificador.

5. En una producción `VarRef` el identificador izquierdo debe coincidir con el de alguna entidad declarada en el proceso. El identificador derecho debe coincidir con el identificador de un campo del tipo de entidad al que pertenezca dicha entidad.

6. Toda producción `VarRef`, como expresión entera, debe corresponder a un campo declarado con tipo entero.

7. Toda producción `VarRef`, como expresión de rango enumerado, debe corresponder a un campo declarado con tipo enumerado.

8. Toda producción `VarRef`, como expresión booleana, debe corresponder a un campo declarado con tipo booleano.

9. En toda comparación (igualdad o desigualdad) de expresiones de rango enumerado ambas expresiones deben ser compatibles, conforme a la definición dada anteriormente.

10. En una producción `Assign` se debe cumplir que las referencias a variable de la primera, cuarta y quinta reglas sean de tipo *Booleano*, la de la segunda regla sea de tipo entero y la de la tercera regla sea de tipo enumerado (y la expresión enumerada de la derecha compatible con ella).

4.3.3. Especificaciones

En este apartado se completa la definición de la gramática de CFM definiendo la regla `Spec`, que permite definir una especificación, esto es, el conjunto de requisitos a verificar. Los distintos tipos de requisitos presentados en este apartado se describen en el capítulo 6.

Una definición CFM puede contener cualquier número de elementos `Spec`. Este elemento contiene internamente un conjunto de requisitos:

```
Spec       ::= "specification" S SpecId S? "{" S? Req* S? "}"
SpecId     ::= Id
```

Un requisito debe pertenecer a alguno de los tipos de requisitos definidos en CFM. Por tanto, puede ser una invariante, un objetivo, un pre–requisito de una transición funcional, un post–requisito de una transición funcional, un requisito de ejecutabilidad de una transición funcional, un requisito de ejecutabilidad de todas las transiciones funcionales o un requisito expresado mediante lógica temporal:

```
Req        ::= InvarReq | GoalReq | PreReq | PostReq
                | ExeReq | ExeAllReq | TLReq
ReqId      ::= Id
```

Se definen las invariantes y los objetivos mediante un predicado *Booleano*, de la siguiente forma:

```
InvarReq   ::= "invariant" S ReqId S? "{" S? BooleanExp
               S? "}" S?
GoalReq    ::= "goal" S ReqId S? "{" S? BooleanExp
               S? "}" S?
```

Los pre–requisitos y post–requisitos son aplicables a una transición funcional concreta. Por ello, su definición debe hacer referencia al identificador de la actividad y de la transición funcional:

```
PreReq     ::= "prereq" S ReqId S? "(" S? ActivityId S? "," S?
               TransitionId S? ")" S? "{" S? BooleanExp
               S? "}" S?
PostReq    ::= "postreq" S ReqId S? "(" S? ActivityId S? "," S?
               TransitionId S? ")" S? "{" S? BooleanExp
               S? "}" S?
```

Se puede especificar requisitos de ejecutabilidad de transiciones funcionales concretas. Por otra parte, con el fin de hacer más compacta la notación, se incluye un tipo de requisito que aplica a todas las transiciones funcionales definidas en el proceso:

```
ExeReq     ::= "exe" S ReqId S? "(" S? ActivityId S? "," S?
               TransitionId S? ")" S? ";" S?
ExeAllReq  ::= "exeall" S ReqId S? ";" S?
```

Por último, los requisitos expresados mediante lógica temporal deben indicar qué lógica temporal concreta utilizan.

```
TLReq      ::= "tl" S ReqId S? "(" S? TLId S? ")" S? "{"
               S? TLExp S? "}" S?
TLId       ::= "ltl" | "ctl"
TLExp      ::= LTLExp | CTLExp
```

En principio, se permite emplear las lógicas temporales LTL y CTL, por ser las más ampliamente extendidas en herramientas de verificación. Si en un futuro se desease permitir otro tipo de lógica, se pueden redefinir las reglas `TLId` y `TLReq`.

La gramática definida para estas expresiones de lógica temporal es la siguiente. Se toma como referencia para definir esta gramática la sintaxis de la herramienta NuSMV, especificada en su manual de usuario [26]. En primer lugar, se define la gramática de las expresiones CTL:

```
CTLExp     ::= BooleanExp
               | "(" S? CTLExp S? ")"
               | "!" S? CTLExp
               | CTLExp S? "&" S? CTLExp
               | CTLExp S? "|" S? CTLExp
               | CTLExp S? "xor" S? CTLExp
               | CTLExp S? "->" S? CTLExp
               | CTLExp S? "<->" S? CTLExp
```

```
               | "EG" S? CTLExp
               | "EX" S? CTLExp
               | "EF" S? CTLExp
               | "AG" S? CTLExp
               | "AX" S? CTLExp
               | "AF" S? CTLExp
               | "E" S? "[" S? CTLExp S? "U" S? CTLExp S? "]"
               | "A" S? "[" S? CTLExp S? "U" S? CTLExp S? "]"
```

Las expresiones LTL se definen de la siguiente forma:

```
LTLExp     ::= BooleanExp
               | "(" S? LTLExp S? ")"
               | "!" S? LTLExp
               | LTLExp S? "&" S? LTLExp
               | LTLExp S? "|" S? LTLExp
               | LTLExp S? "xor" S? LTLExp
               | LTLExp S? "->" S? LTLExp
               | LTLExp S? "<->" S? LTLExp
               | "X" S? LTLExp
               | "G" S? LTLExp
               | "F" S? LTLExp
               | LTLExp S? "U" S? LTLExp
               | LTLExp S? "V" S? LTLExp
```

Por otra parte, aplican las siguientes limitaciones en la definición de especificaciones:

1. No debe haber, en una misma especificación, más de un requisito con el mismo identificador.
2. No puede haber, en la misma definición de un proceso, más de una especificación con el mismo identificador.
3. En las reglas de `PreReq`, `PostReq` y `ExeReq`, la referencia al identificador de actividad debe coincidir con el de una de las actividades definidas para el proceso.
4. En las reglas de `PreReq`, `PostReq` y `ExeReq`, la referencia al identificador de transición funcional debe coincidir con el de una de las transiciones funcionales definidas para la actividad referenciada.

4.4. Líneas generales de representación de procesos de negocio

Una vez definido el *Modelo Formal Común*, se proponen unas líneas generales de representación, mediante este formalismo, de procesos de negocio definidos mediante lenguajes de definición de alto nivel (capa 3 de la arquitectura). Estas líneas generales se proponen fundamentalmente con dos objetivos. Por una parte, mostrar cómo este sistema formal es capaz de representar definiciones de procesos de negocio. Por otra parte, servir de referencia para el diseño de transformaciones desde lenguajes de capa 3 al formalismo de la capa 2 de la arquitectura.

4.4.1. Casos

Tal y como se define en el cuadro 2.2, un caso es una instancia concreta de un proceso, normalmente asociada a un cliente concreto. Para representar los casos de un proceso, en CFM se supone cada caso totalmente aislado de los demás. Se entiende que cada caso se ejecuta de forma independiente de los demás con sus propias instancias de los atributos. Un caso de un proceso puede acceder a sus propios atributos, pero no a los del resto de casos. Dado que esto puede suponer una limitación en la capacidad de modelado del formalismo CFM, se analiza en el apartado 4.5.3.

4.4.2. Objetos manipulados

Uno de los conceptos básicos del modelado basado en actividad es el de *objeto manipulado* o *entidad*. Como se indica en su definición (véase el cuadro 2.2), un objeto manipulado representa un documento, un objeto material o simplemente un conjunto de datos. Las distintas tareas del proceso operan sobre los objetos manipulados, ya sea modificándolos o simplemente consultando información.

Por ejemplo, una factura a emitir a un cliente se puede considerar un objeto manipulado. Algunas tareas del proceso pueden modificarla, por ejemplo, estableciendo los datos del cliente, el precio total, la fecha, etc. Otras tareas, por ejemplo las asociadas con contabilidad, pueden simplemente consultar el precio y la fecha para registrar la operación en los libros de contabilidad.

De cara a trabajar con un objeto manipulado en el proceso, es necesario modelar sus propiedades. Obviamente, no es necesario modelar todas sus propiedades, sino sólo aquellas que resulten relevantes desde el punto de vista del proceso. Por ejemplo, en el caso de una factura, es probable que a nivel de lógica del proceso sólo resulte relevante el precio total de la misma.

Desde el punto de vista del formalismo CFM, se puede representar cada una de estas propiedades relevantes mediante un atributo. Concretamente, se puede definir un *tipo de entidad* para especificar mediante *campos* estas propiedades del objeto. Cada instancia del objeto se modela entonces como una *entidad* de CFM.

Una de las dudas razonables que se plantea en este punto es que, dado que CFM sólo permite definir atributos *Booleanos*, enteros (limitados a un rango de valores finito) y enumerados, no es capaz de modelar de forma cómoda otros tipos de datos como, por ejemplo, una cadena de texto, un número arbitrariamente grande perteneciente al conjunto de los números enteros o un número real. En el apartado 4.5.1 se analiza en detalle esta limitación del sistema formal.

Ejemplo

En este apartado se ilustra con un ejemplo el trabajo con objetos manipulados. Supongamos que se desea modelar una factura en un proceso de compra de un bien o servicio. Supongamos que el diseñador del proceso identifica la siguiente información relevante en una factura:

- Identificador de cliente (permite recuperar los datos completos del mismo, incluido un

valor que indica la importancia del cliente para la empresa: muy importante, importante, normal).

- Fecha de emisión.
- Importe de la factura.
- Pago (indica si el pago ha sido realizado o no).

Una posible representación de la factura desde el punto de vista de la verificación del proceso podría ser la siguiente:

```
enttype factura_t {
    identificador: abstract;
    importe: integer;
    pago: boolean;
}
```

El importe, en este caso, se modela como entero. Sin embargo, dependiendo de las características del proceso, podría ser más conveniente declararlo como abstracto o abstraer su rango de valores a un conjunto enumerado (véase el apartado 4.5.1). Obviamente, un dato *Booleano* es suficiente para indicar si la factura está o no pagada.

Se ha decidido modelar el identificador como un dato abstracto, porque no resulta relevante en sí mismo para el flujo del proceso. Sin embargo, supongamos que en el proceso se toman decisiones con respecto a la importancia del cliente. *A priori*, parece necesario en este caso modelar explícitamente el identificador. Sin embargo, en realidad basta con modelar esta importancia con un nuevo campo. Este campo será enumerado con tres posibles valores:

```
enttype factura_t {
    identificador: abstract;
    importancia: enum(muy_importante, importante, normal);
    importe: integer;
    pago: boolean;
}
```

Finalmente, se pueden declarar cada uno de los objetos factura mediante una entidad. En este caso se declaran dos distintas:

```
entity factura1: factura_t;
entity factura2: factura_t;
```

4.4.3. Variables

Algunos lenguajes de definición de procesos permiten modelar explícitamente el concepto de *variable*. En CFM se puede representar una variable como una entidad, de la misma forma que se representa un objeto manipulado.

Un aspecto a resaltar es el hecho de que en CFM todas las entidades tienen ámbito global, en el sentido de que una entidad es accesible desde cualquier actividad y transición funcional del proceso. No es posible declarar una entidad de forma local a una actividad o conjunto de actividades. En el apartado 4.5.2 se analizan las limitaciones que esto puede suponer.

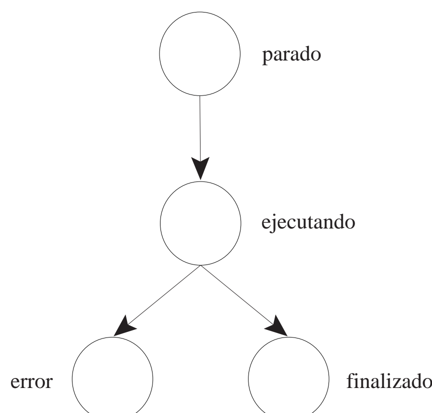

Figura 4.1: Ejemplo de un ciclo de vida sencillo de una actividad, con sólo cuatro estados posibles.

4.4.4. Actividades

La representación de las actividades del proceso es uno de los aspectos clave en la utilización del sistema formal común. Para modelar cada actividad, se propone utilizar un diagrama de estados simple que represente su ciclo de vida. La ejecución de la actividad se representa como una evolución por su ciclo de vida. Durante su ejecución, la actividad puede modificar el valor de atributos del proceso, e incluso controlar la ejecución de otras actividades. En los siguientes apartados se profundiza en estos aspectos.

Ciclo de vida

El ciclo de vida de una actividad se puede representar como un diagrama de estados. Debe representar los distintos estados que puede atravesar una actividad durante su ejecución, desde antes de su comienzo hasta su finalización.

En la figura 4.1 se representa, a modo de ejemplo, un ciclo de vida muy simple de una actividad con cuatro estados. El primer estado (`parado`) representa al proceso antes de comenzar su ejecución. El segundo (`ejecutando`), representa al proceso en ejecución. El estado `finalizado` representa que la actividad ha finalizado correctamente su ejecución. El estado `error` representa que la actividad ha finalizado su ejecución con errores.

El estado de cada actividad dentro de su propio ciclo de vida forma parte del estado global del proceso. Por tanto, se crea un nuevo tipo de entidad para modelar este ciclo de vida, y una entidad de este tipo para cada una de las actividades. En el ejemplo anterior:

```
enttype ciclo_vida_t {
    estado: enum(parado, ejecutando, finalizado, error);
}
entity actividad_1: ciclo_vida_t;
```

En un mismo proceso, distintas actividades pueden ser modeladas mediante un ciclo de vida distinto. Como se verá más adelante para el caso de BPEL4WS, esto puede resultar muy útil en casos en que distintos tipos de actividades tengan comportamientos distintos.

Transiciones

Dado que la evolución de una actividad por su ciclo de vida supone un cambio en el estado del proceso, dicha evolución debe ser modelada como transiciones funcionales. El propio diagrama de estados de la actividad impone restricciones en las precondiciones (dominio) de cada una de estas transiciones funcionales.

Siguiendo con el ejemplo de ciclo de vida planteado en el apartado anterior, la transición funcional que modela el inicio en la ejecución de la actividad sería la siguiente:

```
transition iniciar {
    domain: {actividad_1.estado = parado}
    action: {actividad_1.estado = ejecutando}
}
```

Esta transición funcional establece, como condición para iniciar la ejecución de la actividad, que dicha actividad no haya comenzado todavía su ejecución. El inicio en su ejecución se modela modificando su estado a `ejecutando`. De un modo semejante se pueden modelar el resto de las transiciones funcionales de la actividad. Sin embargo, cabe destacar que existe una diferencia: de un mismo estado (`ejecutando`) parten dos transiciones funcionales alternativas. En próximos apartados se profundiza en el modelado de este tipo de estructuras.

```
transition finalizar {
    domain: {actividad_1.estado = ejecutando}
    action: {actividad_1.estado = finalizado}
}
transition error {
    domain: {actividad_1.estado = ejecutando}
    action: {actividad_1.estado = error}
}
```

Acceso a objetos manipulados

Una actividad de un proceso de negocio es capaz de obtener información de objetos manipulados, así como modificarlos. Por tanto, es necesario que en CFM una actividad pueda leer o modificar el valor de los atributos que representen a los objetos manipulados.

Por ejemplo, supongamos una actividad que gestione el envío por correo de un producto, conforme al ciclo de vida planteado anteriormente. Por política de la empresa, esta actividad sólo puede comenzar cuando la factura (modelada en el apartado 4.4.2) haya sido pagada. Supongamos que, tras un tiempo de procesado no conocido *a priori*, se da en esta actividad la orden de envío estableciendo valor *cierto* en una variable *Booleana* llamada `envio.permitido`. Con esto finaliza la ejecución de la actividad. Se podría modelar esto con las siguientes dos transiciones funcionales:

```
activity enviar {
    transition iniciar {
       domain: {enviar.estado = parado & factura.pago}
       action: {enviar.estado = ejecutando}
    }
    transition finalizar {
       domain: {enviar.estado = ejecutando}
```

```
        action: {enviar.estado = finalizado & envio.permitido}
    }
}
```

En el ejemplo anterior se puede observar que el solo hecho de activarse la primera transición funcional cambia el estado del proceso de tal forma que la segunda transición funcional puede ejecutarse inmediatamente. Si, como parece, la ejecución de ambas transiciones funcionales se produce de forma consecutiva, cabe preguntarse por qué no se modelan como una única transición funcional. La respuesta a esto es que en realidad, tal y como se comenta en el enunciado del ejemplo, puede pasar un tiempo desde que se paga la factura hasta que se ordene su envío. En el modelo se refleja este tiempo mediante el estado intermedio de la actividad (`ejecutando`). Nótese que en este sistema formal no se modela el tiempo transcurrido entre transiciones funcionales, sino las posibles secuencias de ejecución de las mismas. El hecho de que la segunda transición pueda ser ejecutada inmediatamente no modela, en absoluto, que esto ocurra en cero unidades de tiempo. Es más, mientras la actividad esté en estado `ejecutando`, podrían estarse ejecutando transiciones funcionales correspondientes a otras actividades concurrentes.

4.4.5. Control de flujo

Si la ejecución de las actividades no estuviese sujeta a restricciones, todas ellas podrían ser ejecutadas en cualquier orden y con cualquier nivel de paralelismo. Como se ha visto en el apartado 2.2.1, una de las perspectivas de un proceso de negocio que se modela en los lenguajes de definición de procesos de negocio es el *control de flujo*, que establece restricciones a la ejecución libre de las actividades. En este apartado se expone una forma de modelado de esta perspectiva en CFM.

Se puede modelar esta perspectiva de forma explícita o implícita. Una restricción es explícita cuando en ella se indica de forma explícita la o las actividades con las cuales existe la dependencia. Una restricción es implícita cuando no modela explícitamente con qué actividad o actividades existe la dependencia, sino que establece las restricciones a través de objetos manipulados.

Nótese que una restricción explícita también puede crear restricciones implícitas: si una actividad A depende de otra actividad B, y a su vez B depende de una actividad C, el hecho de establecer una restricción explícita de dependencia de A con respecto a B implica que exista una restricción de A con respecto a C.

En uno de los ejemplos planteados en el apartado 4.4.4 se establece una restricción implícita a la ejecución de la actividad `enviar`: sólo puede ser ejecutada si la factura está pagada, lo que impone una restricción de que alguna actividad que marque el pago de la factura haya sido ejecutada con antelación.

En este apartado se profundiza en el establecimiento de restricciones explícitas. Aunque existen varias alternativas para el establecimiento de restricciones en la ejecución de actividades, se ha optado por escoger una de ellas, que se expone a continuación. Posteriormente se analizarán otras alternativas.

Las restricciones impuestas a una actividad pueden ser establecidas en distintos estados dentro del ciclo de vida de la actividad. Por tanto, no afectan sólo a la ejecución de la actividad en global, sino también la ejecución de sus transiciones funcionales de forma

individual. La alternativa propuesta en este trabajo consiste en establecer, en el dominio de cada transición funcional objeto de las dependencias, las restricciones mediante referencias a los atributos que controlan el ciclo de vida de las actividades con las cuales existan dichas dependencias. Si una restricción controla el inicio de la ejecución de la actividad, entonces las transiciones funcionales objeto de las restricciones serán aquellas que modelen el inicio de su ejecución. Si controlase cuándo puede finalizar la ejecución de una actividad, entonces las transiciones funcionales afectadas serían aquellas que modelen la finalización de su ejecución. Por supuesto, aunque es un caso menos habitual, nada impide que se establezcan restricciones a transiciones funcionales intermedias en la ejecución de una actividad.

A continuación se plantea un ejemplo sencillo que ilustra lo explicado anteriormente. En los próximos apartados se plantea el modelado de los patrones de control flujo más habituales, mediante la aplicación de las restricciones oportunas en cada caso.

Ejemplo

Continuando con uno de los ejemplos del apartado 4.4.4, supongamos que se elimina la restricción implícita con respecto al pago de la factura, y se modela explícitamente, de tal forma que la actividad `enviar` sólo pueda comenzar una vez haya finalizado la actividad `gestionar_cobro`. Por otra parte, se establece una restricción explícita intermedia que exige que haya finalizado la actividad `extraer_de_almacen` antes de activar el envío y una restricción implícita al final que exige que la actividad sólo puede finalizar cuando se reciba notificación de entrega. La declaración de la actividad sería la siguiente:

```
activity enviar {
    transition iniciar {
       domain: {enviar.estado = parado
                & gestionar_cobro.estado = finalizado}
       action: {enviar.estado = ejecutando}
    }
    transition esperar_almacen {
       domain: {enviar.estado = ejecutando
                & !envio.permitido
                & extraer_de_almacen.estado = finalizado}
       action: {envio.permitido}
    }
    transition finalizar {
       domain: {enviar.estado = ejecutando
                & envio.recibido}
       domain: {enviar.estado = finalizado}
    }
}
```

Nótese que en este caso, aunque se sigue modelando la ejecución de la actividad mediante el ciclo de vida sencillo introducido a modo de ejemplo en el apartado 4.4.4, en realidad se están definiendo de forma implícita dos sub–estados para el estado `ejecutando` mediante el objeto manipulado que representa los datos del envío del producto.

Otras alternativas

La alternativa expuesta anteriormente no es la única posible para establecer restricciones. Otro posible modelo consiste en modelar un nuevo estado *permitido* en el ciclo de vida de las actividades. Si se quisiese modelar la restricción de que la actividad A no puede ser ejecutada hasta que no finalizase la actividad B, se podría añadir a la acción de la transición funcional que modela la finalización de B una asignación que cambiase el estado de A a *permitido*. La principal desventaja de este modelo está en que no es fácil representar restricciones con respecto a más de una actividad. Por otra parte, las restricciones que aplican a una actividad estarían distribuidas en las declaraciones de otras actividades, haciendo considerablemente más difícil su detección en una inspección visual. Es más, los modelos serían más complejos debido a la adición de un nuevo estado al ciclo de vida de las actividades.

4.4.6. Ejecución secuencial de actividades

La ejecución secuencial de actividades es fácilmente controlable mediante el mecanismo de control de flujo expuesto anteriormente. A cada actividad de la secuencia se le establece una restricción que impida el comienzo de su ejecución antes de que finalice la actividad que la preceda en la secuencia.

El siguiente ejemplo muestra la ejecución en secuencia de varias actividades. Por simplicidad, sólo se muestra la transición funcional de inicio de cada actividad:

```
activity primera {
    transition iniciar {
       domain: {primera.estado = parado}
       action: {primera.estado = ejecutando}
    }
    ...
}
activity segunda {
    transition iniciar {
       domain: {segunda.estado = parado
                & primera.estado = finalizado}
       action: {segunda.estado = ejecutando}
    }
    ...
}
activity tercera {
    transition iniciar {
       domain: {tercera.estado = parado
                & segunda.estado = finalizado}
       action: {tercera.estado = ejecutando}
    }
    ...
}
```

4.4.7. Ejecución concurrente de actividades

El modelado de la ejecución concurrente de actividades es inmediato: las actividades que no tengan ningún tipo de dependencias entre sí, ni explícitas ni implícitas, se ejecutarán de forma concurrente.

Si fuese necesario sincronizar el inicio de una actividad con la finalización de varias actividades concurrentes, se podría hacer esto estableciendo la restricción adecuada en el dominio de la transición funcional de inicio de esta actividad:

```
activity fin_concurrencia {
    transition iniciar {
       domain: {fin_concurrencia.estado = parado
                & concurrente1.estado = finalizado
                & concurrente2.estado = finalizado
                & concurrente3.estado = finalizado}
       action: {fin_concurrencia.estado = ejecutando}
    }
    ...
}
```

4.4.8. Ejecución condicional de actividades

La ejecución condicional permite controlar el hecho de que una actividad se ejecute o no en función del resultado de la evaluación de una condición, relativa al estado de objetos manipulados e incluso de otras actividades. A más bajo nivel, permite hacer lo mismo con la ejecución de transiciones funcionales. La evaluación de esta condición se asocia a un instante concreto: si en dicho instante se evalúa como falsa, la actividad no debe ser ejecutada aunque en el futuro haya algún instante en que dicha condición tome valor cierto. Una excepción a lo anterior sería que la actividad estuviese incluida en un bucle, en cuyo caso podría ser posible la ejecución de la actividad en algún instante de evaluación futuro. Una generalización de la ejecución condicional de una actividad es la selección condicional de una actividad entre varias, donde todas ellas tengan asociadas condiciones mutuamente excluyentes.

Una posible forma de modelar este comportamiento podría consistir simplemente en añadir la condición deseada mediante una *y–lógica* al predicado del dominio de la transición funcional de inicio de la actividad. Sin embargo, esto no modela correctamente el comportamiento condicional, dado que no impide ejecuciones futuras si cambiase el valor al que se evalúa la condición. Es necesario combinarlo con algún mecanismo que impida su ejecución futura.

Una de estas alternativas consistiría en, si el resultado de la evaluación es falso, mover la actividad a un estado *finalizado*, de tal forma que no pueda ejecutarse en el futuro. Sin embargo, esta alternativa tiene el problema de que, desde el punto de vista de otra actividad que dependa de esta, no sería distinguible si la actividad llegó a estado *finalizado* después de haberse ejecutado, o si nunca llegó a ejecutarse porque la condición se evaluó con valor falso.

La solución propuesta a dicho problema sería definir una nueva actividad. En esta actividad se tomaría una decisión, en base a la evaluación de la condición, acerca de qué actividad debería ser ejecutada (también podría ocurrir el caso de que ninguna deba ser ejecutada). En un atributo de control indicaría dicha decisión. Cada actividad sujeta a la ejecución condicional consultaría dicho atributo para detectar si puede o no ser ejecutada.

Ejemplo

Para ilustrar esto, se plantea el siguiente ejemplo con dos actividades alternativas. Una de ellas se ejecuta si el importe de una factura es superior o igual a 6000 y no ha sido pagada, y la otra si es inferior y no ha sido pagada. En caso de que la factura haya sido pagada, no se ejecuta ninguna de las dos.

```
enttype condicional_sel_t {
    seleccionado = enum(impago_mayor, impago_menor, ninguna);
}
entity condicional_sel: condicional_sel_t;
activity condicional {
    transition seleccion_mayor {
       domain: {condicional.estado = parado
                & factura.importe >= 6000
                & !factura.pago}
       action: {condicional.estado = ejecutando
                & condicional_sel = impago_mayor}}
    transition seleccion_menor {
       domain: {condicional.estado = parado
                & factura.importe < 6000
                & !factura.pago}
       action: {condicional.estado = ejecutando
                & condicional_sel = impago_menor}}
    transition seleccion_ninguna {
       domain: {condicional.estado = parado
                & factura.pago}
       action: {condicional.estado = finalizado
                & condicional_sel = ninguna}}
    transition finalizar {
       domain: {condicional.estado = ejecutando
                & (impago_mayor.estado = finalizado
                   | impago_menor.estado = finalizado)}
       action: {condicional.estado = finalizado}}
}
activity impago_mayor {
    transition iniciar {
       domain: {... & condicional.estado = ejecutando
                & condicional_sel = impago_mayor}
       action: {impago_mayor.estado = ejecutando}}
    ...
}
activity impago_menor {
    transition iniciar {
       domain: {... & condicional.estado = ejecutando
                & condicional_sel = impago_menor}
       action: {impago_menor.estado = ejecutando}}
    ...
}
```

4.4.9. Ejecución iterativa de actividades

Se puede modelar la ejecución iterativa de una actividad mediante una actividad de control que se encargue de gestionar el ciclo de vida de la actividad o actividades a ser ejecutadas de forma iterativa.

La ejecución iterativa suele basarse en la evaluación de una condición antes o después de cada iteración, para decidir si se continúa con la ejecución del bucle o se finaliza. Por otra parte, cada iteración debe controlar la ejecución de las actividades internas del bucle.

Por ejemplo, un bucle tipo *mientras–que* (*while*) con una condición asociada *condición* se puede modelar de la siguiente forma:

```
activity mientras {
    transition inicio {
       domain: {mientras.estado = parado}
       action: {mientras.estado = ejecutando
                & repetir.estado = parado}}
    transition continuar {
       domain: {mientras.estado = ejecutando
                & repetir.estado = finalizado
                & condición}
       action: {repetir.estado = parado}}
    transition finalizar {
       domain: {mientras.estado = ejecutando
                & (repetir.estado = parado
                   | repetir.estado = finalizado)
                & ! condición}
       action: {mientras.estado = finalizado}}
}
activity repetir {
    transition iniciar {
       domain: {mientras.estado = ejecutando
                & iniciar.estado = parado
                & condición}
       action: {...}
    ...
}
```

En el apartado 4.5.2 se analizan las limitaciones que pueden suponer, desde el punto de vista del modelado de estructuras iterativas, la ausencia de variables locales y la imposibilidad de instanciación múltiple de atributos.

4.4.10. Composición jerárquica de actividades

Algunos lenguajes de definición de procesos, como es el caso de BPEL4WS o BPML, permiten definir actividades que, a su vez, contengan sub–actividades. Por ejemplo, en BPEL4WS se representa una secuencia de actividades mediante una actividad compuesta `sequence` que contiene a las sub–actividades que deben ser ejecutadas en secuencia. La semántica habitual de una actividad compuesta es la siguiente:

- Cuando se inicia la ejecución de la actividad compuesta, se permite que comience la ejecución de una o más de sus sub–actividades. Por ejemplo, si la actividad representa una secuencia, la primera actividad de la secuencia puede comenzar su ejecución una vez la actividad compuesta lo haya hecho.
- Cuando finaliza la ejecución de la última sub–actividad, la actividad compuesta finaliza su ejecución. Por ejemplo, en una actividad compuesta cuyas sub–actividades deben ser ejecutadas en paralelo, la actividad finaliza cuando todas sus sub–actividades lo hayan hecho.

La composición jerárquica de actividades puede resultar útil aún cuando el lenguaje de definición de procesos no lo permita, dado que facilita la implementación sencilla de ejecuciones en secuencia, en paralelo, condicionales, iterativas, gestión de fallos con ámbitos anidados, etc. Por otra parte, la composición de actividades puede simplificar las interacciones de sus sub–actividades con otras actividades del proceso. Por ejemplo, si una actividad tiene una dependencia con varias actividades que deben ser ejecutadas en paralelo, es más sencillo declarar una dependencia con una actividad compuesta que englobe a todas ellas, que con todas las actividades directamente.

4.4.11. No–determinismo

Las definiciones de procesos de negocio CFM son, en general, no deterministas, en el sentido de que para un estado del proceso puede existir más de un estado hacia el cual este puede evolucionar. En este caso, según la definición del sistema formal básico dada en el apartado 4.1, se selecciona de forma no determinista hacia cuál de ellos avanza. Desde el punto de vista de la verificación con exploración exhaustiva, todos los estados posibles deben ser explorados. Por tanto, se realiza una bifurcación en el árbol de caminos a explorar. Este no–determinismo aparece implícitamente en un proceso por el hecho de permitir la ejecución concurrente de actividades: la definición del proceso no especifica un orden concreto de ocurrencia de eventos de inicio o finalización de las actividades, sino que se pueden producir en cualquier orden.

Además del no–determinismo que aparece implícitamente debido a la concurrencia, existen otras situaciones en las que introducir explícitamente no-determinismo en un proceso permite simplificar su propia definición. Una de estas situaciones es la abstracción de aspectos acerca del control de flujo del proceso que no resulten relevantes desde el punto de vista de la verificación.

Por ejemplo, supongamos que en un determinado proceso se toma una decisión condicional, basada en cálculos muy complejos que obligan a representar una gran cantidad de atributos. Esta decisión afecta a la ejecución posterior del proceso, bifurcando el flujo en dos ramas alternativas. Si se modela explícitamente la toma de la decisión, es necesario representar los atributos en los cuales se basa la decisión y la propia lógica de toma de decisión. Si estos atributos no fuesen relevantes desde el punto de vista de la verificación, se podrían abstraer mediante no–determinismo. Para ello, no se declararían en el modelo CFM (o se declararían con tipo *abstracto*) y se permitiría que en un momento dado se escogiese una entre varias alternativas, sin especificar cómo se toma dicha decisión. Desde el punto de vista de la exploración durante la verificación, se formará una bifurcación en el árbol de caminos. El hecho de abstraer atributos innecesarios puede en muchas ocasiones simplificar en gran medida el tamaño del espacio de estados del proceso, y reducir el número de transiciones funcionales o, en todo caso, hacerlas más simples.

Esta técnica de abstracción tiene la propiedad de que todos los caminos seleccionables en el sistema original serán seleccionables también en el sistema abstraído. Sin embargo, es posible que en el sistema abstraído sean seleccionables caminos que no son seleccionables en el sistema original. Desde el punto de vista de la verificación es necesario, por tanto, que el diseñador sea cuidadoso cuando añade no–determinismo, dado que puede existir alguna propiedad de la especificación que en el proceso sin abstracción se evalúe con distinto valor

que en el proceso con abstracción. En el siguiente ejemplo se ilustra la introducción de no determinismo en el proceso y se muestra cómo algunas propiedades pueden ser evaluadas con distinto valor.

El no–determinismo resulta útil, por otra parte, para modelar la ejecución de actividades del proceso que no se ejecuten de forma automática, sino que requieran intervención humana. En este caso, se combina el mecanismo de no–determinismo con el de abstracción para modelar los distintos resultados de la ejecución de dicha actividad.

Ejemplo

Supongamos que en el ejemplo planteado en el apartado 4.4.8 se abstrae la toma de decisión acerca del tipo de impago. La actividad de selección se simplificaría:

```
activity condicional {
    transition seleccion_mayor {
       domain: {condicional.estado = parado}
       action: {condicional.estado = ejecutando
                & condicional_sel = impago_mayor}}
    transition seleccion_menor {
       domain: {condicional.estado = parado}
       action: {condicional.estado = ejecutando
                & condicional_sel = impago_menor}}
    transition seleccion_ninguna {
       domain: {condicional.estado = parado}
       action: {condicional.estado = finalizado
                & condicional_sel = ninguna}}
    transition finalizar {
       domain: {condicional.estado = ejecutando
                & (impago_mayor.estado = finalizado
                   | impago_menor.estado = finalizado)}
       action: {condicional.estado = finalizado}}
}
```

En este ejemplo las tres transiciones funcionales tienen exactamente el mismo dominio, y además son mutuamente excluyentes, dado que el hecho de ejecutar cualquiera de ellas cambia el estado de la actividad en su ciclo de vida de tal forma que ninguna otra pueda ser seleccionada. Hay propiedades a las cuales esta abstracción no afecta. Por ejemplo, la afirmación: *para algún camino de computación es posible que se ejecute la actividad* `impago_mayor` es cierta en el proceso original y en el abstraído. Sin embargo, la propiedad *nunca puede ocurrir que se seleccione la actividad* `impago_mayor` *si la factura está pagada* es falsa en el modelo abstraído, pero cierta en el original. El origen de este problema está en que en el sistema original los caminos `impago_mayor` e `impago_menor` son sólo seleccionables si la factura ha sido pagada, mientras que en el sistema abstraído son seleccionables independientemente del pago.

En este caso sería sencillo detectar este problema por inspección visual, pero en general podría no serlo. Para mostrar este hecho, supongamos que el diseñador es consciente de que debe introducir también en la decisión el estado del pago, reduciendo así el no–determinismo a sólo las dos primeras opciones. Supongamos además que en algún momento anterior en el proceso se marca una factura como *muy importante* si su importe es superior a 15000. Supongamos, además, que el proceso define que en el camino para impagos mayores un alto

cargo de la empresa debe contactar con un alto cargo de la empresa cliente, mientras que en caso de impagos menores se contacta únicamente a nivel de administrativos de menor rango. La propiedad *dado un pedido marcado como importante, siempre que se contacta con el cliente debido a dicho pedido, es un alto cargo quien lo hace* sería, en principio, cierta en el sistema original, pero falsa en el abstraído, dado que en este último se pueden seleccionar los caminos de impago mayor o menor independientemente del importe de la factura o su importancia. En este caso la relación entre la propiedad a verificar y la abstracción aplicada es menos obvia que en el caso anterior.

Por tanto, aunque la abstracción mediante no determinismo permite simplificar las definiciones de procesos, esta simplificación no da lugar a un modelo equivalente. Es necesario que el diseñador sea extremadamente cuidadoso cuando aplique esta técnica, para evitar resultados de verificación incongruentes con el sistema original. Esto tiene también implicaciones en el desarrollo de herramientas de transformación desde lenguajes de la capa 3 a lenguajes de la capa 2. Estas herramientas nunca deberían automáticamente aplicar este tipo de abstracción si no pueden demostrar que el sistema sería equivalente desde el punto de vista de la especificación.

4.5. Limitaciones

En este apartado se analizan las distintas limitaciones identificadas en el modelo CFM desde el punto de vista de su capacidad expresiva para representar procesos de negocio. Para cada una de ellas, se justifica su importancia.

4.5.1. Limitaciones en la representación de atributos

El sistema formal propuesto en este capítulo no es capaz de representar tipos de datos que surgen de forma natural cuando se pretende representar las propiedades de los objetos manipulados por un proceso de negocio. En concreto, resulta imposible representar tipos de datos que pueden tomar un conjunto infinito de valores. Por ejemplo, podemos citar, entre otros, cadenas de texto, números enteros arbitrariamente grandes o números reales. Tampoco es práctico representar por enumeración tipos de datos que pueden tomar un conjunto finito, pero grande, de valores. En el caso de una cadena de texto, aunque se acotase el número de caracteres para reducir su rango de valores a un conjunto finito, seguiría siendo impracticable (aunque formalmente posible) representarla como dato enumerado.

La principal causa de este problema es el hecho de que el sistema formal sólo puede tener, por definición, un conjunto finito de estados y, por tanto, todos los atributos deben tener asociado un conjunto finito de valores. Esta decisión de diseño es razonable, dado que las técnicas de verificación más potentes funcionan en sistemas con un número finito de estados. Permitir la definición de atributos con infinitos valores supondría limitar el sistema a la aplicación de técnicas de *verificación deductiva*. En el capítulo 6 se razona por qué esto no resulta adecuado.

Dado que hay razones de peso para no introducir en el sistema atributos con un rango infinito de valores, ¿cómo afecta esto a la expresividad del sistema, es decir, a su capacidad para representar procesos de negocio? A continuación se argumenta por qué esto no tiene por qué suponer una limitación importante a la expresividad.

El objetivo principal del CFM es la verificación de procesos de negocio. Por tanto, no es necesario que sea capaz de modelar completamente los procesos, sino sólo aquellas características de los mismos que sean relevantes desde el punto de vista de la verificación. Teniendo en cuenta esto, cabe preguntarse si realmente son relevantes todos los posibles valores de una determinada propiedad de un objeto manipulado, o si por el contrario se pueden aplicar técnicas de abstracción sin reducir por ello potencia al proceso de verificación.

Supongamos que, dado el conjunto infinito V de valores de un atributo, se define una partición finita $\{V_1, V_2, \ldots, V_n\}$ del mismo tal que:

$$V = \bigcup_{i=1}^{n} V_i \tag{4.8}$$

$$\bigcap_{i=1}^{n} V_i = \emptyset \tag{4.9}$$

Si se definiese convenientemente esta partición, en muchos casos lo relevante desde el punto de vista de la verificación no sería el valor concreto del atributo, sino a cuál de los subconjuntos de la partición pertenece. Dado que el número de subconjuntos de la partición es, por definición, finito, entonces se podría abstraer el atributo a un rango enumerado donde cada valor representa la pertenencia a uno de los subconjuntos.

Aunque pueden existir procesos para los cuales una partición en subconjuntos relevantes de los valores de un atributo conste de un número muy elevando de subconjuntos, esto no es razonable en la práctica, dado que supondría la existencia de número de decisiones condicionales en la definición del proceso igualmente elevado. Aún así, la abstracción del valor de los atributos tiene también sus limitaciones, como se muestra al final del siguiente ejemplo.

Ejemplo

Con este ejemplo se pretende hacer más clara la discusión anterior, así como resaltar, para concluir, una posible limitación de esta técnica de abstracción.

Supongamos el proceso de negocio de gestión de reclamaciones de una compañía aseguradora. En algún momento del proceso se evalúa el importe de la reclamación. Supongamos que, dependiendo del importe de la reclamación, el proceso debe ser encaminado por unas u otras actividades (camino A para cuantías menores que 120 euros, camino B para cuantías entre 120 y 600 euros y camino C para el resto).

En principio, es natural representar el importe como un número natural, en céntimos de euro, que podría tomar valores arbitrariamente grandes. Su conjunto de posibles valores es infinito. Aunque existiese una cota limitando el importe máximo, resulta razonable suponer que el conjunto de valores continuaría siendo, aunque finito, considerablemente grande.

Dadas las características de este proceso de negocio, resulta conveniente realizar una partición del conjunto de valores ($V = \mathbb{N}$) en tres subconjuntos: $V_1 = [0, 120)$, $V_2 = [120, 600)$ y $V_3 = [600, \infty)$. De esta forma, se podría representar el importe como un atributo de tipo enumerado con tres posibles valores:

```
importe: enum(bajo,medio,alto);
```

Esta técnica tiene también algún inconveniente. El principal se deriva del hecho de que no permite realizar operaciones matemáticas con los valores. Supongamos que en algún punto del proceso se establece, bajo determinadas circunstancias, una compensación adicional de 50 euros sobre el importe inicialmente evaluado. Ahora, algunos valores del subconjunto `bajo` podrían cambiar al subconjunto `medio`, mientras que otros permanecerían en el mismo subconjunto. En este caso se podría solucionar el problema haciendo una partición más fina: $V_1 = [0, 70)$, $V_2 = [70, 120)$, etc. Sin embargo, esto no solucionaría el problema si la cantidad a sumar, en lugar de ser conocida en tiempo de diseño, se estableciese durante la ejecución de cada instancia del proceso. Una solución que funcionaría en este caso consiste en modelar también de forma abstracta la operación de suma de la compensación adicional, mediante tantas transiciones funcionales como posibles resultados se pudiesen obtener. En este caso se añade *no–determinismo* al modelo del proceso para representar la incertidumbre debida a las abstracciones realizadas. Todas las transiciones funcionales compartirían el mismo dominio pero tendrían distinta acción:

```
transition alternativa_1 {
    domain: {...}
    action: {reclamacion.importe = bajo}
}
transition alternativa_2 {
    domain: {...}
    action: {reclamacion.importe = medio}
}
transition alternativa_3 {
    domain: {...}
    action: {reclamacion.importe = alto}
}
```

Cabe destacar también que las técnicas descritas en este ejemplo no sólo son una solución para poder representar ciertos tipos de atributos con rango infinito de valores. También permiten reducir considerablemente la complejidad del problema en situaciones en que se declaran atributos de tipo entero. Aunque su rango de valores es finito, provocan un crecimiento drástico en el tamaño del espacio de estados, como se ha mostrado en el ejemplo del apartado 4.2.2. Es frecuente que los datos enteros puedan ser abstraídos sin perder por ello potencia de verificación.

Conclusiones

A partir del análisis planteado para este problema, se puede concluir que es posible normalmente modelar procesos que posean atributos con un número infinito, o finito pero grande, de posibles valores. Para ello, se puede recurrir a realizar abstracciones en el valor de los atributos o añadir no–determinismo. Por otra parte, estas técnicas también permiten reducir considerablemente la complejidad asociada a la verificación.

4.5.2. Ausencia de entidades de ámbito local

En CFM todas las entidades, y por tanto los atributos, tienen ámbito global. Por tanto, no es posible declarar entidades o atributos de ámbito local. Los principales motivos por los que resulta interesante contar con entidades de ámbito local son los siguientes:

- Limitación del acceso a las entidades según su ubicación en el proceso. Esto reduce la probabilidad de que se produzcan efectos colaterales debido a accesos indebidos a las entidades. En consecuencia, podrían declararse entidades con el mismo nombre siempre y cuando perteneciesen a ámbitos no solapados.

- Instanciación múltiple de una misma entidad. En el modelo habitual de lenguajes de programación convencionales, las variables locales se almacenan en la pila del programa, de tal forma que, en un momento dado, pueden existir múltiples instancias de una misma variable local. Esto facilita, por ejemplo, la programación de algoritmos recursivos y la programación multihilo. Aplicado a CFM, esto facilitaría la instanciación múltiple de entidades y, por tanto, la instanciación múltiple de actividades.

El hecho de no limitar el acceso a una entidad a determinadas actividades no supone una gran limitación en CFM. El modelo está principalmente pensado para que los modelos CFM se generen con herramientas automatizadas a partir de definiciones de procesos de la capa 3. Las propias herramientas de transformación, si son correctas, deben generar procesos de acuerdo a las limitaciones en el acceso existentes en la definición original del proceso. Si existen varios objetos con el mismo nombre pero distinto ámbito en la definición original, estas herramientas pueden cambiar el nombre a una de ellas para evitar colisiones de nombre en la definición CFM.

Sin embargo, el hecho de no permitir la instanciación múltiple de una misma entidad, basada en un modelo de pila, sí puede suponer una limitación para representar procesos de negocio expresados en determinados lenguajes de definición de procesos. Algunas situaciones en que se hace patente esta limitación son la representación de subprocesos y la instanciación múltiple de actividades en estructuras de flujo iterativas (bucles). Se analizan a continuación estos conceptos y las limitaciones de CFM en su representación.

Subprocesos

El concepto de subproceso es similar al de subrutina en un lenguaje de programación convencional. Aunque no es un concepto habitual entre los principales lenguajes de definición de procesos de negocio, aparece en alguno de ellos, como es el caso de BPML. Una subrutina está compuesta principalmente por un conjunto de actividades y, opcionalmente, entidades de ámbito local. Nótese que tanto los parámetros de entrada y salida de la subrutina como las propias entidades que controlan el ciclo de vida de las actividades de la subrutina pueden ser representados también como entidades de ámbito local.

El sistema CFM, que sólo permite declarar entidades de ámbito global, puede modelar subrutinas siempre y cuando cumplan la siguiente restricción: no debe haber en ningún momento más de una instancia simultánea de una misma subrutina. Esto es así porque, si no se permite que una subrutina se pueda invocar a sí misma directamente, ni indirectamente a través de otras subrutinas, no es necesario instanciar múltiples veces ninguna de estas entidades de ámbito local. En consecuencia, pueden ser representadas como entidades de ámbito global.

Si se permitiesen múltiples instancias simultáneas, acotadas en número para cada subrutina en tiempo de diseño, se podrían declarar tantas entidades globales como fuese necesario para cada entidad de ámbito local. Sin embargo, sería necesario modificar las definiciones de

CFM para permitir que una misma actividad pueda hacer referencia a las entidades asociadas con cada una de sus instancias. Para hacer esto, una posible solución sería incluir un índice de anidamiento de subrutina, y utilizar este índice para seleccionar las entidades adecuadas.

En cualquier caso, por no definir el concepto de *pila*, el formalismo CFM es incapaz de representar subrutinas que puedan ser ejecutadas un número arbitrario de veces.

Conclusiones

De la discusión planteada en este apartado se puede concluir que el hecho de no disponer de una *pila* de entidades dificulta la representación de algunos patrones avanzados presentes en algunos lenguajes de definición de procesos, como es el caso de las entidades de ámbito local, los subprocesos y los bucles. Aunque es posible representar parcialmente estos patrones, dichas representaciones pueden ser excesivamente complejas o incompletas.

4.5.3. Limitaciones en la comunicación entre distintos casos

Una limitación habitual de lenguajes de definición de procesos de negocio es la representación de casos, o instancias del proceso, de forma aislada e independiente del resto de casos del mismo proceso. Aalst [5] identifica esta limitación, justifica que en algunas situaciones es necesaria la existencia de sincronización entre distintos casos, y propone, para resolver el problema, un mecanismo llamado *Proclets*.

El formalismo CFM también sufre dicha limitación: cada caso se modela de forma aislada, tal y como se expone en el apartado 4.4.1. Por tanto, si fuese necesario representar un proceso con coordinación entre casos, habría que recurrir a aplicar técnicas de *abstracción*, probablemente mediante *no–determinismo*.

Conclusiones

El formalismo CFM debe ser capaz de representar procesos definidos con una amplia variedad de lenguajes de definición. Dado que la mayoría de estos lenguajes y formalismos de definición de procesos de negocio modelan también de forma aislada cada caso del proceso, el hecho de que también lo haga el formalismo CFM no supone una limitación excesivamente importante.

4.5.4. Limitaciones en la instanciación múltiple de actividades

El formalismo CFM permite instanciar una actividad más de una vez, siempre y cuando las instancias no se ejecuten de forma concurrente. También permite instanciar una actividad más de una vez si se conoce, en tiempo de diseño, el número máximo de instancias. Sin embargo, no permite la instanciación múltiple de actividades cuando dichas actividades pueden ser ejecutadas concurrentemente y no se conoce en tiempo de diseño el número máximo. Esta limitación está muy relacionada con la expuesta en el apartado en relación a la imposibilidad de declarar entidades de ámbito local.

Cada instancia necesita al menos un atributo para controlar su propio ciclo de vida, pero el formalismo CFM no permite instanciar un mismo atributo en más de una ocasión. Cuando

las actividades no se ejecutan de forma concurrente (por ejemplo, en el interior de un bucle), no existe ningún problema, porque se puede reutilizar la misma instancia del atributo, siempre y cuando se establezca de nuevo su valor inicial cuando corresponda. Cuando las actividades se pueden ejecutar de forma concurrente, pero se conoce en tiempo de diseño el número máximo de instancias, se puede duplicar la declaración del atributo y de la actividad tantas veces como sea necesario, solucionando también así el problema. Sin embargo, cuando no se conoce el número máximo en tiempo de diseño, no es posible solucionar el problema, dado que el formalismo no permite instanciar atributos en tiempo de ejecución.

En el apartado 5.1.4 se profundiza en estas limitaciones, dado que una parte de los patrones de *workflow* se refieren a la instanciación múltiple de actividades.

Conclusiones

El problema presentado en este apartado sí puede suponer una limitación en la capacidad del formalismo CFM. En el capítulo 5 se ve que, de hecho, los patrones que CFM no puede representar correctamente están relacionados la instanciación múltiple de atributos y actividades. En cualquier caso, si se observan los resultados de realizar un análisis basado en patrones a distintos lenguajes y formalismos, presentados en los cuadros 5.1, 5.2 y 5.3, se ve que sólo el lenguaje YAWL da soporte a estos patrones de instancias múltiples (patrones 14 y 15). Por tanto, aunque es una limitación importante, no resulta crítica.

4.6. Conclusiones

En este capítulo se ha presentado la definición de un formalismo para representar procesos de negocio y composiciones de servicios Web, con el fin de integrarlo en la capa 2 de la arquitectura propuesta en el capítulo 3.

La definición está compuesta de tres niveles de abstracción o capas. La de menor nivel define el sistema formal básico, la intermedia lo dota de conceptos cercanos al campo del modelado de procesos de negocio basado en actividad, y la de mayor nivel define una notación textual para describir procesos conformes al formalismo. El sistema formal básico no es, en sí mismo, una contribución original, dado que se define como un diagrama estado–transición etiquetado. Sin embargo, sí lo son las otras dos capas que componen el modelo CFM.

La principal idea original presentada en este capítulo es la metodología expuesta para representar procesos de negocio y composiciones de servicios Web mediante el formalismo. Esta metodología permite representar de forma bastante natural y sencilla las principales construcciones presentes en procesos de negocio con modelado basado en actividad. Dado que debe ser aplicable a distintos lenguajes de definición de procesos, se presenta la metodología de forma considerablemente abstracta. Sin embargo, en el capítulo 7 se concreta esta metodología con el objetivo de aplicarla al lenguaje BPEL4WS.

Cierto es que el modelado es más visual e intuitivo con otros lenguajes y formalismos. Sin embargo, la principal ventaja de este formalismo radica en su sencillez para realizar verificaciones: es un formalismo bien conocido al cual se puede aplicar una amplia gama de herramientas de verificación. En el capítulo 6 se profundiza en las capacidades de verificación de este sistema formal, y se puede observar esta característica de sencillez con dos

herramientas de *model checking* concretas: Spin y SMV.

Por último, se realiza un estudio de las principales limitaciones que podría tener CFM para la representación de procesos de negocio. Aunque algunas de estas limitaciones llegan a tener una importancia moderada, ninguna de ellas supone un obstáculo grave para que resulte adecuado para la representación de procesos de negocio. Es más, en el capítulo 5 se analiza desde un punto de vista más formal la adecuación de CFM para representar procesos de negocio, y se concluye que, aunque con alguna pequeña limitación, resulta adecuado para formar parte de la capa 2 de la arquitectura.

Capítulo 5

Análisis del Modelo Formal Común basado en Patrones de Workflow

The absence of a universal organizational "theory", and standard business process modeling concepts, it is contended, explains and ultimately justifies the major diferences in workflow languages – fostering up a "horses for courses" diversity in workflow languages.

(Aalst, Hofstede, Kiepuszewski y Barros, "Workflow Patterns", Distributed and Parallel Databases, 14(3), pag. 5-51, 2003)

El objetivo de este capítulo es comprobar que el *Modelo Formal Común* es un formalismo adecuado para la capa 2 del la arquitectura propuesta en esta tesis, desde el punto de vista de su capacidad expresiva en la perspectiva de control. Para ello, se realizará un análisis de expresividad basado en los *patrones de workflow* [9]. Como se menciona en el apartado 2.2.3, estos veinte patrones han sido diseñados como una herramienta de comparación de la capacidad expresiva y adecuación de distintos lenguajes de definición de procesos de negocio. Cada uno de ellos representa una funcionalidad útil para el modelado de procesos de negocio. La adecuación de cada lenguaje se mide, mediante este tipo de análisis, en términos de los patrones representables de forma natural mediante dicho lenguaje. Se ha escogido esta metodología de análisis por su amplia difusión en el ámbito académico y a la disponibilidad de los resultados de aplicar este análisis a numerosos lenguajes [125, 7, 6, 3, 80]. Los *patrones de workflow* se centran en la perspectiva de control, pero no modelan la perspectiva de datos.

En la primera parte de este análisis se plantea, para cada uno de los veinte patrones, una discusión acerca de la capacidad del formalismo CFM para representarlo. Posteriormente, se compara la expresividad de CFM con la de otros lenguajes de definición de procesos de negocio. Se concluye exponiendo brevemente los resultados más relevantes del análisis y de la comparación.

5.1. Patrones de *workflow* en CFM

En este apartado se analiza la capacidad del formalismo CFM para representar cada uno de los veinte *patrones de workflow*. Para cada uno de ellos, se presenta en primer lugar una breve descripción del mismo y se discute si CFM es capaz de representarlo. Si es posible representarlo, se expone cómo. Si no, se indica por qué no es posible su representación. La descripción completa de los patrones y sus alternativas de implementación se discuten en [9]. Por simplicidad, en los ejemplos planteados en este apartado se modela el ciclo de vida de cada actividad mediante el ejemplo propuesto en el apartado 4.4.4 (figura 4.1).

En el análisis de algunos patrones se emplea el concepto de *hilo de control*. Este concepto es similar al de *hilo* en lenguajes de programación. Un hilo de control define un conjunto de actividades que deben ser ejecutadas en secuencia (una tras otra y en un orden predeterminado). En una misma instancia de un proceso de negocio pueden estar ejecutándose en paralelo varios hilos de control. En algunos patrones también se emplea el concepto de *rama*. Las ramas de un proceso son secuencias de actividades que se ejecutan o no de acuerdo con la evaluación de una condición en un momento dado. Aunque una rama tiene una gran semejanza con un hilo de control, se diferencia en que la ejecución de una rama no implica necesariamente la creación de un nuevo hilo de control. Esto es así cuando se selecciona una rama de entre varias ramas alternativas, en cuyo caso no se realiza una división en hilos de control paralelos. Sin embargo, cuando se selecciona más de una rama de entre varias, en realidad las ramas seleccionadas se ejecutan en paralelo y, por tanto, también se pueden ver como varios hilos de control.

5.1.1. Patrones de control de flujo básico

Los patrones de este grupo se centran en aspectos elementales del flujo de control de un proceso de negocio. La mayoría de ellos han sido discutidos en el apartado 4.4 al introducir las líneas generales de representación de procesos mediante CFM.

Patrón 1: secuencia

Se permite la ejecución de una actividad del proceso después de haber terminado otra actividad del mismo proceso.

Solución La solución a este patrón se ha estudiado en el apartado 4.4.6. Se puede secuenciar la ejecución de dos actividades añadiendo una condición en el dominio de la transición funcional de inicio de la segunda actividad, de tal forma que no pueda comenzar su ejecución mientras la primera actividad no esté en estado *finalizado*. Esta solución puede ser generalizada fácilmente a la ejecución en secuencia de más de dos actividades, aplicando esto mismo a cada par de actividades consecutivas de la secuencia. A continuación se muestra un ejemplo con dos actividades:

```
activity A1 {
    transition inicio {
       domain: {A1.estado = parado}
       action: {A1.estado = ejecutando}
    }
    ...
```

```
}
activity A2 {
    transition inicio {
       domain: {A2.estado = parado & A1.estado = finalizado}
       action: {A2.estado = ejecutando}
    }
    ...
}
```

Patrón 2: división en paralelo

En un punto dado del proceso un hilo de control se divide en múltiples hilos de control que pueden ser ejecutados en paralelo, de tal forma que las actividades de un hilo puedan ser ejecutadas concurrentemente o en cualquier orden con respecto a las de los otros hilos.

Solución Como se explica en el apartado 4.4.7, en CFM las actividades se ejecutan, por defecto, en paralelo, salvo que se delimiten hilos de control mediante estructuras de ejecución en secuencia. Para modelar este patrón se recurre a crear una actividad auxiliar que se ejecute en el hilo de control inicial, inmediatamente antes de la división en paralelo. A la primera actividad (o la envolvente jerárquicamente) de cada nuevo hilo de control se le añade en el dominio de la transición funcional de inicio una condición que establece que la actividad auxiliar debe estar en ejecución. Esto es, se bloquea la ejecución de cada hilo hasta que se haya alcanzado el punto de la división en paralelo en el hilo de control principal.

En algunos casos será necesario realizar una unificación posterior de todos los hilos de control en uno único, de forma sincronizada, mientras que en otros casos no es esto necesario. El primer caso (división en paralelo seguida de una sincronización posterior) se plantea tras exponer el patrón 3 (sincronización). En el siguiente ejemplo se plantea el segundo caso, en el cual no es necesaria una sincronización posterior. Por tanto, se puede dar por finalizada la ejecución de la actividad auxiliar una vez se hayan iniciado todos los hilos de control. El ejemplo muestra cómo tras finalizar la actividad A0 se divide el hilo de control principal en dos nuevos hilos, iniciados por las actividades A1 y A2. La actividad auxiliar se llama S:

```
activity S {
    transition inicio {
        domain: {S.estado = parado & A0.estado = finalizado}
        action: {S.estado = ejecutando}}
    transition fin {
        domain: {S.estado = ejecutando & A1.estado != parado
                 & A2.estado != parado}
        action: {S.estado = parado}}
}
activity A1 {
    transition inicio {
       domain: {A1.estado = parado & S.estado = ejecutando}
       action: {A1.estado = ejecutando}
    }
    ...
}
activity A2 {
    transition inicio {
       domain: {A2.estado = parado & S.estado = ejecutando}
       action: {A2.estado = ejecutando}
    }
    ...
}
```

Patrón 3: sincronización

Representa un punto en el proceso en el cual varios subprocesos o hilos de control paralelos finalizan y convergen de forma sincronizada en un único hilo de control. El hecho de que la convergencia sea sincronizada implica que la actividad siguiente al punto de convergencia no se puede ejecutar hasta que todos los hilos de entrada hayan finalizado. Se asume que cada hilo de entrada a la sincronización se ejecuta una única vez. Si no fuese el caso, consúltense los patrones 13 a 15.

Solución Para realizar la sincronización se puede añadir una actividad auxiliar que espere a que todos los hilos de entrada finalicen su ejecución. Para ello, en el dominio de su transición funcional de inicio establecerá que la última actividad de cada uno de los hilos (o la actividad envolvente de cada uno de ellos, dependiendo de la estrategia empleada para representar un hilo) debe estar en estado *finalizado*. La actividad auxiliar pasará en ese momento directamente de estado *parado* a estado *finalizado*. Nótese que si hubiese más de un estado que representase la finalización del ciclo de vida de una actividad (por ejemplo uno para la finalización correcta y otro para la finalización con errores), se realizaría una o–lógica de todos esos estados. A continuación se muestra un ejemplo en que se sincronizan las últimas actividades de dos hilos (A1 y A2) mediante una actividad auxiliar (S):

```
activity S {
    transition inicio {
        domain: {S.estado = parado & A1.estado = finalizado
                 & A2.estado = finalizado}
        action: {S.estado = finalizado}}
}
```

Una alternativa que no requiere la creación de una actividad auxiliar está en añadir el código necesario para la sincronización directamente en la primera actividad tras el punto de convergencia, añadiendo a su transición funcional de inicio las mismas condiciones que las expuestas para la actividad auxiliar.

Solución conjunta a los patrones 2 y 3 En ocasiones resulta interesante combinar una división en paralelo (patrón 2) con una sincronización posterior (patrón 3). Si todos los hilos de la división se vuelven a unir en la sincronización, es posible englobar todo el bloque de actividades dentro de una actividad jerárquicamente superior que controle la división y sincronización. La división se controla mediante su transición funcional de inicio. La sincronización, mediante su transición funcional de finalización. El siguiente ejemplo muestra cómo un hilo de control se divide en dos tras ejecutar A0. El primer hilo se inicia con la actividad A1 y finaliza con A5, mientras que el segundo se inicia con A6 y finaliza con A8. Se omite el código de A5 y A8 porque no es relevante para comprender el ejemplo.

```
activity S {
    transition inicio {
        domain: {S.estado = parado & A0.estado = finalizado}
        action: {S.estado = ejecutando}}
    transition fin {
        domain: {S.estado = ejecutando & A5.estado = finalizado
                 & A8.estado = finalizado}
        action: {S.estado = parado}}
```

```
}
activity A1 {
    transition inicio {
       domain: {A1.estado = parado & S.estado = ejecutando}
       action: {A1.estado = ejecutando}
    }
    ...
}
activity A6 {
    transition inicio {
       domain: {A6.estado = parado & S.estado = ejecutando}
       action: {A6.estado = ejecutando}
    }
    ...
}
...
```

Patrón 4: selección exclusiva

En un punto del proceso se selecciona una entre varias ramas alternativas, basándose en una decisión o en el valor de atributos del proceso.

Solución La solución a este patrón está en añadir a la actividad de inicio de cada rama, o a la actividad que envuelva a cada rama, la condición asociada a dicha rama. Con este modelo, las condiciones deben ser disjuntas, para evitar que se seleccione más de una rama. Otra alternativa, que se discute una vez explicado el patrón 5, sería envolver la decisión y todas las actividades de sus ramas en una actividad, en los casos en que fuese posible. A continuación se muestra un ejemplo en que, tras la ejecución de la actividad A0, se decide entre las ramas iniciadas por las actividades A1 y A2, dependiendo de si el atributo `factura.importe` es, respectivamente, mayor o igual que 100, o menor que cien:

```
activity A1 {
    transition inicio {
       domain: {A1.estado = parado & A0.estado = finalizado
                & A2.estado = parado & factura.importe >= 100}
       action: {A1.estado = ejecutando}
    }
    ...
}
activity A2 {
    transition inicio {
       domain: {A2.estado = parado & A0.estado = finalizado
                & A1.estado = parado & factura.importe < 100}
       action: {A2.estado = ejecutando}
    }
    ...
}
```

Patrón 5: mezcla simple

Este patrón representa un punto en el proceso en el cual dos o más ramas alternativas se unen sin sincronización. Se asume que se activa exactamente una de las ramas. En caso de que se pudiese activar más de una rama, consúltense los patrones 8 y 9.

Solución La representación de este patrón es similar a la del patrón 3, teniendo en cuenta que en un caso tenemos ramas alternativas y en otro hilos de control paralelos. La diferencia está en que en este patrón no es necesario esperar a que todas las ramas finalicen, sino que basta con que lo haga una de ellas. De hecho, sólo una de ellas se podría estar ejecutando en un momento dado. Para representar este patrón, se puede definir una actividad auxiliar, añadiendo al dominio de su transición funcional de inicio una condición, compuesta por *o–lógicas*, que establezca que la última actividad (o la actividad envolvente) de una de las ramas debe haber finalizado. Se puede evitar definir esta actividad auxiliar si se añade la condición al dominio de la primera actividad a ser ejecutada tras la mezcla. A continuación se muestra un ejemplo en que se introduce una actividad auxiliar (S) para mezclar tres ramas, finalizadas por las actividades A1, A2 y A3:

```
activity S {
    transition inicio {
        domain: {S.estado = parado
                 & (A1.estado = finalizado | A2.estado = finalizado
                    | A3.estado = finalizado)}
        action: {S.estado = finalizado}}
}
```

Solución conjunta a los patrones 4 y 5 En las situaciones en que se pueda combinar de forma estructurada una selección exclusiva con una mezcla, esto es, se mezclen exactamente las ramas generadas en la selección, se puede definir una actividad que envuelva jerárquicamente a las actividades de todas las ramas alternativas. Se controlan la división y la mezcla mediante las transiciones de inicio y finalización de la actividad envolvente. En el siguiente ejemplo se ilustra esta alternativa. La selección se realiza tras la finalización de la actividad A0, mediante la ejecución de una de las transiciones funcionales de inicio de la actividad envolvente (S). Una de las ramas comienza con la actividad A1 y acaba con A5. La otra comienza con A6 y acaba con A8.

```
activity S {
    transition inicio_A1 {
        domain: {S.estado = parado & A0.estado = finalizado
                 & factura.importe >= 100}
        action: {S.estado = ejecutando & S.seleccion = sel_A1}}
    transition inicio_A6 {
        domain: {S.estado = parado & A0.estado = finalizado
                 & factura.importe < 100}
        action: {S.estado = ejecutando & S.seleccion = sel_A6}}
    transition fin {
        domain: {S.estado = ejecutando
                 & (A5.estado = finalizado | A8.estado = finalizado)}
        action: {S.estado = parado}}
}
activity A1 {
    transition inicio {
       domain: {A1.estado = parado & S.estado = ejecutando
                & S.seleccion = sel_A1}
       action: {A1.estado = ejecutando}
    }
    ...
}
activity A6 {
    transition inicio {
       domain: {A6.estado = parado & S.estado = ejecutando
```

```
              & S.seleccion = sel_A6}
       action: {A6.estado = ejecutando}
    }
    ...
}
...
```

5.1.2. Patrones avanzados de selección y sincronización

Este grupo de patrones modelan situaciones de selección condicional, mezcla, división y sincronización más complejas que las presentadas en el grupo de patrones básicos. Mientras que la mayoría de los lenguajes de definición de procesos dan soporte a los patrones básicos, son pocos, según se expone en [9], los que dan soporte a estos patrones avanzados.

Patrón 6: selección múltiple

En un punto del proceso, de acuerdo a una decisión o a datos de control del proceso, se seleccionan una o más alternativas, que se ejecutan en paralelo. La diferencia con el patrón 4 está en que se puede seleccionar más de una alternativa.

Solución En CFM se puede modelar este patrón de una forma muy similar al patrón 4, con la diferencia de que en esta ocasión las condiciones no tienen que ser necesariamente disjuntas. En el siguiente ejemplo la selección se realiza tras finalizar la actividad A0. La actividad A1 se ejecuta si el importe es mayor que 100. Además, si es mayor que 200, se ejecuta la actividad A2 en paralelo con A1:

```
activity A1 {
    transition inicio {
       domain: {A1.estado = parado & A0.estado = finalizado
              & factura.importe > 100}
       action: {A1.estado = ejecutando}
    }
    ...
}
activity A2 {
    transition inicio {
       domain: {A2.estado = parado & A0.estado = finalizado
              & factura.importe > 200}
       action: {A2.estado = ejecutando}
    }
    ...
}
```

Patrón 7: mezcla sincronizada

En un punto del proceso varias ramas paralelas convergen en un único hilo, y deben ser sincronizadas. Cada rama puede haber sido activada o no (mediante primitivas de selección anteriores). Si hay más de una rama activa, las ramas activas deben ser sincronizadas. Esto quiere decir que la siguiente actividad en el hilo sincronizado se podrá ejecutar cuando hayan convergido todas las ramas activas. Si sólo hay una activa (independientemente del número de ramas no activas que haya) no es necesario sincronizar. Se asume que una rama activa no puede ser activada de nuevo mientras no se haya realizado la sincronización correspondiente a la activación anterior.

Solución La solución a este patrón es similar a la del patrón 3 (sincronización). La diferencia está en que en este caso no se debe esperar a todos los hilos, sino solamente a las ramas activas. Para ello, se puede añadir un atributo *Booleano* a cada rama para indicar si está o no activa. Dado que la definición del patrón asume que un hilo no puede volver a activarse hasta que haya sido sincronizado, es suficiente con una variable por hilo. Una vez realizada la sincronización, es necesario restablecer el valor inicial en este atributo para permitir que los hilos puedan ser activados de nuevo en el futuro.

El siguiente ejemplo muestra la implementación de este patrón para tres hilos (con variables de activación `hilo1`, `hilo2` e `hilo3` en la entidad de control *activacion*). Las actividades finales de cada hilo son A1, A2 y A3. La sincronización se realiza en una actividad S:

```
activity S {
    transition inicio {
        domain: {S.estado = parado
                 & (A1.estado = finalizado | !activacion.hilo1)
                 & (A2.estado = finalizado | !activacion.hilo2)
                 & (A3.estado = finalizado | !activacion.hilo3)}
        action: {S.estado = finalizado & !activacion.hilo1
                 & !activacion.hilo2 & !activacion.hilo3}}
}
```

Solución conjunta a los patrones 6 y 7 Habitualmente aparecen asociados los patrones 6 y 7 en las definiciones de procesos de negocio. En el primero se realiza la selección múltiple, dando lugar a varios hilos de los cuales no todos tienen por qué estar activos. En el segundo se mezclan sincronizadamente los hilos activos. Si todos los hilos que parten de la selección múltiple se unen en la mezcla sincronizada, se puede envolver jerárquicamente todo el bloque de actividades en una única actividad que controle su ejecución. A continuación se desarrolla esto con un ejemplo.

En primer lugar, es necesario definir la entidad que almacene, para cada hilo, si dicho hilo está activo, inactivo o si todavía no se ha evaluado su condición asociada:

```
enttype S_activacion_t {
    A0: enum(no_evaluado, inactivo, activo);
    A1: enum(no_evaluado, inactivo, activo);
}
entity S_activacion: S_activacion_t;
```

A la actividad de control S se le añade un estado intermedio en su ciclo de vida (`activando`), en el cual se realiza la evaluación de la condición asociada a cada hilo. Una vez realizada esta evaluación para todos los hilos, se pasa a estado `ejecutando`. En este momento, todas las actividades que inicien hilos evaluados como activos pueden comenzar su ejecución. Finalmente, cuando la última actividad de todos los hilos activos finalice, se da por finalizada la actividad de control.

```
activity S {
    transition inicio {
        domain: {S.estado = parado & A0.estado = finalizado}
        action: {S.estado = activando & S_activacion.A1 = no_evaluado
                 & S_activacion.A2 = no_evaluado}
    transition activar_A1 {
        domain: {S.estado = activando & S_activacion.A1 = no_evaluado
                 & factura.importe > 100}
```

```
        action: {S_activacion.A1 = activo}}
    transition desactivar_A1 {
        domain: {S.estado = activando & S_activacion.A1 = no_evaluado
                 & !(factura.importe > 100)}
        action: {S_activacion.A1 = inactivo}}
    transition activar_A2 {
        domain: {S.estado = activando & S_activacion.A2 = no_evaluado
                 & factura.importe > 200}
        action: {S_activacion.A2 = activo}}
    transition desactivar_A2 {
        domain: {S.estado = activando & S_activacion.A2 = no_evaluado
                 & !(factura.importe > 200)}
        action: {S_activacion.A2 = inactivo}}
    transition ejecutar {
        domain: {S.estado = activando & S_activacion.A1 != no_evaluado
                 & S_activacion.A2 != no_evaluado}
        action: {S.estado = ejecutando}}
    transition fin {
        domain: {S.estado = ejecutando
                 & (A5.estado = finalizado | S_activacion.A1 = inactivo)
                 & (A6.estado = finalizado | S_activacion.A2 = inactivo)}
        action: {S.estado = parado}}
}
activity A1 {
    transition inicio {
       domain: {A1.estado = parado & S.estado = ejecutando
                & S_activacion.A1 = activo}
       action: {A1.estado = ejecutando}
    }
    ...
}
...
```

Patrón 8: mezcla múltiple

En un punto del proceso varias ramas paralelas convergen en un único hilo, sin sincronización. Si más de una de estas ramas ha sido activada, posiblemente de forma concurrente, la actividad siguiente a la mezcla debe ser activada una vez para cada uno de los hilos activos. Este patrón permite que varios hilos compartan un mismo comportamiento final común sin necesidad de redefinir una vez para cada hilo las actividades que modelan dicho comportamiento.

Solución El formalismo CFM no permite representar este patrón, porque no da soporte de forma natural a la ejecución concurrente de múltiples instancias de una misma actividad. Dado que el número de hilos es conocido en tiempo de diseño, se puede modelar este comportamiento en CFM redefiniendo las actividades comunes para cada uno de sus hilos. A pesar de ello, dado que no es una forma natural de resolver el patrón, consideraremos de todas formas que CFM no da soporte a este patrón.

Patrón 9: discriminador

El discriminador representa un punto en el proceso que espera la finalización del primer hilo entre varios antes de permitir la ejecución de la siguiente actividad. Desde el momento en que se comienza a ejecutar la siguiente actividad, se *ignora* la finalización del resto de los

hilos activos. Una vez todos los hilos han acabado, se reinicia el discriminador para permitir que se pueda volver a instanciar en el futuro (por ejemplo, dentro de un bucle).

Solución La solución a este patrón es relativamente sencilla en CFM. Se puede implementar mediante una actividad que se inicie cuando una de las actividades finales de los hilos o ramas de entrada finalice. Una vez comience su ejecución, la siguiente actividad puede comenzar su ejecución. Cuando hayan finalizado todos los hilos de entrada, se marca la actividad como finalizada. Se muestra a continuación un ejemplo de la implementación del discriminador. La actividad S representa el discriminador, las actividades A1, A2 y A3, la última actividad de cada uno de sus tres hilos de entrada, y la actividad A5, la actividad siguiente al discriminador.

```
activity S {
    transition inicio {
        domain: {S.estado = parado
                 & (A1.estado = finalizado | A2.estado = finalizado
                    | A3.estado = finalizado)}
        action: {S.estado = ejecutando}}
    transition fin {
        domain: {S.estado = ejecutando & A1.estado = finalizado
                 & A2.estado = finalizado & A3.estado = finalizado}
        action: {S.estado = parado}}
}
activity A5 {
    transition inicio {
        domain: {A5.estado = parado & S.estado != parado}
        action: {A5.estado = ejecutando}}
...
}
```

5.1.3. Patrones estructurales

Distintos sistemas de gestión de procesos de negocio imponen distintas restricciones a los procesos, que en ocasiones dificultan el modelado de determinados procesos de una forma natural. En este apartado se incluyen dos patrones que muestran algunas de estas restricciones.

Patrón 10: Ciclos arbitrarios

Un punto en el proceso de negocio en el cual una o más actividades se pueden ejecutar repetidamente, sin restricción en el número, localización o entrelazado de dichos puntos en el proceso. La diferencia de un ciclo arbitrario con un ciclo estructurado radica en que los ciclos estructurados tienen un único punto de entrada al bucle y un único punto de salida del bucle, y no pueden ser solapados (esto es, sólo puede haber solapes entre dos bucles si uno de ellos está anidado dentro del otro).

Solución En el apartado 4.4.9 se muestra la implementación de un ciclo estructurado. Sin embargo, no es suficiente para resolver un ciclo arbitrario, en el que puede haber varios puntos de entrada o salida. En CFM se pueden modelar este tipo de bucles siempre y cuando no haya instanciación múltiple de actividades. Esto es así, por ejemplo, si toda la estructura repetitiva se ejecuta siempre en un único hilo.

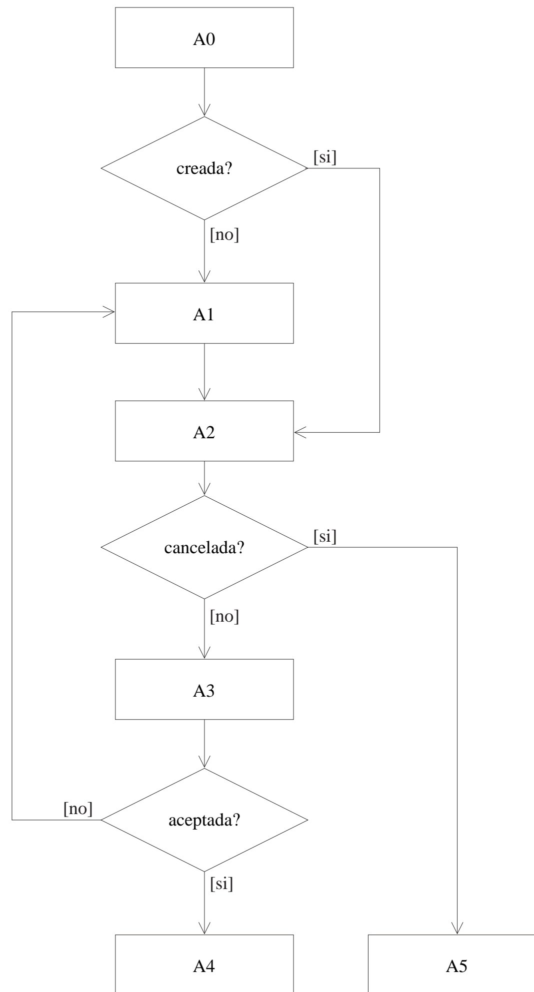

Figura 5.1: Ejemplo de diagrama de flujo de un ciclo arbitrario (patrón 10).

A continuación se muestra como ejemplo la implementación del proceso representado en la figura 5.1. En él, las actividades A1, A2 y A3 se ejecutan de forma repetitiva. Hay dos puntos de entrada al bucle y dos puntos de salida.

```
enttype oferta_t {
    creada: boolean;
    enviada: boolean;
    aceptada: boolean;
    cancelada: boolean;
    importe: abstract;
}
entity oferta: oferta_t;

activity A1 {
    transition inicio {
        domain: {A1.estado = parado
                 & ((A0.estado = finalizado & !oferta.creada)
                   |(A3.estado = finalizado & !oferta.aceptada)}
        action: {A1.estado = ejecutando & A2.estado = parado
                 & A3.estado = parado}}
```

```
    transicion fin {
        domain: {A1.estado = ejecutando}
        action: {A1.estado = finalizado & oferta.creada}}
}
activity A2 {
    transition inicio {
        domain: {A2.estado = parado &
                 A0.estado = finalizado & oferta.creada}
        action: {A2.estado = ejecutando}}
    transicion fin_enviada {
        domain: {A2.estado = ejecutando}
        action: {A2.estado = finalizado & oferta.enviada}}
    transicion fin_cancelado {
        domain: {A2.estado = ejecutando}
        action: {A2.estado = finalizado & oferta.cancelada}}
}
activity A3 {
    transition inicio {
        domain: {A3.estado = parado &
                 A2.estado = finalizado & ! oferta.cancelada}
        action: {A3.estado = ejecutando}}
    transicion fin_aceptada {
        domain: {A3.estado = ejecutando}
        action: {A3.estado = finalizado & oferta.aceptada}}
    transicion fin_cancelado {
        domain: {A2.estado = ejecutando}
        action: {A2.estado = finalizado & !oferta.aceptada
                 & A1.estado = parado}}
}
activity A4 {
    transition inicio {
        domain: {A4.estado = parado &
                 A3.estado = finalizado & oferta.aceptada}
        action: {A4.estado = ejecutando}}
...
}
activity A5 {
    transition inicio {
        domain: {A5.estado = parado &
                 A2.estado = finalizado & oferta.cancelada}
        action: {A5.estado = ejecutando}}
...
}
```

La transición funcional de inicio de la actividad A2 puede resultar algo confusa. Nótese que la condición `A0.estado = finalizado & oferta.creada` define los dos posibles caminos de entrada a esta actividad: directamente desde A0, y a través de A1, dado que, tras la ejecución de esta última actividad, la oferta está creada y A0 permanece en estado *finalizado*.

Patrón 11: finalización implícita

Un determinado proceso o subproceso debería ser finalizado cuando no haya nada más por hacer, esto es, cuando no haya actividades en ejecución y ninguna otra actividad pueda ser activada.

Solución Por construcción del formalismo CFM, un proceso que no pueda evolucionar, esto es, ninguna transición funcional pueda ser activada, está en un estado final. Este es el

caso cuando todas las actividades que hayan comenzado a ser ejecutadas están finalizadas y ninguna otra puede ser ejecutada. Por tanto, CFM da soporte a este patrón.

En algunos casos resulta conveniente crear una actividad que represente al proceso completo y lo envuelva jerárquicamente. En este caso, se puede definir una transición funcional para este proceso que marque su subestado como finalizado cuando ninguna otra actividad pueda ser ejecutada.

5.1.4. Patrones con instancias múltiples

Los patrones de este grupo están relacionados con la existencia de *instancias múltiples*. Desde un punto de vista teórico, esto se corresponde con varios hilos de ejecución que comparten una misma definición.

Patrón 12: instancias múltiples sin sincronización

En el contexto de un único caso (o instancia del proceso), se pueden crear múltiples instancias de la misma actividad, creando nuevos hilos de control que se pueden ejecutar de forma concurrente. Cada hilo se comporta de forma independiente a los otros hilos. No es necesario que los hilos sean sincronizados.

Solución El formalismo CFM no da soporte de forma natural a este patrón, pero sí permite implementarlo mediante el lanzamiento de una instancia independiente de otro proceso. Según la discusión realizada en [9], esta es una solución válida para este patrón. Su inconveniente está en que el nuevo proceso se ejecutará de forma independiente al proceso inicial.

Cuando se realice la instanciación del proceso nuevo (llamémosle B), el proceso inicial (llamémosle A) podrían establecer valores iniciales en los atributos del B. A partir de este momento la ejecución del proceso B sería independiente de la ejecución del proceso A y del resto de instancias que se creen o hayan creado de este proceso B. En todo caso, nuestra opinión es que de esta forma no se está dando soporte total al patrón, sino únicamente parcial, dado que la solución que proponemos es muy restrictiva.

Patrón 13: instancias múltiples con conocimiento a priori en tiempo de diseño

Una actividad de una misma instancia del proceso se puede ejecutar múltiples veces. El número de instancias de cada actividad para una misma instancia del proceso es conocido en tiempo de diseño. Puede ocurrir que otra actividad esté esperando para ser ejecutada a que finalicen todas las instancias iniciadas de la actividad.

Solución Si se conoce el número de instancias, basta con replicar la actividad en cuestión, y la entidad que representa su ciclo de vida, tantas veces como sea necesario.

Patrón 14: instancias múltiples con conocimiento a priori en tiempo de ejecución

Una actividad de una misma instancia del proceso se puede ejecutar múltiples veces. El número de instancias es variable y puede depender de características de la instancia concreta del proceso, pero es conocido en tiempo de ejecución, antes de que las instancias de las

actividades tengan que ser creadas. Puede ocurrir que otra actividad esté esperando para ser ejecutada a que finalicen todas las instancias iniciadas de la actividad.

Solución El formalismo CFM no da soporte a este patrón, porque no permite la instanciación múltiple de una misma actividad.

Patrón 15: instancias múltiples sin conocimiento a priori en tiempo de ejecución

Una actividad de una misma instancia del proceso se puede ejecutar múltiples veces. El número de instancias es variable y no se conoce en tiempo de diseño ni de ejecución, antes de que las instancias sean creadas. Puede ocurrir que otra actividad esté esperando para ser ejecutada a que finalicen todas las instancias iniciadas de la actividad. Se diferencia del patrón 14 en que se pueden crear más instancias de la actividad durante o después de la ejecución de otras instancias de la misma.

Solución El formalismo CFM no da soporte a este patrón, porque no permite la instanciación múltiple de una misma actividad.

5.1.5. Patrones basados en estado

En algunas situaciones es necesario modelar de forma explícita el estado de un proceso. Los patrones de este grupo representan situaciones de este tipo.

Patrón 16: decisión aplazada

En un punto del proceso de negocio en el cual una de entre varias ramas es seleccionada, no se realiza la decisión de forma explícita (basada en datos o en una decisión), sino que el entorno ofrece varias alternativas. Sólo una de estas alternativas puede ser ejecutada. La decisión se aplaza hasta el mismo momento en que una de las ramas se active, momento en el cual se desactivan las demás. Por tanto, la decisión se toma tan tarde como sea posible. Este patrón se diferencia del patrón 4 en que en este último se podía tomar la decisión durante la ejecución de la actividad anterior a la división condicional. Este patrón sirve, por ejemplo, para el modelado de eventos. Si el sistema está en espera de recibir varios eventos, cada uno procesado en una rama alternativa, no debe tomar una decisión hasta el mismo momento en que se reciba uno de ellos.

Solución El formalismo CFM da soporte a este patrón. En el siguiente ejemplo, las actividades A1 y A2 se ejecutan después de haber finalizado la actividad A0, cuando ocurra un determinado evento. Este evento determina cuál de las dos ramas de ejecución se selecciona. Se modela el evento de forma abstracta, mediante la introducción de no–determinismo (véase el apartado 4.4.11).

```
activity A1 {
    transition inicio {
                 /* evento 1 */
        domain: {A1.estado = parado & A0.estado = finalizado
                 & A2.estado = parado}
```

```
        action: {A1.estado = ejecutando}}
...
}
activity A2 {
    transition inicio {
                 /* evento 2 */
        domain: {A2.estado = parado & A0.estado = finalizado
                 & A1.estado = parado}
        action: {A2.estado = ejecutando}}
...
}
```

Patrón 17: encaminamiento paralelo alternante

Un conjunto de actividades se ejecutan en orden arbitrario. Cada actividad del conjunto se ejecuta una vez, el orden se decide en tiempo de ejecución, y en ningún instante de tiempo puede haber dos actividades del conjunto en ejecución (para una instancia del proceso dada). Este patrón se diferencia del patrón 1 (ejecución en secuencia) en que se permite que las actividades sean ejecutadas en cualquier orden.

Solución El formalismo CFM da soporte a este patrón. Para cada actividad del conjunto se define el dominio de su transición funcional de inicio de tal forma que se pueda ejecutar sólo si ninguna otra actividad del mismo está en ejecución. Si ninguna está en ejecución, se ejecutará cualquiera de las actividades que todavía no hayan sido ejecutadas, seleccionada de forma no–determinista. En el siguiente ejemplo las actividades A1, A2 y A3 se ejecutan de acuerdo a este patrón. La ejecución comienza cuando la actividad A0 finaliza. Se muestra también la actividad A4, que se puede ejecutar cuando todas las actividades del conjunto hayan finalizado su ejecución.

```
activity A1 {
    transition inicio {
        domain: {A0.estado = finalizado & A1.estado = parado
                 & !A2.estado = ejecutando & !A3.estado = ejecutando}
        action: {A1.estado = ejecutando}}
...
}
activity A2 {
    transition inicio {
        domain: {A0.estado = finalizado & A2.estado = parado
                 & !A1.estado = ejecutando & !A3.estado = ejecutando}
        action: {A2.estado = ejecutando}}
...
}
activity A3 {
    transition inicio {
        domain: {A0.estado = finalizado & A3.estado = parado
                 & !A1.estado = ejecutando & !A2.estado = ejecutando}
        action: {A3.estado = ejecutando}}
...
}
activity A4 {
    transition inicio {
        domain: {A4.estado = parado & A1.estado = finalizado
                 & A2.estado = finalizado & A3.estado = finalizado}
        action: {A4.estado = ejecutando}}
...
}
```

Patrón 18: hito

Una actividad del proceso es ejecutable sólo si el proceso está en un determinado *estado*. Esto es, es ejecutable si el proceso ha alcanzado un determinado *hito* sin que este haya caducado. Se representa un *hito* como un estado asociado a que el proceso haya finalizado una determinada actividad pero no haya iniciado la ejecución de la actividad que la sigue en una secuencia.

Solución El formalismo CFM también es capaz de dar soporte a este patrón, de forma explícita o implícita. La forma explícita consiste en definir un atributo *Booleano* que tome valor *cierto* cuando se alcance el hito y valor *falso* cuando caduque el hito. La forma implícita consiste en, en lugar de definir un nuevo atributo, modelar el patrón mediante la variable de estado de las dos actividades que definen a dicho hito.

En el siguiente ejemplo se modela una actividad A3 cuya actividad puede iniciarse sólo si el proceso se encuentra en el hito delimitado por las actividades A1 y A2. Este hito se modela de forma implícita. Nótese que no siempre se ejecutará la actividad A3: puede ocurrir que el hito caduque (esto es, comience la ejecución de la actividad A2).

```
activity A1 {
    transition inicio {
        domain: {A1.estado = parado & A0.estado = finalizado}
        action: {A1.estado = ejecutando}}
...
}
activity A2 {
    transition inicio {
        domain: {A2.estado = parado & A1.estado = finalizado}
        action: {A2.estado = ejecutando}}
...
}
activity A3 {
    transition inicio {
        domain: {A3.estado = parado & A1.estado = finalizado
                & A2.estado = parado}
        action: {A3.estado = ejecutando}}
...
}
```

5.1.6. Patrones de cancelación

En este grupo se incluyen dos patrones de cancelación de instancias de actividades y de instancias de procesos.

Patrón 19: cancelar actividad

Se deshabilita una actividad cuya ejecución está permitida, esto es, se elimina un hilo que está en espera de la ejecución de una actividad. Este patrón es útil, por ejemplo, para cancelar actividades que estén a la espera de ocurrencia de eventos, de tal forma que los eventos no sean procesados.

Solución Se puede modelar este patrón en CFM mediante la utilización de un atributo adicional *Booleano* cuyo valor sea *cierto* si la actividad debe ser cancelada. En el dominio

de la transición de inicio de la actividad se establece que dicho atributo debe tener valor falso para que la actividad pueda ser ejecutada. Si desde algún punto del procesos se desea cancelar la actividad, basta con establecer un valor *cierto* en este atributo.

En el siguiente ejemplo se muestra una actividad A1 que inicia un hilo y permanece en espera de que ocurra un evento, y una actividad A2 que, en una de sus transiciones, cancela la ejecución del hilo de A1.

```
enttype cancelacion_t {
    cancelado: boolean;
}
entity A1_cancel: cancelacion_t;

activity A1 {
    transition inicio {
        domain: {A1.estado = parado & A0.estado = finalizado
                 & !A1_cancel.cancelado}
        action: {A1.estado = ejecutando}}
    transition cancelar {
        domain: {A1.estado != cancelado & A1.estado != finalizado
                 & A1_cancel.cancelado}
        action: {A1.estado = cancelado}}
...
}
activity A2 {
...
    transition cancelar_A1 {
        domain: {...}
        action: {A1_cancel.cancelado}}
...
}
```

Patrón 20: cancelar caso

Se cancela por completo un caso o instancia del proceso, esto es, se eliminan todas las instancias de actividades pertenecientes a dicho caso.

Solución El formalismo CFM permite la cancelación de una instancia del proceso. Para ello, se añade un atributo *Booleano* cuyo valor sea cierto si el proceso debe ser cancelado. La actividad desde la cual se quiera cancelar el proceso debe establecer valor *cierto* en esta variable. Existen varias formas de implementar este patrón. Una de ellas consiste en añadir una transición funcional de cancelación a cada actividad. En el siguiente ejemplo se muestra una solución en que la actividad A1 puede ser cancelada de forma aislada (patrón 19) o con todo el proceso. La actividad A2 inicia la cancelación del proceso.

```
enttype cancelacion_t {
    cancelado: boolean;
}
entity A1_cancel: cancelacion_t;
entity case_cancel: cancelacion_t;

activity A1 {
    transition inicio {
        domain: {A1.estado = parado & A0.estado = finalizado
                 & !A1_cancel.cancelado & !case_cancel.cancelado}
        action: {A1.estado = ejecutando}}
    transition cancelar {
        domain: {A1.estado != cancelado & A1.estado != finalizado
```

```
                 & (A1_cancel.cancelado | case_cancel.cancelado)}
        action: {A1.estado = cancelado}}
...
}
activity A2 {
...
    transition cancelar_A1 {
        domain: {...}
        action: {case_cancel.cancelado}}
...
}
```

5.1.7. Conclusión

En este apartado se ha mostrado que el formalismo CFM es capaz de modelar todos los patrones de *workflow* excepto tres: los patrones 8, 14, y 15. Por otra parte, el soporte al patrón 12 es sólo parcial, dadas las restricciones que impone a la instanciación múltiple de bloques de actividades. La causa de todas estas limitaciones es única: el formalismo CFM no permite crear múltiples instancias de una misma actividad de forma concurrente si no se conoce en tiempo de diseño el número de instancias. Sin embargo, sí da soporte a múltiples instancias de una misma actividad siempre y cuando en un instante de tiempo dado pueda haber sólo una activa. Esta limitación se debe a que sólo existe una instancia de cada uno de los atributos de control que gestionan el estado de una actividad. Como se muestra en el próximo apartado, en que se comparan estos resultados con los de otros lenguajes de definición de procesos, esta es una limitación que presentan numerosos lenguajes.

5.2. Comparación con otros lenguajes de definición de procesos

Los patrones de *workflow* son una herramienta útil para comprar la adecuación y expresividad del formalismo CFM con los de otros lenguajes de definición de procesos de *workflow*. En los cuadros 5.1, 5.2 y 5.3 se compara CFM con otros lenguajes de definición de procesos de negocios y composiciones de servicios Web. Los datos de los análisis de estos lenguajes, excepto del propio CFM, han sido extraídos sin modificación alguna de [125, 6, 8]. En el cuadro se representa con "+" el hecho de que el lenguaje dé soporte al patrón, con "-" que no dé soporte al patrón y con "+/-" que dé sólo soporte de forma parcial al patrón. En los próximos apartados se analizan estos resultados.

En el cuadro 5.1 se muestra una comparación de CFM con otros cuatro lenguajes que, en general, pueden ser considerados lenguajes de composición de servicios Web: BPEL4WS [14], WSFL [89], XLANG [95] y BPML [15].

Si se compara el formalismo CFM con BPEL4WS, se observa que todos los patrones modelables con este lenguaje lo son también con CFM. De todas formas, el modelado del patrón 12 (instancias múltiples sin sincronización), es más potente y sencillo en BPEL4WS. En este lenguaje se puede implementar este patrón definiendo un proceso auxiliar para el bloque de código a ser instanciado múltiples veces. Cada instancia de este nuevo proceso se puede crear mediante el envío de un un mensaje desde el proceso principal (actividad `invoke`), y la recepción en el proceso auxiliar (actividad `receive`) con el atributo `createInstance` activado. Los resultados de este análisis no sirven para demostrar que todo proceso BPEL4WS sea representable mediante CFM, dado que existen funcionalida-

Patrón	CFM	BPEL4WS	XLANG	WSFL	BPML
1 (seq)	+	+	+	+	+
2 (par-spl)	+	+	+	+	+
3 (synch)	+	+	+	+	+
4 (ex-ch)	+	+	+	+	+
5 (simple-m)	+	+	+	+	+
6 (m-choice)	+	+	-	+	-
7 (sync-m)	+	+	-	+	-
8 (multi-m)	-	-	-	-	+/-
9 (disc)	+	-	-	-	-
10 (arb-c)	+	-	-	-	-
11 (impl-t)	+	+	-	+	+
12 (mi-no-s)	+/-	+	+	+	+
13 (mi-dt)	+	+	+	+	+
14 (mi-rt)	-	-	-	-	-
15 (mi-no)	-	-	-	-	-
16 (def-c)	+	+	+	-	+
17 (int-par)	+	+/-	-	-	-
18 (milest)	+	-	-	-	-
19 (can-a)	+	+	+	+	+
20 (can-c)	+	+	+	+	+

Cuadro 5.1: Comparación de CFM con BPEL4WS, XLANG, WSFL y BPML. Fuentes: [125, 6].

des en BPEL4WS no representables mediante estos patrones. Sin embargo, los resultados sí suponen un respaldo a la capacidad expresiva de CFM. En el capítulo 7 se profundiza en la representación de procesos BPEL4WS mediante CFM.

Si se compara con los lenguajes WSFL y XLANG, se puede observar que todos los patrones modelables con ellos son también modelables con el formalismo CFM. Lo mismo ocurre con BPML, con la salvedad del patrón 8 (mezcla múltiple), cuyo soporte, según se describe en [6], es sólo parcial en BPML.

En el cuadro 5.2 se muestra la comparativa con XPDL [127], que es el lenguaje de definición de procesos estandarizado por el WfMC, el formalismo de redes de Petri de alto nivel, tal y como se define en [2], y YAWL [8], un lenguaje basado en redes de Petri y diseñado con la intención de solucionar las limitaciones encontradas en este formalismo para el modelado de algunos patrones de *workflow*.

En este cuadro se puede ver que la potencia expresiva de CFM, en términos de patrones de *workflow*, es mayor que la de XPDL y redes de Petri de alto nivel, con una excepción. El patrón 8 (mezcla múltiple) se puede modelar con redes de Petri, pero no mediante CFM.

Sin embargo, se puede apreciar también que la potencia expresiva de YAWL es superior a la de CFM. La diferencia está en la capacidad de YAWL para modelar instancias múltiples de actividades. Hay que tener en cuenta que YAWL ha sido diseñado, precisamente, a partir del análisis basado en patrones, tomando como formalismo de partida las redes de Petri de alto nivel y enriqueciéndolo con la funcionalidad necesaria para dar soporte al resto de patrones

Patrón	CFM	XPDL	HL–PN	YAWL
1 (seq)	+	+	+	+
2 (par-spl)	+	+	+	+
3 (synch)	+	+	+	+
4 (ex-ch)	+	+	+	+
5 (simple-m)	+	+	+	+
6 (m-choice)	+	+	+	+
7 (sync-m)	+	+	-	+
8 (multi-m)	-	-	+	+
9 (disc)	+	-	-	+
10 (arb-c)	+	+	+	+
11 (impl-t)	+	+	-	-
12 (mi-no-s)	+/-	+	+	+
13 (mi-dt)	+	+	+	+
14 (mi-rt)	-	-	-	+
15 (mi-no)	-	-	-	+
16 (def-c)	+	-	+	+
17 (int-par)	+	-	+	+
18 (milest)	+	-	+	+
19 (can-a)	+	-	+/-	+
20 (can-c)	+	-	-	+

Cuadro 5.2: Comparación de CFM con XPDL, redes de Petri de alto nivel (HL-PN) y YAWL. Fuentes: [8].

(instanciación múltiple, sincronizaciones complejas y vuelta hacia atrás no local [8]).

En el cuadro 5.3 se muestra la comparación de CFM con otros lenguajes analizados en [8]. Se puede apreciar que la potencia expresiva de CFM desde el punto de vista de los patrones de *workflow* es, en general, superior a la de estos lenguajes. Los lenguajes analizados son ADEPT-flex[114], OPENflow [65], EPC, Mentor [102], CTR [119], Meteor [97], Mobile [75], WASA [122], Exotica [100] y CFs [35].

5.3. Conclusiones

En este capítulo se ha analizado la capacidad expresiva del formalismo CFM desde el punto de vista de la perspectiva de control, mediante un estudio basado en patrones de *workflow*. De este estudio se puede concluir que la capacidad expresiva de CFM es superior a la de la mayoría de los lenguajes analizados, con la excepción de YAWL. Se ha identificado que la principal limitación de CFM es la imposibilidad para modelar múltiples instancias de una misma actividad que se ejecuten concurrentemente. De todas formas, como se puede apreciar en las tablas mostradas en apartados anteriores, esta es una limitación común a muchos lenguajes de definición de procesos de negocio y composiciones de servicios Web.

Dados los resultados de YAWL con respecto a este análisis, cabe la posibilidad de integrar este lenguaje, y no CFM, en la capa 2 de la arquitectura propuesta en esta tesis. YAWL,

Patrón	CFM	ADEPT-flex	OPENflow	EPC	Mentor	CTR	Meteor	Mobile	WASA	Exotica	CFs
1 (seq)	+	+	+	+	+	+	+	+	+	+	+
2 (par-spl)	+	+	+	+	+	+	+	+	+	+	+
3 (synch)	+	+	+	+	+	+	+	+	+	+	+
4 (ex-ch)	+	+	+	+	+	+	+	+	+	+	+
5 (simple-m)	+	+	+	+	+	+	+	+	+	+	+
6 (m-choice)	+	-	+	+	-	+	+	+	+	+	-
7 (sync-m)	+	+	-	+	-	+	-	-	+	+	-
8 (multi-m)	-	-	-	-	-	-	+	-	-	-	-
9 (disc)	+	+	-	-	+	-	+/-	+	-	-	-
10 (arb-c)	+	-	-	+	-	-	+	-	-	-	-
11 (impl-t)	+	-	-	+	-	-	-	-	+	+	+
12 (mi-no-s)	+/-	-	-	-	-	-	+	-	-	-	-
13 (mi-dt)	+	+	+	+	+	+	+	+	+	+	+
14 (mi-rt)	-	-	-	-	-	-	-	-	-	-	-
15 (mi-no)	-	-	-	-	-	-	-	-	-	-	-
16 (def-c)	+	+	-	-	+	+	-	-	-	-	-
17 (int-par)	+	-	-	-	-	-	-	+	-	-	-
18 (milest)	+	-	-	-	+/-	-	-	-	-	-	-
19 (can-a)	+	-	-	-	+	-	-	-	-	-	-
20 (can-c)	+	-	-	-	+	-	-	-	-	-	-

Cuadro 5.3: Comparación de CFM con otros lenguajes de definición de procesos de negocio presentados en la literatura. Fuentes: [8].

al igual que CFM, está formalmente definido y podría ser verificado mediante distintas herramientas de verificación. Por otra parte, posee una gran capacidad de expresividad, que hace factible la transformación de procesos de la capa 3 a YAWL. La principal desventaja de YAWL frente a CFM está en que no ha sido diseñado para modelar la perspectiva de datos de los procesos. Esto supone una gran limitación de cara a representar procesos modelados, por ejemplo, con BPEL4WS o BPML, que incorporan como parte fundamental del lenguaje el concepto de *variable*.

Por tanto, tras estudiar los resultados de este análisis, consideramos que el formalismo CFM tiene una capacidad expresiva suficiente, desde el punto de vista de la perspectiva de control de flujo, para ser integrado como formalismo común en la capa 2 de nuestra arquitectura.

Capítulo 6

Verificación

Concurrency–related problems are not rare oddities
that appear only in obscure corners of software engineering.

(Gerald J. Holzmann, "The Spin Model Checker", pág. 5, 2004)

Tal y como se expone en el apartado 2.1, la verificación de requisitos funcionales en la definición de un proceso consiste en desarrollar una especificación del mismo, como un conjunto de requisitos, y comprobar si la definición cumple dichos requisitos. En otras palabras, consiste en proporcionar una lista de requisitos que el diseñador espera que se cumplan en la definición del proceso, y demostrar si dichos requisitos, aplicados a la definición, son ciertos o falsos.

En la arquitectura propuesta en el capítulo 3 se propone aplicar herramientas de verificación existentes a los procesos definidos mediante el formalismo CFM. Para ello, es necesario transformar estas definiciones de procesos, y sus especificaciones, del formalismo CFM al lenguaje de entrada de las herramientas de verificación empleadas. Por otra parte, resultaría útil, aunque no imprescindible, proporcionar una transformación en sentido inverso de los resultados de la verificación al dominio de la definición CFM.

En primer lugar, en el apartado 6.1, se analizan otros trabajos relacionados para analizar los tipos de requisitos que permiten verificar.

A partir de los resultados de este análisis, en el apartado 6.2, se discute qué tipos de requisitos podría ser interesante o factible verificar en una definición CFM. Se profundiza tanto en requisitos generales, válidos para una gran mayoría de procesos, como en requisitos específicos, definidos a medida para procesos concretos.

Posteriormente, en los apartados 6.3 y 6.4, se propone detalladamente cómo integrar las herramientas de *model checking* Spin [72, 73] y NuSMV [29]. Para ello, se propone un mecanismo automático de transformación de definiciones de procesos y especificaciones CFM a los lenguajes de entrada de ambas herramientas.

En el capítulo 7 se concretan los principios expuestos en este apartado para un lenguaje de definición de procesos concreto: BPEL4WS, con el objetivo de complementar esta

exposición y mostrar sus posibilidades de aplicación.

6.1. Definición de especificaciones en trabajos relacionados

En este apartado se analizan los tipos de requisitos propuestos para la verificación de procesos de negocio en otros trabajos relacionados.

Aalst define en [2] la propiedad de *solidez* (del inglés, *soundness*) como una propiedad importante para comprobar si un proceso de negocio, definido mediante *redes de workflow*, está correctamente construido. Las redes de *workflow*, o *WF–nets*, son el formalismo propuesto por Aalst, basado en redes de Petri de alto nivel. Dado que la definición original de esta propiedad es específica para dicho formalismo, se adapta a continuación su definición para que sea aplicable a cualquier formalismo de definición de procesos de negocio basado en actividad. Se puede decir que una definición de un proceso es *sólida* si verifica las siguientes propiedades:

- *Para toda instancia, el proceso finalizará eventualmente y, una vez finalizado, no será posible ejecutar ninguna actividad.* Esta propiedad garantiza que el proceso esté libre de situaciones de *abrazo mortal*. Además, garantiza que, una vez finalizado el proceso, no quede ninguna actividad con posibilidades de ser ejecutada. El término *finalización del proceso* es abstracto. Su significado concreto depende del formalismo o lenguaje de definición de procesos al cual se aplique.

- *No debe haber actividades muertas, esto es, para cada actividad definida en el proceso debería existir al menos un camino de ejecución en que dicha actividad sea ejecutada.* Esta propiedad garantiza que no existe ninguna actividad definida en el proceso cuya ejecución sea imposible.

En la definición original, se añade también que debe existir un único lugar (según su significado en redes de Petri) de entrada y un único lugar de salida, y que toda actividad o condición debe estar en un camino desde el lugar de entrada hacia el lugar de salida. Dado que son propiedades específicas para redes de Petri, y además esta última condición se corresponde, en parte, con la de ausencia de actividades muertas, no se tienen en cuenta en este análisis.

Eshuis también desarrolla en [41] el tema de verificación de requisitos funcionales de procesos de negocio, especificados mediante las semánticas formales definidas para adaptar los diagramas de actividad de UML a este dominio de aplicación. Clasifica los requisitos a verificar en dos grupos: *requisitos ad–hoc* y *requisitos generales*. Los requisitos generales deben cumplirse para cualquier definición de proceso de negocio, mientras que los requisitos *ad–hoc* son específicos de una definición de proceso de negocio concreta.

Los requisitos *ad–hoc* que el autor propone a modo de ejemplo comprueban propiedades temporales LTL basadas en la ejecución o no ejecución de determinadas actividades del proceso. Por ejemplo, dado un proceso de negocio de procesado y envío de pedidos de clientes, se comprueban propiedades del estilo de: *se envía factura al cliente si y sólo si se produce o extrae del almacén algún bien*. El autor define esta propiedad con la siguiente fórmula CTL*, expresada mediante una sintaxis definida específicamente para su trabajo:

$$\mathbf{F}(in(Produce) \lor in(Fillorder)) \iff \mathbf{F}in(Sendbill) \tag{6.1}$$

El autor propone una lista de cuatro requisitos generales para todo proceso. Dado que los requisitos se enuncian en dicho trabajo específicamente para el formalismo propuesto, se exponen a continuación de forma más abstracta, para que resulten adaptables a otros lenguajes y formalismos:

- *Todos los caminos deben alcanzar eventualmente un estado final*. Este requisito es similar a uno de los propuestos por Aalst.

- *Ausencia de actividades muertas y de transiciones muertas entre actividades.* Este requisito es también similar a uno de los planteados por Aalst. Dadas las características del formalismo de Eshuis, este requisito se expresa en realidad mediante dos requisitos distintos.

- La tercera propiedad es demasiado específica para el formalismo de Eshuis. Especifica que el diagrama de actividad no puede divergir, esto es, siempre alcanzará un estado estable en algún momento futuro para todos los caminos. No tiene sentido aplicar este concepto de estabilidad a un proceso de negocio general.

En resumen, los requisitos propuestos por Eshuis son bastante similares a los definidos por Aalst bajo el concepto de *solidez*. Sin embargo, Eshuis permite la definición de propiedades *ad–hoc*, lo cual no se contempla en el trabajo de Aalst.

Foster propone en [55] varios tipos de propiedades generales que todas las definiciones de composiciones de procesos BPEL4WS deberían verificar. Su diferencia con los trabajos anteriores está en que son propiedades diseñadas no para la verificación de un proceso aislado, sino para la composición de procesos. Clasifica estas propiedades en tres grupos:

- Compatibilidad de interfaz: la semántica de los mensajes enviados y recibidos por los distintos procesos que se componen debe ser compatible.

- Compatibilidad de seguridad: la composición debe estar libre de *abrazo mortal*, principalmente[1].

- Compatibilidad de viveza: la composición debe, eventualmente, finalizar. Los mensajes recibidos son procesados en orden FIFO.

6.2. Definición de especificaciones en CFM

De los trabajos relacionados presentados anteriormente se pueden extraer varias ideas que resultaría interesante utilizar para definir especificaciones en el formalismo CFM. Principalmente, son la distinción entre requisitos generales y específicos, así como el conjunto de requisitos extraídos de la definición de *solidez* proporcionada por Aalst.

Una especificación se define como un conjunto de requisitos. Según su ámbito de aplicación, se puede clasificar estos requisitos en tres categorías: requisitos generales, requisitos específicos de una metodología de transformación y requisitos específicos.

[1] Pese a que el autor clasifica la ausencia de *abrazo mortal* como propiedad de seguridad, esta propiedad es, en realidad, de viveza.

Los requisitos generales son aplicables a todo proceso definido mediante el formalismo CFM. Fundamentalmente, comprueban que la estructura de los procesos sea correcta. En el apartado 6.2.4 se profundiza en este tipo de requisitos.

Los requisitos específicos de una metodología de transformación son comunes a todos los procesos CFM obtenidos mediante la aplicación de una misma metodología de transformación. Por ejemplo, los requisitos de esta categoría serían aplicables a todas las definiciones CFM obtenidas a partir de procesos BPEL4WS mediante la metodología de transformación expuesta en el capítulo 7. Estos requisitos deben ser definidos, si procede, con la propia definición de la metodología de transformación.

Los requisitos específicos son aquellos asociados a una definición de un proceso concreta. El diseñador del proceso puede definir sus propios requisitos, basándose en los tipos de requisitos permitidos en el formalismo CFM.

En este apartado se desarrolla una propuesta para la definición de requisitos para especificaciones basadas en el formalismo CFM. En primer lugar, se introduce la notación en que se basa la definición de los distintos requisitos y tipos de requisitos. A continuación, se define formalmente el concepto de *especificación*, tal y como se entiende en este trabajo. Posteriormente, se proponen y definen distintos tipos de requisitos básicos para construir especificaciones. Finalmente, se proponen requisitos generales que toda definición de un proceso CFM debería cumplir.

6.2.1. Notación

Con el objetivo de definir los requisitos que forman parte de especificaciones en el formalismo CFM, se utiliza la siguiente notación, basada en el sistema formal básico de este formalismo.

El hecho de que un requisito q dado se cumpla para un proceso P se denota como:

$$P \models q \tag{6.2}$$

El hecho de que un predicado *Booleano* p, función de los atributos que componen el estado de un proceso, se evalúe con valor *cierto* para un estado s dado se denota como:

$$s \models p \tag{6.3}$$

El hecho de que una determinada propiedad p, expresada mediante una lógica temporal, se evalúe con valor *cierto* para un proceso P se denota como:

$$P \models p \tag{6.4}$$

Los predicados *Booleanos* se expresan mediante los operadores habituales de lógica de proposiciones: $\wedge, \vee, \neg$, $\implies$, $\iff$, etc. Los predicados de lógica temporal se expresan combinando los operadores anteriores con los operadores propios de lógicas temporales: **A**, **E**, **F**, **G**, **X**, **U** y **R**.

Por otra parte, se denota el conjunto de todas las actividades definidas para un proceso como ACT. Si una transición funcional f pertenece a una determinada actividad a, se denota como $f \in a$, cometiendo un pequeño abuso de notación.

6.2.2. Estructura de una especificación

Una especificación Q del formalismo CFM se define como un conjunto de requisitos. El hecho de que un proceso P sea conforme a una especificación Q se denota como $P \models Q$ y se define como:

$$P \models Q \iff \forall q \in Q,\ P \models q \tag{6.5}$$

6.2.3. Tipos de requisitos

Una especificación CFM puede contener requisitos de distintos tipos. En concreto, proponemos los siguientes:

- *Invariantes.*
- *Objetivos.*
- *Pre–requisitos* y *post–requisitos* de transiciones funcionales.
- *Ejecutabilidad* de transiciones funcionales.
- *Requisitos expresados mediante lógicas temporales.*

El concepto de *invariante* aparece habitualmente en la literatura. Se trata de un predicado *Booleano*, expresado en términos de las propiedades de un estado del proceso, que debe ser evaluado con valor *cierto* en todos los estados alcanzables por el proceso. Si la invariante se denota como $q_{inv}\,(p)$, donde p es este predicado:

$$P \models q_{inv}\,(p) \iff \forall e \in E(P),\ \forall s \in e,\ s \models p \tag{6.6}$$

Recuérdese que $E(P)$ denota el conjunto de todas las posibles ejecuciones del proceso P.

La propiedad equivalente a una invariante, utilizando notación de lógica temporal, es la siguiente:

$$P \models q_{inv}\,(p) \iff P \models \mathbf{AG}p \tag{6.7}$$

El concepto de *objetivo* consiste en un predicado *Booleano*, expresado en términos de las propiedades de un estado del sistema, que debe ser evaluado con valor *cierto* en todos los estados finales del proceso. El objeto de este tipo de requisito es especificar qué debería ocurrir al finalizar el proceso. Si puede finalizar en algún estado en que la propiedad se evalúe a valor *falso*, entonces no se cumple el requisito. Esta propiedad permite detectar, por ejemplo, situaciones de *abrazo mortal*, dado que un abrazo mortal provoca que se alcance un estado final no esperado por el diseñador que, en general, no debería verificar la propiedad *objetivo*. Si el objetivo se denota como $q_{obj}\,(p)$, donde p denota este predicado:

$$P \models q_{obj}\,(p) \iff \forall e \in E(P),\ final(e) \models p \tag{6.8}$$

En realidad, dado que el conjunto formado por los estados finales de todas las ejecuciones no es más que el conjunto de estados finales del proceso, denotado como S_f, se puede también enunciar esta propiedad como:

$$P \models q_{obj}\,(p) \iff \forall s \in S_f,\ s \models p \iff S_f \subseteq \{s\ /\ s \models p\} \tag{6.9}$$

Dado que mediante lógicas temporales no se puede modelar el concepto de estado final, se puede definir esta propiedad, suponiendo que una vez alcanzado un estado final este se repite infinitas veces, como:

$$P \models q_{obj}\,(p) \iff P \models \mathbf{AF}(\mathbf{AG}p) \tag{6.10}$$

Los *pre–requisitos* y *post–requisitos* de una transición funcional son predicados *Booleanos*, expresados en términos de las propiedades de un estado del sistema, que deben ser evaluados con valor *cierto* en todos los estados origen y destino, respectivamente, de la transición funcional, alcanzables en alguna ejecución del proceso. Es importante señalar que, a pesar de que una transición funcional sea aplicable a un estado para el cual el predicado de un pre–requisito se evalúe con valor *falso*, el proceso verifica el requisito si dicho estado no forma parte de ninguna posible ejecución del proceso. Si se denotan estos requisitos como $q_{pre}\,(f,p)$ y $q_{post}\,(f,p)$, donde f es la transición funcional y p el predicado:

$$P \models q_{pre}\,(f,p) \iff \forall e \in E(P),\ \forall i\ /\ e^i \in dom(f)\ \ y\ \ f(e^i) = e^{i+1},\ e^i \models p \tag{6.11}$$

$$P \models q_{post}\,(f,p) \iff \forall e \in E(P),\ \forall i\ /\ e^i \in dom(f)\ \ y\ \ f(e^i) = e^{i+1},\ e^{i+1} \models p \tag{6.12}$$

Un requisito de *ejecutabilidad* de una transición funcional se cumple si y sólo si existe al menos una ejecución posible del proceso en la cual se ejecute dicha transición funcional. Si se denota este requisito como $q_{exe}\,(f)$, donde f es la transición funcional:

$$P \models q_{exe}\,(f) \iff \exists e \in E(P),\ \exists i\ /\ e^i \in dom(f)\ \ y\ \ f(e^i) = e^{i+1} \tag{6.13}$$

También es posible especificar requisitos utilizando lógicas temporales. El soporte a este tipo de requisitos depende del tipo de lógicas temporales que permita verificar la herramienta utilizada. En general, se denota a este tipo de requisitos como $q_{lt}\,(p)$, donde p es la propiedad y el subíndice puede denotar el tipo de lógica concreta mediante la cual se expresa dicha propiedad.

$$P \models q_{lt}\,(p) \iff P \models p \tag{6.14}$$

6.2.4. Requisitos generales

Tal y como se ha expuesto anteriormente, los requisitos recogidos bajo el concepto de *solidez* de Aalst resultan especialmente interesantes para comprobar la corrección de cualquier proceso de negocio. Por tanto, a continuación se adapta cada uno de ellos al formalismo CFM.

Requisito RG–1: finalización eventual del proceso

Toda instancia del proceso debe, eventualmente, finalizar. Este es un requisito fundamental, dado que demuestra la ausencia de *abrazos mortales* o de ciclos de duración infinita en el proceso. Se puede descomponer este requisito en dos requisitos: ausencia de ciclos de duración infinita y finalización en un estado correcto.

Por una parte, el proceso no puede contener ciclos de duración infinita, dado que, cuando se da un ciclo de este tipo, el proceso no alcanza nunca un estado final. En algunas ocasiones, se puede asumir que un ciclo sí representa un progreso para el proceso. En estos casos se

puede relajar la condición para permitir este tipo de ciclos. Un ciclo se da como válido si alguna de las transiciones funcionales del mismo supone un progreso del proceso. El diseñador puede marcar una o más transiciones funcionales con el atributo *progress* para indicar que suponen un progreso. En el apartado 6.3.2 se propone cómo verificar la ausencia de bucles, y se propone un ejemplo.

Por otra parte, el proceso, siempre que alcance un estado final, debe estar en un estado final válido. Si se define un predicado fin que se evalúa con valor *cierto* en un estado si y sólo si dicho estado representa una finalización válida del proceso, el requisito se puede expresar en forma de *objetivo* asociado a dicho predicado. Por tanto:

$$q_{obj}\,(fin) \tag{6.15}$$

Se puede definir esta propiedad también de forma equivalente como:

$$q_{obj}\,(fin) \iff S_f \subseteq \{s \;/\; s \models fin\} \tag{6.16}$$

Dado que la forma de representación concreta de un proceso en el formalismo CFM es dependiente del lenguaje en el cual dicho proceso haya sido definido originalmente, la forma de representar el predicado de esta propiedad también depende de ello. Por ejemplo, si se representa el proceso mediante una actividad p, de tal forma que el inicio de dicha actividad modele el inicio de una instancia del proceso, y el fin de dicha actividad modele su finalización, entonces el predicado debe comprobar si dicha actividad ha alcanzado un estado final. Utilizando el ciclo de vida presentado en la figura 4.1, se enunciaría la propiedad fin como:

$$fin \equiv (p.estado = finalizado \vee p.estado = error) \tag{6.17}$$

Requisito RG–2: finalización correcta del proceso

Una vez finalizada una instancia de un proceso, no debe ser posible la ejecución de ninguna transición funcional asociada a dicho proceso.

Este requisito establece que todo estado de finalización del proceso debe ser un estado final desde el punto de vista del sistema formal básico. Su definición en términos del sistema formal básico es:

$$\{s \;/\; s \models fin\} \subseteq S_f \tag{6.18}$$

Dado que el requisito RG–1 establece la condición con el símbolo de exclusión contrario, se puede deducir que una condición necesaria para que un proceso cumpla conjuntamente los requisitos RG–1 y RG–2 es:

$$S_f = \{s \;/\; s \models fin\} \tag{6.19}$$

El requisito RG–2 puede ser verificado imponiendo el siguiente pre–requisito a todas las transiciones funcionales del proceso:

$$\forall f \in F,\; q_{pre}\,(f, \neg fin) \tag{6.20}$$

Si se define fin de la misma forma que en el ejemplo propuesto para el requisito RG–1, entonces el requisito RG–2 se expresaría como:

$$\forall f \in F,\; q_{pre}\,(f, (p.estado \neq finalizado \wedge p.estado \neq error)) \tag{6.21}$$

Requisito RG–3: finalización de todas las actividades del proceso

Una vez una instancia del proceso alcanza un estado final, todas sus actividades deben haber finalizado. Esto es, cuando el proceso finaliza su ejecución no puede haber ninguna actividad que no esté en un estado final correcto, a no ser que no haya sido ejecutada.

Se puede expresar este requisito mediante un *objetivo*. El predicado depende del ciclo de vida con que se modele la ejecución de actividades. Por ejemplo, con el ciclo de vida de la figura 4.1, se expresaría este requisito de la siguiente forma:

$$q_{obj}\left(\bigwedge_{a\in ACT}(a.estado = finalizado \vee a.estado = error \vee a.estado = parado)\right) \tag{6.22}$$

Requisito RG–4: ejecutabilidad de todas las transiciones funcionales del proceso

Toda transición funcional del proceso debe ser ejecutada en al menos una ejecución del mismo. Esto es, no puede haber ninguna transición funcional cuya ejecución sea imposible.

Se puede expresar este requisito directamente de la siguiente forma, donde F denota el conjunto de todas las transiciones funcionales de un proceso:

$$\forall f \in F,\ P \models q_{exe}(f) \tag{6.23}$$

La generación automática de definiciones de procesos CFM a partir de definiciones expresadas mediante otros lenguajes puede dar lugar a transiciones funcionales no ejecutables, sin que ello suponga un error en el proceso o el algoritmo de transformación. En estos casos, puede ser conveniente no exigir este requisito, o eximir a determinadas transiciones funcionales de su cumplimiento.

Requisito RG–5: ejecutabilidad de todas las actividades del proceso

Toda actividad del proceso debe ser ejecutada en al menos una ejecución del proceso. Esto es, no puede haber ninguna actividad cuya ejecución sea imposible.

Este requisito es menos restrictivo que el requisito RG–4. Se puede expresar en función de requisitos con respecto a la ejecutabilidad de transiciones funcionales, considerando que una actividad se ejecuta si se ejecuta al menos una de las transiciones funcionales que la definen. Si a es la actividad, el requisito RG–5 se expresaría en función del requisito de ejecutabilidad de sus transiciones funcionales como:

$$\forall a \in ACT,\ \exists f \in a\ /\ P \models q_{exe}(f) \tag{6.24}$$

6.3. Verificación de procesos mediante Spin

El apartado 2.1.7 del capítulo de estado del arte presenta la herramienta de *model checking* Spin. Esta herramienta permite verificar sistemas definidos mediante el lenguaje Promela. Se definen en este lenguaje tanto el propio sistema como su especificación. Holzmann, desarrollador principal de Spin y Promela, expone el lenguaje Promela, cómo se usa Spin y los algoritmos que implementa internamente en [72, 73].

```
/* Constants for enumerated values */
(...)

/* Typedefs for entity types */
(...)

/* Data objects for entities */
(...)

/* The process */
proctype process_name
{
    /* initializations */
    (...)

    /* main loop: functional transitions and assertions */
    (...)
}

/* Init process */
init
{
    run process_name()
}
```

Figura 6.1: Estructura básica de la definición de un proceso y su especificación en Promela.

Para verificar un proceso definido mediante el formalismo CFM es necesario, primero, transformar su definición y especificación a un modelo Promela. A continuación, se puede realizar la verificación sobre dicho modelo y devolver los resultados al dominio de CFM. En este apartado se expone una metodología de transformación de CFM a Promela, y se comenta cómo realizar las verificaciones con Spin.

6.3.1. Transformación de CFM a Promela

Un proceso CFM se representa en Promela mediante un proceso y su correspondiente tipo de proceso (palabra clave *proctype*). El nombre asignado al tipo de proceso es el propio nombre del proceso CFM. La figura 6.1 representa la estructura global de la definición Promela que se genera. En los siguientes apartados se expone cómo se transforma cada uno de los componentes de un proceso CFM a Promela.

Entidades y tipos de entidades

Los tipos de entidades pueden ser transformados directamente a estructuras de datos de tipo registro de Promela (palabra clave *typedef*). El nombre asignado a este tipo de datos en Promela es el asignado al propio tipo de entidad CFM, con el prefijo “`T_`”.

Cada uno de los campos del tipo de entidad CFM se transforma en un campo del elemento correspondiente de Promela. A cada campo se le asigna como valor por defecto el propio valor por defecto que le corresponda en el formalismo CFM. Los tipos primitivos se transforman de la siguiente forma:

- Tipo *boolean*: se transforma a tipo *Booleano* de Promela (palabra clave *bool*).

- Tipo *integer*: se transforma a tipo *short*, dado que este tipo de datos CFM se representa con 16 bits. Podría implementarse algún mecanismo que permitiese al diseñador seleccionar el tamaño del tipo de datos destino de acuerdo a sus necesidades.
- Tipo *enum*: se transforma a un tipo de datos *bit*, *byte* o *short* dependiendo del número de valores definidos para el tipo de datos (menor o igual que 2, 256 y 8191, respectivamente). No se emplea el tipo *mtype* de Promela dada su restricción en el número máximo de valores declarables (256). Los valores enumerados se transforman, para facilitar la legibilidad del código, a constantes, cuyo nombre se compone como la concatenación del nombre del tipo de entidad, campo y valor, separados por el carácter de *subrayado*. Para ello, se emplea la palabra clave *#define* del preprocesador de Promela.
- Tipo *abstract*: se ignora en la transformación a Promela, dado que no debe influir en la verificación.

Una *entidad* se declara en Promela como un objeto de datos cuyo nombre coincide con el de la entidad y cuyo tipo de datos es el correspondiente a la transformación de su tipo de entidad.

Para ilustrar lo expuesto en este apartado, se propone un ejemplo en la figura 6.2. En él se declaran dos tipos de entidades (*OilOrder* y *OilConfirmation*) y dos entidades, cada una correspondiente a uno de los tipos anteriores. A todos los campos se les introduce su valor por defecto.

Si a una entidad se le asocia un conjunto de valores iniciales, entonces se establecen estos valores iniciales al inicio del elemento *proctype*. Mediante el comando *if* de Promela, sin condición asociada, se selecciona de forma no determinista un valor inicial para la entidad de entre los declarados. La figura 6.3 muestra un ejemplo.

Expresiones

Las expresiones CFM tienen una estructura y conjunto de operadores muy similares a las expresiones Promela y son, por tanto, fácilmente transformables. La transformación se realiza en dos modos: modo de evaluación y modo de asignación. En el *modo de evaluación* se genera una expresión evaluable a un valor *Booleano*, entero o enumerado. En el *modo de asignación* se genera una expresión con una o más instrucciones de asignación de valores a atributos.

Si el elemento principal de la expresión es un operador, se transforman recursivamente sus operandos, sin cambiar el modo de transformación, y se genera la expresión Promela correspondiente. Todos los operadores del formalismo CFM tienen un operador equivalente en Promela.

Si el elemento principal de una expresión es una referencia a variable, esto es, una producción *VarRef* en la gramática de CFM, entonces se transforma a una referencia al objeto de datos correspondiente a su entidad, teniendo en cuenta también el campo referenciado. En caso de estar en *modo de asignación*, se asigna valor 1 en la expresión resultante, dado que se trata de una asignación a dato *Booleano* especificada de forma compacta. El caso de asignación a valor 0 se gestiona directamente durante la transformación del operador de negación en *modo de asignación*.

```
/* message {http://www.it.uc3m.es/jaf/ns/OliveOilService}OilOrder */
enttype OilOrder {
    oilType: enum (olive, soja, sunflower);
    quantity: integer;
    customerId: enum (_abstract__none);
}

/* message {http://www.it.uc3m.es/jaf/ns/OliveOilService}OilConfirmation */
enttype OilConfirmation {
    accepted: boolean;
}

entity oOrder: OilOrder;
entity oOrderConfirm: OilConfirmation;
```

```
/* enum constants */
#define OilOrder_oilType_olive 0
#define OilOrder_oilType_soja 1
#define OilOrder_oilType_sunflower 2
#define OilOrder_customerId__abstract__none 0

/* enttype definitions */
typedef T_OilOrder {
    byte oilType = OilOrder_oilType_olive;
    int quantity = 0;
    bit customerId = OilOrder_customerId__abstract__none;
};
typedef T_OilConfirmation {
    bool accepted = 0;
};

/* entity definitions */
T_OilOrder oOrder;
T_OilConfirmation oOrderConfirm;
```

Figura 6.2: Ejemplo de transformación de entidades y tipos de entidades del formalismo CFM a lenguaje Promela.

Si el elemento principal es una expresión de igualdad (o desigualdad), se transforman recursivamente sus dos operandos. El operando de la derecha se transforma en modo de *evaluación*, independientemente de que el modo actual sea *asignación*. El propio operador de igualdad se transforma a un símbolo "=" en modo *asignación* y "==" en modo *evaluación*. El operador de desigualdad se transforma a "!=" y sólo puede aparecer en modo *evaluación*. Por otra parte, hay que gestionar algunas situaciones especiales. Si uno de los operandos de la igualdad es una constante de tipo enumerado, el otro operando puede ser otra constante o una referencia a variable. Si es otra constante, se reemplaza la expresión completa por "1" o "0" dependiendo de si son iguales o distintos. Si se trata de una referencia a variable, se busca el término constante en la lista de valores del campo de la entidad correspondiente, y se transforma al nombre simbólico que se le haya asignado a dicho valor previamente[2].

Se muestran ejemplos de transformación de expresiones una vez expuesta la transforma-

[2]Esto es, el nombre asignado al principio del modelo Promela mediante una instrucción *#define*, tal y como se describe en el apartado anterior.

```
entity oOrder: OilOrder = {{olive, 1, abstract_none}
                          {sunflower, 1, abstract_none}};
```

```
proctype Example
{
    /* initialization of entity oOrder */
    if
        :: oOrder.oilType = OilOrder_oilType_olive;
           oOrder.quantity = 1;
           oOrder.customerId = OilOrder_customerId__abstract__none
        :: oOrder.oilType = OilOrder_oilType_sunflower;
           oOrder.quantity = 1;
           oOrder.customerId = OilOrder_customerId__abstract__none
    fi
(...)
}
```

Figura 6.3: Ejemplo de transformación de entidades CFM con valores iniciales alternativos a Promela. El tipo de entidad *OilOrder* se declara en el ejemplo de la figura 6.2.

ción de transiciones funcionales, en el apartado siguiente.

Actividades y transiciones funcionales

Las actividades en CFM representan sólo una agrupación lógica de transiciones funcionales, pero no tienen ningún significado desde el punto de vista del sistema formal básico. Por tanto, no se transforman de forma explícita a Promela, sino que únicamente se transforman sus transiciones funcionales.

Las transiciones funcionales se transforman en el interior del elemento *proctype* correspondiente al proceso, dentro de un bucle *do* que internamente contiene un bloque condicional *if*. Cada una de las transiciones se transforma a una cláusula de este elemento *if*. La figura 6.4 muestra esta estructura. La guarda de dicha cláusula, esto es, la condición, debe ser la transformación del dominio de la transición funcional, en modo de *evaluación*. A continuación de la guarda, se debe introducir la transformación de su expresión de acción, en modo de *asignación*.

De esta forma, el modelo Promela se ejecuta en un bucle, y en cada iteración selecciona una transición funcional cuyo dominio se evalúe a valor *cierto*. Una vez seleccionada, ejecuta su expresión de acción. Si en algún momento ninguno de los dominios se evalúa con valor *cierto*, se ha encontrado un estado final. En ese caso, se selecciona la cláusula *else*, que finaliza el bucle con una instrucción *break*.

En la figura 6.5 se muestra un ejemplo de transformación de una actividad con varias transiciones funcionales.

6.3.2. Verificación de los distintos tipos de requisitos

En este apartado se describe cómo se puede utilizar Spin para verificar los distintos tipos de requisitos que pueden formar parte de una especificación CFM.

```
proctype process_name
{
    /* intializations */
    (...)

    /* main loop */
    do
        /* functional transitions */
        :: if
            :: (domain1) -> (action1)
            :: (domain2) -> (action2)
            :: (...)
            :: else -> break
        fi;

        /* assertions for invariants */
        (...)
    od;

    /* assertions for goals */
    (...)
}
```

Figura 6.4: Estructura básica del elemento *proctype* y de la transformación de transiciones funcionales.

Invariantes

El predicado asociado a una invariante debe tomar valor *cierto* en todos los estados alcanzables por el proceso. Por tanto, cada vez que el modelo Promela desarrolle un nuevo estado del proceso, es necesario evaluar el predicado de cada una de las invariantes y comprobar que el resultado de dicha evaluación sea *cierto*. Para ello, se puede colocar una sentencia *assert* para cada una de las invariantes, dentro del bucle *do* principal y a continuación del bloque *if*. Esta sentencia *assert* será ejecutable inmediatamente a continuación de la ejecución de cada transición funcional.

En la figura 6.6 se muestra un ejemplo de codificación de dos invariantes. El predicado de las invariantes se transforma a Promela de la misma forma que el resto de expresiones, en modo de *evaluación*.

Si Spin informa de que alguna de estas cláusulas *assert* es violada, entonces el requisito correspondiente no se cumple. Por otra parte, Spin es capaz de generar un contraejemplo al requisito, esto es, la traza de una ejecución en la cual se viola dicha cláusula.

Dado que los requisitos de este tipo se transforman a cláusulas *assert* en lenguaje Promela, es necesario configurar la verificación de Spin para que realice un análisis de requisitos de *seguridad* (*safety*), con comprobación de cláusulas *assert*.

Objetivos

La representación de requisitos de tipo *objetivo* se puede realizar también mediante cláusulas *assert* en el modelo Promela. Dado que el predicado asociado al requisito debe ser cierto únicamente en los estados finales del proceso, se ubican estas cláusulas *assert* a continuación del cierre del bucle *do* principal.

```
activity response_neg_act_3 {
    transition begin {
        domain: {(prepare_neg_response_act_2_lc__.state=completed
                  & response_neg_act_3_lc__.state=not_started)}
        action: {response_neg_act_3_lc__.state=running}
    }
    transition complete {
        domain: {response_neg_act_3_lc__.state=running}
        action: {response_neg_act_3_lc__.state=completed}
    }
}
```

```
fi
    /* transition response_neg_act_3::begin */
    :: ((prepare_neg_response_act_2_lc__.state==BPEL_lc_basic_state_completed)
         && (response_neg_act_3_lc__.state==BPEL_lc_basic_state_not_started))
       -> response_neg_act_3_lc__.state=BPEL_lc_basic_state_running

    /* transition response_neg_act_3::complete */
    :: ((response_neg_act_3_lc__.state==BPEL_lc_basic_state_running))
       -> response_neg_act_3_lc__.state=BPEL_lc_basic_state_completed

    (...)
if
```

Figura 6.5: Ejemplo de transformación de transiciones funcionales a Promela.

En la figura 6.7 se muestra un ejemplo de la representación mediante Promela de un objetivo. Al igual que en el caso de las invariantes, el predicado se transforma en modo de *evaluación*. Spin es capaz de proponer contraejemplos para los requisitos de tipo *objetivo* que no se cumplan.

Es necesario configurar la verificación de Spin para que realice un análisis de requisitos de *seguridad* (*safety*), con comprobación de cláusulas *assert*.

Pre y post–requisitos de transiciones funcionales

El predicado *Booleano* de un *pre–requisito* de una transición funcional debe ser siempre evaluado con valor cierto inmediatamente antes de la ejecución de la misma. Se puede representar en Promela mediante una cláusula *assert*, colocada inmediatamente antes del predicado de acción de la transición funcional.

La representación de un *post–requisito* es similar, pero la cláusula *assert* se coloca inmediatamente a continuación del predicado de acción.

La figura 6.8 muestra un ejemplo de especificación de pre y post–requisitos, así como su representación en el proceso Promela resultante de la conversión.

Al igual que en el caso de los requisitos de tipo *invariante* y *objetivo*, es necesario configurar la verificación de Spin para que realice un análisis de requisitos de *seguridad* (*safety*), con comprobación de cláusulas *assert*.

```
invariant invariant_0 {
    (!process_oliveProcess_lc__.state=running
     | (response.waitingDays <= request.request__deadline))
}
invariant invariant_2 {
    (!process_oliveProcess_lc__.state=running
     | ((response.price <= request.request__maxPrice)
     | response.waitingDays=request.request__deadline))
}
```

```
/* main loop */
do
    /* functional transitions */
    :: if
       (...)
    fi;

    /* assertions for invariants*/
    /* invariant invariant_0 */
    assert((!(process_oliveProcess_lc__.state==BPEL_lc_state_running))
           || ((response.waitingDays <= request.request__deadline)));
    /* invariant invariant_2 */
    assert((!(process_oliveProcess_lc__.state==BPEL_lc_state_running))
           || (((response.price <= request.request__maxPrice))
           || (response.waitingDays==request.request__deadline)));
od;
```

Figura 6.6: Verificación de invariantes con Spin.

Ejecutabilidad de transiciones funcionales

Spin permite detectar fragmentos de código Promela que no son ejecutables en ningún caso. Para cada una de las transiciones funcionales de un proceso CFM que no sean ejecutables, Spin detectará que el código correspondiente a la transformación de su predicado de acción no es ejecutable.

Por tanto, para detectar transiciones funcionales no ejecutables, es suficiente con realizar la verificación, que detectará qué líneas del modelo Promela no son ejecutables. Tras finalizar esta, hay que identificar a qué transición funcional pertenece cada una de las líneas detectadas como no ejecutables.

Para verificar este tipo de requisitos, es necesario configurar la verificación para que realice un análisis de requisitos de *seguridad* (*safety*), y activar la opción de *análisis de ejecutabilidad*. Nótese que los resultados del análisis de ejecutabilidad no son correctos si se configura la verificación en modo de requisitos de *viveza* (*liveness*).

Requisitos expresados mediante lógicas temporales

El sistema de representación de requisitos con lógica temporal en Spin es la cláusula *never*. Esta cláusula consiste en un requisito que impone que el comportamiento, finito o infinito, descrito por la misma no puede ocurrir *nunca* en el sistema. Este comportamiento se describe mediante la sintaxis habitual de Promela.

En el sistema CFM se han seleccionado LTL y CTL como lenguajes de expresión de

```
goal goal_1 {
    (response_act_15_lc__.state=completed
     | response_neg_act_3_lc__.state=completed)
}
```

```
/* main loop */
do
    /* functional transitions */
    :: if
       (...)
    fi;

    /* assertions for invariants*/
    (...)
od;

/* assertions for goals */
assert((response_act_15_lc__.state==BPEL_lc_basic_state_completed)
       || (response_neg_act_3_lc__.state==BPEL_lc_basic_state_completed))
```

Figura 6.7: Verificación de requisitos de tipo *objetivo* con Spin.

lógicas temporales, dado el soporte que la mayoría de las herramientas de verificación dan a estas lógicas temporales. En Spin es posible verificar requisitos expresados mediante LTL, ya que proporciona un mecanismo de traducción de propiedades mediante LTL a cláusulas *never*. Sin embargo, esta herramienta no permite verificar, en general, requisitos expresados mediante CTL. En cualquier caso, es importante resaltar que la expresividad de LTL es menor que la de las cláusulas *never* de Spin.

Dada una propiedad LTL, Spin busca posibles caminos en el proceso tales que sea cierta. Si encuentra alguno, lo notifica como un error y produce la traza de un camino en el cual sea cierta. Por ejemplo, si se introduce la propiedad $\mathbf{G}p$, en realidad Spin intentará demostrar $\mathbf{EG}p$. Si encuentra un camino tal que $\mathbf{G}p$ es cierto, entonces se cumple la propiedad $\mathbf{EG}p$, y Spin muestra dicho camino para demostrarlo.

Sin embargo, el enfoque habitual de verificación es distinto: cuando un diseñador plantea un requisito, espera que dicho requisito sea cierto, y que el sistema notifique como un error el hecho de que no lo sea, produciendo en este caso un contraejemplo. Por tanto, el diseñador debe expresar cada requisito LTL en Spin como una propiedad tal que la violación del requisito sea equivalente al cumplimiento de la propiedad. Por ejemplo, el requisito LTL $\mathbf{AG}(p \implies (p\mathbf{U}q))$ debe ser codificado en Spin como $\neg\mathbf{G}(p \implies (p\mathbf{U}q))$. Spin intentará encontrar un camino que demuestre la veracidad de $\mathbf{E}(\neg\mathbf{G}(p \implies (p\mathbf{U}q)))$. Si lo encuentra, quiere decir que el requisito inicial no se cumple porque, teniendo en cuenta que $\mathbf{E}(f) \iff \neg\mathbf{A}(\neg f)$:

$$\mathbf{E}(\neg\mathbf{G}(p \implies (p\mathbf{U}q))) \iff \neg\mathbf{AG}(p \implies (p\mathbf{U}q)) \tag{6.25}$$

Continuando con el mismo ejemplo, el siguiente comando Spin genera una cláusula *never* cuya violación es equivalente a una violación del requisito $\mathbf{AG}(p \implies (p\mathbf{U}q))$. Nótese que, en sintaxis Spin, el operador “[]” denota el operador $\mathbf{G}$ de LTL.

```
$ spin -f '![](p -> (p U q))'
```

```
prereq test (switch1_act_3, begin) {
    request.color__msg_colors=red
}
postreq test (switch1_act_3, begin) {
    request.color__msg_colors=white
}
```

```
/* functional transitions */
:: if
   (...)
   /* transition switch1_act_3::begin */
   :: ((receive1_act_2_lc__.state==BPEL_lc_basic_state_completed) &&...)
      ->
      assert(request.color__msg_colors==ColorMessage_color__msg_colors_red);
      atomic{switch1_act_3_lc__.state=BPEL_lc_basic_state_running;...}
      assert(request.color__msg_colors==ColorMessage_color__msg_colors_white);
fi;
```

Figura 6.8: Verificación de requisitos de tipo *pre–requisito* y *post–requisito* con Spin.

```
never {    /* ![](p -> (p U q)) */
T0_init:
        if
        :: (! ((q)) && (p)) -> goto accept_S4
        :: (1) -> goto T0_init
        fi;
accept_S4:
        if
        :: (! ((q))) -> goto accept_S4
        :: (! ((p)) && ! ((q))) -> goto accept_all
        fi;
accept_all:
        skip
}
```

La violación de la cláusula *never* anterior se produce si el flujo dentro de misma llega al final (etiqueta *accept_all*), o si se produce un camino que pasa infinitas veces por la etiqueta *accept_S4*. En [73] se detalla la semántica de las cláusulas *never* (capítulo 4) así como el algoritmo de transformación de propiedades LTL a cláusulas *never* (capítulo 6).

En el ejemplo anterior, las expresiones p y q pueden ser sustituidas por cualquier predicado *Booleano* relativo a los atributos del proceso. Sin embargo, Spin no permite que muchos operadores, como los aritméticos y de comparación, aparezcan en fórmulas LTL. La solución a este problema consiste en emplear la cláusula *#define* para asociar un nombre simbólico al predicado *Booleano* complejo. Este nombre simbólico sí puede aparecer en las propiedades LTL.

En resumen, la verificación de un requisito LTL se realiza traduciendo la negación el mismo a una cláusula *never*, y realizando la verificación con la opción *verificación de cláusula never* activada.

En cada verificación, Spin sólo permite realizar la verificación de una cláusula *never*. Si existen varios requisitos LTL en la especificación CFM, cada uno de ellos debe ser verificado en una sesión de verificación distinta, o todos ellos deben ser combinados en un único requisito LTL.

```
activity while_act_3 {
    transition begin {...}
    transition continue[progress] {
            domain: {(!request.can_continue & while_act_3_lc__.state=running
                      & receive_act_4_lc__.state=completed)}
            action: {receive_act_4_lc__.state=not_started}
    }
    transition complete {...}
    transition cancel {...}
}
```

```
/* transition while_act_3::continue */
:: ((!(request.can_continue))
    && (while_act_3_lc__.state==BPEL_lc_basic_state_running)
    && (receive_act_4_lc__.state==BPEL_lc_basic_state_completed))
   -> progress_while_act_3_continue:
      receive_act_4_lc__.state=BPEL_lc_basic_state_not_started
```

Figura 6.9: Transformación de una transificón funcional marcada mediante el atributo *progress*.

Ausencia de ciclos de duración infinita

La verificación del requisito general RG–1 supone verificar, además de un requisito de tipo objetivo, un requisito de ausencia de bucles de duración infinita. En este apartado se detalla cómo.

Spin permite verificar la ausencia de este tipo de bucles mediante la opción de verificación *liveness/non–progress cycles*[3]. Si durante la verificación se detecta un bucle sin progreso, Spin notifica el error y muestra una traza del mismo.

Para permitir la presencia de bucles en los cuales se produzca *progreso*, aquellas transiciones funcionales marcadas en el modelo CFM como *progress* se transforman estableciendo una etiqueta *progress* en su predicado de acción, tal y como se muestra en la figura 6.9. Nótese que, para permitir la existencia de más de una transición funcional con atributo *progress*, la etiqueta generada en Promela toma como sufijo el nombre de actividad y transición funcional. De esta forma se evita una posible colisión de nombre con la etiqueta *progress* de otras transiciones funcionales.

La verificación de requisitos de *viveza* (*liveness*) en Spin es incompatible con la verificación de requisitos de *seguridad* (*safety*) y de alcanzabilidad de código. Por ello, es necesario realizar al menos dos verificaciones independientes de un proceso para poder verificar tanto requisitos de ausencia de ciclos como de invariantes, objetivos, etc.

Ejemplo

En el capítulo 8 se muestra, a modo de caso de estudio, la verificación de un proceso BPEL4WS. Se espera que este caso de estudio resulte útil para la comprensión de los conceptos planteados en este apartado. En el apéndice A.3 se muestra el código Promela

[3]Se puede seleccionar en el diálogo de XSpin o mediante la opción “-l” en la ejecución del programa de verificación generado por Spin.

```
/* Typedefs for entity types */
(...)

MODULE main
VAR
    /* Data objects for entities */
    (...)
INIT
    /* initializations */
    (...)
TRANS
    /* transitions */
    (...)

/* specification */
(...)
```

Figura 6.10: Estructura básica de la definición de un proceso y su especificación en SMV.

correspondiente al proceso BPEL4WS estudiado.

6.4. Verificación de procesos mediante NuSMV

En este apartado se describe la integración de otra herramienta de *model checking* en la arquitectura: NuSMV. Esta herramienta permite realizar *model checking* mediante distintas técnicas, principalmente *model checking simbólico* mediante OBDDs y *model checking acotado*. Se describe brevemente en el apartado 2.1.7.

Las pruebas realizadas durante la elaboración de esta tesis han mostrado que, con los tipos de procesos de negocio empleados, el rendimiento de Spin es considerablemente mejor que el de SMV. Por ello, el esfuerzo dedicado a la integración de esta herramienta ha sido menor que en el caso de Spin. En consecuencia, aunque se describe igualmente cómo integrarlo en la arquitectura, no se hace con el mismo nivel de detalle.

6.4.1. Transformación de CFM a SMV

Una definición de proceso CFM se representa en SMV directamente mediante enumeración de las variables que componen el estado del sistema y sus transiciones, expresadas de forma muy similar a las transiciones funcionales de CFM. Por ello, la traducción es, en general, bastante directa e intuitiva. En la figura 6.10 se muestra la estructura básica de una definición SMV de un proceso CFM.

Cada uno de los tipos de entidades del modelo CFM se transforma como un módulo SMV, tal y como se explica en el apartado siguiente. El resto del proceso, incluyendo las entidades, sus inicializaciones y las transiciones funcionales, dan lugar a otro módulo, llamado *main*.

Tipos de entidades y entidades

Para cada tipo de entidad se define un módulo SMV. El nombre asignado al módulo es el del tipo de entidad CFM, con el prefijo "*_et_*". Cada uno de los campos del tipo de

entidad se representa mediante una variable dentro de dicho módulo. Los tipos primitivos se transforman de la siguiente forma:

- Tipo *boolean*: tipo *boolean* de SMV.
- Tipo *integer*: entero enumerado mediante un rango comprendido entre un valor mínimo y un valor máximo. Por defecto, dado que CFM representa enteros de 16 bits, el rango es $-2^{15} \dots 2^{15} - 1$. Dado que el tamaño de los tipos enteros influye considerablemente en el tiempo de computación y cantidad de memoria consumidos por NuSMV, el diseñador puede configurar la transformación para asignar un rango específico, más pequeño, a cada campo entero.
- Tipo *enum*: se transforma como un dato de rango enumerado de SMV.
- Tipo *abstract*: se ignora en la transformación, dado que no influye en la verificación.

Cada una de las entidades se declara directamente, en el bloque *VAR* del módulo *main*, como instancias del módulo correspondiente a su tipo de entidad.

En la figura 6.11 se muestra un ejemplo de transformación de tipos de entidades y entidades CFM a lenguaje SMV.

Expresiones

El lenguaje de expresiones de CFM es bastante similar, en cuanto a operadores, al de SMV. Al igual que en el caso de Promela, es necesario realizar la transformación en dos modos: *modo de evaluación* y *modo de asignación*. Al igual que en CFM, el operador de comparación de igualdad y el de asignación de SMV son iguales. Es más, en la definición de transiciones se mezclan directamente expresiones de dominio y de acción. Sin embargo, una diferencia muy importante está en que, al igual que en formalismos como Z, en SMV se especifican las transiciones diferenciando explícitamente el valor actual de una variable y su valor siguiente. Por ejemplo, la expresión mostrada a continuación aplica cuando el estado actual verifica $a < 5$ y $b = 2$, y su resultado consiste en incrementar el valor de a en el siguiente estado:

```
a < 5 & b = 2 & next(a) = a + 1
```

Como se puede observar, el uso del operador *next()* permite especificar una transición mediante un único predicado *booleano*, sin necesidad de separar dominio y acción.

Se muestran ejemplos de la conversión de expresiones en siguientes apartados, una vez explicado el mecanismo de conversión de transiciones funcionales.

Actividades y transiciones funcionales

Al igual que en el caso de la transformación a Promela, la agrupación de transiciones funcionales en actividades no se tiene en cuenta en la transformación a SMV.

Tal y como se ha mencionado anteriormente, cada transición funcional puede ser representada con un único predicado, sin separación entre *acción* y *dominio*. Los predicados de todas las transiciones funcionales se agrupan, a su vez, en un predicado compuesto por una *o–lógica* de todas ellas. Este predicado se declara en el bloque *TRANS* del módulo *main*.

```
/* message {http://www.it.uc3m.es/jaf/ns/OliveOilService}OilOrder */
enttype OilOrder {
    oilType: enum (olive, soja, sunflower);
    quantity: integer;
    customerId: enum (_abstract__none);
}

/* message {http://www.it.uc3m.es/jaf/ns/OliveOilService}OilConfirmation */
enttype OilConfirmation {
    accepted: boolean;
}

/* entity definitions */
T_OilOrder oOrder;
T_OilConfirmation oOrderConfirm;
```

```
-- enttype OilOrder
-- message {http://www.it.uc3m.es/jaf/ns/OliveOilService}OilOrder
MODULE _et_OilOrder
VAR
    oilType : {_olive, _soja, _sunflower};
    quantity : 0..7;
    customerId : _replaced0_request__customerId;

-- enttype OilConfirmation
-- message {http://www.it.uc3m.es/jaf/ns/OliveOilService}OilConfirmation
MODULE _et_OilConfirmation
VAR
    accepted : boolean;

MODULE main
VAR
    oOrder : _et_OilOrder;
    oOrderConfirm : _et_OilConfirmation;
```

Figura 6.11: Ejemplo de transformación de entidades y tipos de entidades del formalismo CFM a lenguaje Promela.

Un aspecto importante en la transformación de expresiones de acción es que deben especificar el valor en el siguiente estado de *todos* los atributos del modelo CFM, no sólo aquellos que cambien de valor. Si un atributo a no cambia de valor se expresa como $next(a) = a$. Esto provoca que el modelo SMV tenga un tamaño muy grande.

En la figura 6.12 se muestra un ejemplo de transformación de varias transiciones funcionales al modelo SMV.

6.4.2. Verificación de distintos tipos de requisitos

No todos los tipos de requisitos representables en el modelo CFM pueden ser expresados de forma sencilla en el modelo SMV. A continuación se indica qué tipos pueden ser verificados de forma sencilla y cuáles no:

- Invariantes: una invariante $q_{inv}(p)$ puede ser expresada mediante la fórmula CTL **AG**p. La verificación de fórmulas CTL que siguen este patrón puede ser realizada

```
activity response_neg_act_3 {
    transition begin {
        domain: {(prepare_neg_response_act_2_lc__.state=completed
                 & response_neg_act_3_lc__.state=not_started)}
        action: {response_neg_act_3_lc__.state=running}
    }
    transition complete {
        domain: {response_neg_act_3_lc__.state=running}
        action: {response_neg_act_3_lc__.state=completed}
    }
}
```

```
(... /* other functional transitions */ ...)
|
-- activity response_neg_act_3
--
-- transition begin
    ( ((prepare_neg_response_act_2_lc__.state=_completed)
        & (response_neg_act_3_lc__.state=_not_started)) &
    (next(response_neg_act_3_lc__.state)=_running)
     & next(request.request__quantity)=request.request__quantity
     & next(request.request__maxPrice)=request.request__maxPrice
     & next(request.request__deadline)=request.request__deadline
     & (... /* any other attribute that does not change */ ...)
|
-- transition complete
    ( ((response_neg_act_3_lc__.state=_running)) &
    (next(response_neg_act_3_lc__.state)=_completed)
     & next(request.request__quantity)=request.request__quantity
     & next(request.request__maxPrice)=request.request__maxPrice
     & next(request.request__deadline)=request.request__deadline
     & (... /* any other attribute that does not change */ ...)
|
(... /* other functional transitions */ ...)
```

Figura 6.12: Ejemplo de transformación de transiciones funcionales a SMV.

de forma muy eficiente, sin necesidad de construir OBDDs para el modelo. Véase la documentación de NuSMV para más información [26].

- Objetivos: un requisito objetivo $q_{obj}\,(p)$ puede ser expresado también mediante una fórmula CTL de *estabilidad*, que exige que, en todos los posibles caminos, debe alcanzarse eventualmente un estado tal que, a partir de él, el predicado p sea cierto en todos los estados sucesivos. Esta fórmula es $\mathbf{AF}(\mathbf{AG}f)$.

- Pre y post–requisitos de transiciones funcionales: estos requisitos no son representables de forma sencilla en los modelos NuSMV, tal y como se generan a partir de CFM.

- Ejecutabilidad de transiciones funcionales: tampoco se puede verificar este tipo de requisitos de forma sencilla con NuSMV, tal y como se generan a partir de CFM.

- Requisitos expresados con lógicas temporales: NuSMV permite verificar requisitos CTL mediante *model checking simbólico*, y requisitos LTL mediante *model checking*

Requisito	Spin	NuSMV
Invariante	Sí	Sí
Objetivo	Sí	Sí
Pre/Post–requisito	Sí	No
Ejecutabilidad	Sí	No
Lógica temporal	LTL	CTL/LTL

Cuadro 6.1: Comparación de la capacidad de Spin y NuSMV para verificar de forma sencilla los distintos tipos de requisitos.

acotado. Su transformación a partir de requisitos CFM es inmediata, dado que la sintaxis de las expresiones temporales en CFM se ha definido de forma análoga a la empleada por NuSMV.

6.5. Conclusiones

En este capítulo se propone un conjunto de tipos de requisitos mediante los cuales se puede construir una especificación para un proceso definido mediante CFM. Por otra parte, se definen, basándose en resultados de otros autores, un conjunto de cinco requisitos generales, aplicables, en general, a cualquier proceso definido mediante CFM conforme a las líneas generales propuestas en el capítulo 4.

También se propone una posible metodología de transformación de definiciones de procesos CFM, así como sus especificaciones, a lenguaje Promela, que puede ser aplicada de forma automática. Se indica cómo cada uno de los tipos de requisitos puede ser verificado con Spin. Asimismo, se indica cómo verificar los requisitos generales mediante esta herramienta.

Adicionalmente, se propone también una metodología de transformación de los mismos al lenguaje de SMV, aunque descrita en menor detalle.

Ambas metodologías de transformación han sido implementadas en el prototipo de *verbus*. Las pruebas realizadas con distintos procesos muestran que el rendimiento de Spin es claramente superior al de nuSMV, tanto en modo de *model checking simbólico* como de *model checking acotado*, para los tipos de modelos y requisitos que se generan aplicando estas metodologías. Por otra parte, tal y como se recoge en el cuadro 6.1, con Spin es posible verificar más tipos de requisitos, aunque no permite verificar expresiones CTL.

La principal conclusión de este capítulo es que, tal y como se ha mostrado, resulta factible realizar verificaciones automáticas de modelos CFM utilizando herramientas de verificación de propósito general y compiladores automáticos. La implementación del prototipo corrobora esta afirmación.

Capítulo 7

Integración de BPEL4WS en la arquitectura

In terms of standardization, this area [Web services composition]
is still rather immature, although the above mentioned BPEL
seems to be emerging as the leading service composition language.

(Gustavo Alonso, Fabio Casati, Harumi Kuno, Vijay Machiraju,
"Web Services. Concepts, Architectures and Applications", pág. 141, 2004)

En este capítulo se expone una posible vía para integrar el lenguaje BPEL4WS, en su versión 1.1, en la arquitectura propuesta en esta tesis. El objetivo fundamental es mostrar cómo el formalismo CFM es capaz de representar procesos definidos mediante un lenguaje real de definición de procesos. Por otra parte, se pretende que sirva como referencia para facilitar la integración de otros lenguajes de definición en la arquitectura. Para ello, se define la semántica de BPEL4WS mediante el formalismo CFM, esto es, se expone cómo representar procesos BPEL4WS mediante CFM. También se muestra cómo es posible crear una herramienta automática de compilación de definiciones de procesos BPEL4WS a definiciones CFM.

En primer lugar, en el apartado 7.1, se presenta un resumen breve de las principales características del lenguaje BPEL4WS. A continuación, en el apartado 7.2, se propone una semántica para representar mediante CFM los conceptos de BPEL4WS. Se expone esta semántica de forma estructurada, explicando la representación de cada uno de los tipos de actividades y demás características del lenguaje. Para cada uno de ellos, se propone un ejemplo que pretende ilustrar su representación. Posteriormente, en el apartado 7.3, se propone una notación para la representación de requisitos de procesos BPEL4WS, empleando XPath como lenguaje de expresiones. Se muestra cómo transformar estos requisitos al lenguaje CFM. En el apartado 7.4 se analizan las limitaciones detectadas en la representación de procesos BPEL4WS mediante el sistema formal CFM. Finalmente, se presentan las principales

conclusiones acerca de la integración de BPEL4WS en la arquitectura propuesta en esta tesis. Al final del capítulo se recogen las figuras correspondientes a los distintos ejemplos propuestos en los apartados 7.2 y 7.3.

7.1. Introducción a BPEL4WS

En el apartado 2.3 se introducen los conceptos básicos de composición y coordinación de servicios Web. Se resumen, en el apartado 2.3.3, los aspectos principales del lenguaje BPEL4WS. En este apartado se describe en mayor profundidad este lenguaje. Se toma como referencia la especificación de BPEL4WS en su versión 1.1 [14].

7.1.1. Representación y manejo de datos

Al igual que en lenguajes de programación convencionales, los datos manejados por una instancia de un proceso BPEL4WS se almacenan en *variables*.

Cada variable se define mediante un nombre, un ámbito de uso y un tipo de datos. El nombre es un identificador que permite hacer referencia a la variable desde las actividades del proceso. El ámbito de uso de una variable representa el conjunto de actividades que pueden hacer referencia a ella. Está delimitado por la actividad *scope* en el cual se defina. Si se define en el nivel superior del proceso, entonces su ámbito lo forman todas las actividades del proceso.

El tipo de datos se define como un tipo de datos XML Schema [131, 132, 133], un elemento XML definido mediante XML Schema o un tipo de mensaje WSDL [28]. En cualquiera de los tres casos, las definiciones de los tipos se realizan en uno o más ficheros WSDL adjuntos. En el propio proceso BPEL4WS únicamente se hace referencia al nombre dado al tipo de datos en su definición.

El comportamiento de algunas actividades depende del contenido de variables. En ellas, se puede consultar el contenido de una variable (o un fragmento de esta) mediante funciones XPath. En las actividades que involucran envío de un mensaje hacia otra entidad, se referencia a la variable que almacene dicho mensaje. En las actividades que involucran recepción de mensaje, se referencia a la variable en la cual se debe almacenar dicho mensaje. Por otra parte, es posible hacer asignaciones de datos a variables con la actividad *assign*, incluyendo asignaciones parciales a fragmentos de variables. Estos fragmentos se identifican mediante una expresión XPath o un nombre de *parte* de un mensaje WSDL.

7.1.2. El proceso

Cada definición BPEL4WS da lugar a un proceso. El proceso se representa mediante el elemento *process*, que actúa como raíz del documento BPEL4WS. Contiene al resto de definiciones que dan lugar al proceso: enlaces a socios, socios, variables, conjuntos de correlación, manejadores de fallos, manejadores de compensación, manejadores de eventos y la actividad principal del proceso.

Al igual que en el caso de procesos de negocio convencionales, en un momento dado pueden existir múltiples instancias de un mismo proceso BPEL4WS. Un proceso debe comenzar siempre con una actividad de recepción de un mensaje (invocación a una operación

WSDL proporcionada por el propio proceso). Se crea una nueva instancia del proceso para cada una de las invocaciones recibidas para esta operación. Las instancias de un proceso son independientes entre sí: su control de flujo está desconectado y cada una de ellas cuenta con su propia copia de las variables.

7.1.3. Actividades

Las actividades representan unidades de procesado. Al contrario que en lenguajes convencionales de modelado de procesos de negocio, no se modelan de forma explícita actividades de ejecución no automática, esto es, con intervención humana. Se clasifican en dos grandes grupos: estructuradas y básicas. Las estructuradas contienen internamente a otras actividades, y simplemente especifican restricciones que aplican a dichas actividades. Las básicas, por el contrario, representan unidades de trabajo atómicas, que no contienen a otras actividades. Por tanto, un proceso BPEL4WS no es más que un árbol de actividades. Su raíz es la actividad principal del proceso. Los nodos interiores del árbol son actividades estructuradas, mientras que las hojas del árbol son siempre actividades básicas.

En el cuadro 7.1 se muestran las actividades definidas en la especificación de BPEL4WS 1.1. En los siguientes apartados se exponen brevemente las características más relevantes de cada una de ellas.

Actividades de control de flujo

Las actividades estructuradas *sequence*, *flow*, *switch* y *while* controlan el orden de ejecución de sus actividades internas. Representan, respectivamente, ejecución secuencial, concurrente, condicional e iterativa.

La actividad *sequence* establece que las actividades internas deben ser ejecutadas en un orden concreto. No puede comenzar la ejecución de una actividad hasta que no haya finalizado la actividad que la precede.

En la actividad *flow* no se establece, en principio, ninguna limitación al orden y paralelismo de ejecución de las actividades. Sin embargo, se pueden establecer restricciones parciales entre pares de actividades mediante *vínculos* (elemento *link*). Un vínculo une una actividad *origen* y una actividad *destino*, ambas dentro (en cualquier nivel de profundidad) de una misma actividad *flow*. Su semántica especifica que no se puede ejecutar la actividad destino hasta que no se hayan evaluado todos sus vínculos entrantes. Un vínculo se evalúa cuando finaliza la actividad origen. La evaluación de un vínculo consiste en evaluar la expresión *Booleana* especificada en su actividad origen. Por otra parte, si una actividad es destino de más de un vínculo, puede especificar una expresión *Booleana* que, en función de la evaluación de cada uno de estos vínculos entrantes, establece si puede o no ser ejecutada.

La actividad *switch* se define mediante un conjunto de casos, donde cada caso tiene asociada una condición y una actividad. Se selecciona para ser ejecutado el primer caso cuya condición se evalúe con valor *cierto*. Si la condición de todos los casos se evalúa como *falso*, y se ha definido un caso *otherwise*, entonces se selecciona este último. Una vez seleccionado un caso, se ejecuta la actividad definida para dicho caso.

La actividad *while* tiene una condición asociada, y contiene internamente una actividad. La actividad interna se ejecuta iterativamente. En cada iteración se crea una nueva instancia de la misma, y de todas las actividades contenidas en esta en cualquier nivel de profundidad.

invoke	Invocación de una operación de un servicio Web proporcionado por una entidad externa al proceso. Si se trata de una operación asíncrona, finaliza una vez enviado el mensaje de petición. Si es síncrona, finaliza una vez se haya recibido el mensaje de respuesta.
receive	Recepción de un mensaje externo de petición para una operación de un servicio Web proporcionado por el propio proceso. No finaliza hasta que no se reciba una petición. Opcionalmente, puede crear una nueva instancia (caso) del proceso.
reply	Envío de un mensaje de respuesta correspondiente a un mensaje de petición recibido previamente para una operación de un servicio Web proporcionado por el propio proceso.
assign	Asigna un valor a una o más variables.
throw	Señaliza la ocurrencia de un fallo, a ser tratado mediante el mecanismo de gestión de fallos.
wait	Finaliza tras un intervalo de tiempo dado o a una fecha y hora dada.
empty	No hace nada.
compensate	Solicita la ejecución del manejador de compensación de un determinado ámbito.
sequence	Ejecución en orden de sus subactividades, de tal forma que nunca se ejecuta una subactividad de la secuencia hasta que la anterior haya finalizado.
switch	Selección condicional de una subactividad entre varias. Cada subactividad se asocia a una condición, y se ejecuta sólo la primera subactividad cuya condición sea evaluada como cierta.
while	Ejecución iterativa de una subactividad. Se asocia una condición a la actividad *while*.Inicialmente, o tras finalizar una iteración, se evalúa esta condición. Si es cierta, se ejecuta una nueva iteración. Si es falsa, se finaliza la actividad.
flow	Ejecución concurrente de sus subactividades. Permite añadir restricciones a la concurrencia mediante vínculos, de tal forma que la subactividad destino de un vínculo pueda ser ejecutada únicamente tras finalizar la subactividad origen del mismo.
pick	Selección no determinista basada en eventos externos. Se definen subactividades para ser ejecutadas si ocurren determinados eventos (mensajes de entrada o vencimiento de temporizaciones). Se ejecuta la subactividad correspondiente al primer evento que ocurra.
scope	Define un *ámbito* de ejecución para su subactividad.

Cuadro 7.1: Actividades de BPEL4WS 1.1. Las actividades del primer grupo son básicas, mientras que las del segundo son estructuradas.

Antes de iniciar cada iteración, se evalúa la condición. Si el valor es *falso*, se finaliza la actividad *while*. Si no, se inicia la nueva iteración.

Actividades de invocación y provisión de servicios Web

Un proceso BPEL4WS puede invocar operaciones proporcionadas por servicios Web en otras entidades. Para ello, se define la actividad *invoke*. En esta actividad se especifica la operación y puerto. También se especifica de qué variable se toma el mensaje WSDL de entrada de la operación. Si la operación es síncrona (esto es, se espera una respuesta), la actividad se bloquea hasta recibirla, y el mensaje de respuesta se almacena en la variable que se indique a tal efecto.

Una característica especial de esta actividad es que puede contener manejadores de fallos, para responder adecuadamente a fallos que se produzcan en la invocación, ya sean

generados por problemas en las comunicaciones o por un fallo, en terminología WSDL[1], generado por el servidor durante la ejecución de la operación.

Por otra parte, un proceso BPEL4WS también es proveedor de servicios Web. Por tanto, puede recibir invocaciones realizadas por entidades externas. Se pueden recibir invocaciones mediante las actividades *receive* y *pick*, así como en manejadores de eventos asociados al proceso o a una actividad *scope*.

Las actividad *receive* se bloquea hasta recibir una invocación para la operación y puerto especificados. Una vez recibida la invocación, almacena el mensaje de entrada en la variable que se indique.

La actividad *pick* especifica uno o más eventos, con una actividad asociada a cada uno de ellos. Se bloquea hasta que ocurra el primero de los eventos especificados. Cuando ocurre, se ejecuta la actividad asociada a dicho evento. La actividad *pick* finaliza una vez finalizada dicha actividad. Los eventos pueden ser invocaciones a una operación (especificada por puerto y operación) o temporizadores.

Si la invocación recibida se corresponde con una operación síncrona, es necesario que el proceso envíe un mensaje de salida (o un fallo) a la entidad remota. Para ello, se utiliza la actividad *reply*, en que se especifica, de nuevo, el puerto, operación y nombre de la variable que contiene el mensaje de salida.

Hay que tener en cuenta que en un momento dado pueden estar en ejecución múltiples instancias de un mismo proceso BPEL4WS. La asociación entre mensajes WSDL e instancias del proceso se realiza mediante el concepto de *correlación*.

Otras actividades básicas

La actividad *assign* permite asignar valores a variables (o a fragmentos determinados de variables). El origen de la asignación puede ser otra variable (o fragmento) o una expresión XPath.

La actividad *wait* bloquea la ejecución hasta un instante temporal especificado. Este instante se especifica mediante una fecha y hora absolutas o como una duración relativa al inicio de la actividad.

La actividad *empty* representa una operación nula. Finaliza inmediatamente.

La actividad *compensate* forma parte del mecanismo de manejo de compensación. La actividad *throw*, del mecanismo de gestión de fallos. Se explican ambas actividades en el apartado en que se exponen los mecanismos de manejo de fallos y compensación.

Delimitación de *ámbitos*

Un *ámbito* provee un contexto de ejecución para las actividades contenidas en su interior. Internamente contiene una actividad de nivel superior. Esta actividad puede ser compleja y, por tanto, es la raíz de un árbol de actividades. El contexto provisto por el ámbito a todas sus actividades internas está compuesto por:

[1] Una operación WSDL síncrona puede finalizar devolviendo un mensaje de salida o, en caso de que suceda un error o situación excepcional, devolver en vez de este mensaje un *fallo*. El concepto de fallo es similar al concepto de excepción de otros lenguajes de programación.

- Variables: su tiempo de vida es el tiempo de vida del ámbito, y sólo son accesibles a sus actividades internas.
- Manejadores de fallos: actividades a ser ejecutadas en caso de que se genere un *fallo* en alguna actividad interna del ámbito, y no sea capturado en ningún ámbito inferior.
- Manejador de compensación: define la actividad (posiblemente compleja) necesaria para compensar (o deshacer) las acciones llevadas a cabo por las actividades internas del ámbito. Se ejecuta si, tras haber finalizado correctamente la ejecución del ámbito, es necesario compensar los efectos de su ejecución debido a la ocurrencia de fallos en otras actividades del proceso externas al ámbito.
- Manejadores de eventos: actividades a ser ejecutadas en caso de ocurrencia de eventos, ya sean recepción de mensajes o vencimiento de temporizaciones.

Un ámbito se define mediante una actividad *scope*. Por ser una actividad, puede estar contenida en cualquier actividad estructurada. Por tanto, es posible que en el interior de un ámbito pueda haber anidados otros ámbitos, formando un árbol de ámbitos.

Por otra parte, el proceso (actividad *process*) tiene también implícitamente estructura de ámbito. Los mismo ocurre con la actividad *invoke*, que se ejecuta implícitamente en un ámbito virtual. Sin embargo, en este último caso se pueden definir manejadores de fallos y compensación, pero no variables ni manejadores de eventos.

7.1.4. Manejo de fallos y compensación

El lenguaje BPEL4WS incorpora mecanismos de manejo de fallos y compensación, que facilitan la definición del comportamiento del proceso en situaciones en que se produzcan errores o situaciones excepcionales. El objetivo es aumentar la robustez de los procesos. Por otra parte, se reduce la carga de trabajo manual necesaria para manejar instancias de procesos en que ocurran este tipo de situaciones.

El mecanismo de manejo de fallos es similar al mecanismo de excepciones del lenguaje Java. El mecanismo de compensación permite definir código para deshacer, en la medida de los posible, las acciones realizadas por una actividad o conjunto de actividades que han finalizado correctamente su ejecución. Este mecanismo permite integrar en BPEL4WS un modelo de *transacciones de larga duración*.

Manejo de fallos

Un fallo representa un error o situación anómala en el proceso. Se representa mediante un nombre y, opcionalmente, una variable. Esta variable permite adjuntar información adicional acerca del fallo. El nombre del fallo puede ser cualquiera de los definidos en la especificación de BPEL4WS, el de un fallo definido en la interfaz WSDL de los servicios, o cualquier otro que el diseñador haya establecido para su dominio de aplicación.

El manejo de fallos sigue el paradigma *try–catch–throw*. Los fallos definidos por la especificación de BPEL4WS representan errores propios del lenguaje, y pueden surgir en las distintas situaciones que define dicha especificación. Los fallos WSDL pueden surgir durante la invocación de una operación remota. El resto de fallos, definidos por el diseñador, se generan explícitamente mediante la actividad *throw*.

Un manejador de fallos define una actividad, posiblemente estructurada, a ser ejecutada en caso de ocurrencia de un fallo. Un manejador puede especificar un nombre de fallo para el cual resulta adecuado, e incluso una variable que representa la variable del fallo. Se define mediante los elementos *catch* y *catchAll*, dentro del elemento *faultHandlers*.

Una vez se genera un fallo en una actividad, se cancela la ejecución de la misma y el fallo se reenvía a su ámbito inmediatamente superior. En ese momento, se cancelan todas las actividades en ejecución dentro de dicho ámbito. Si el ámbito define al menos un manejador de fallos compatible con el fallo, se selecciona el más adecuado (se define en la especificación de BPEL4WS cómo se realiza esta selección). Una vez finalizada la ejecución del manejador, se finaliza la ejecución del ámbito. A todos los efectos, se considera que el ámbito finaliza con éxito (esto es, la ocurrencia del fallo no es visible desde el resto de actividades del proceso, porque se considera que el manejador de fallos ha realizado las acciones oportunas para reaccionar ante el error). Si, por el contrario, el ámbito no define ningún manejador de fallos compatible, se cancela el ámbito y el fallo se reenvía a su ámbito inmediatamente superior.

El reenvío a ámbitos superiores se realiza hasta que algún ámbito defina un manejador adecuado o se alcance el nivel del proceso. Si se alcanza este nivel, se cancelan todas las actividades del proceso. Si se define algún manejador compatible con el fallo, se ejecuta el más adecuado, y se finaliza el proceso una vez finalizado este manejador. En caso contrario, se cancela directamente la ejecución del proceso.

La ocurrencia de un fallo durante la ejecución de un manejador de fallos provoca directamente la cancelación del ámbito, y su reenvío al ámbito inmediatamente superior.

Compensación

El lenguaje BPEL4WS propone un modelo de transacciones de larga duración. Esto se debe a que la larga duración que puede tener un proceso hace poco conveniente, en general, la utilización del modelo de transacciones ACID[2]. El motivo principal es que sería necesario reservar recursos, en ocasiones, para la duración completa de una transacción, que puede ser incluso de varios días o semanas. En [13] se explica en mayor profundidad la aplicación de modelos de transacciones a procesos de negocio.

El modelo de transacciones de larga duración de BPEL4WS está inspirado en *sagas* [58], y se basa en *compensación*. Cada ámbito puede proporcionar un *manejador de compensación*, que define una actividad, posiblemente estructurada, para compensar (o deshacer, en la medida de lo posible) los efectos de la ejecución de las actividades del ámbito. Se puede solicitar la compensación de un ámbito desde un manejador de fallos o de compensación del ámbito inmediatamente superior, mediante la actividad *compensate*. Sólo se puede ejecutar el manejador de compensación de un ámbito si la ejecución de dicho ámbito ha finalizado correctamente.

Supongamos el ejemplo clásico en que un proceso debe realizar una reserva coordinada de vuelo y alojamiento en hotel, mediante combinación de invocaciones a un proveedor de vuelos y un proveedor de alojamientos. Si no se consigue realizar la reserva conjunta del vuelo y del hotel, entonces no se debe realizar ninguna reserva. Esto es, no es correcto

[2]Modelo de transacciones clásico y ampliamente utilizado en otros dominios, como gestión de bases de datos, basado en garantizar las propiedades de atomicidad, coherencia, aislamiento y permanencia de las operaciones.

realizar sólo una de las reservas. Se puede definir un ámbito B para las actividades de reserva de vuelo, un ámbito C para las actividades de reserva de hotel, y un ámbito A que envuelva a ambos. Supongamos que no se consigue realizar la reserva de vuelo en B, y que entonces se genera un fallo que se captura en un manejador de fallos de A. El manejador de fallos de A puede cancelar en ese momento la ejecución del ámbito C, si continúa en ejecución. Sin embargo, si el ámbito C había terminado previamente su ejecución con éxito (esto es, con la reserva realizada), es necesario cancelar dicha reserva. Para eso, se programaría en el manejador de compensación de C el comportamiento necesario para cancelar la reserva, y se forzaría la compensación de C desde el manejador de fallos del ámbito A.

7.2. Semántica de BPEL4WS basada en CFM

El objetivo de este apartado es aplicar el formalismo CFM a la verificación de procesos BPEL4WS. Para ello, se propone una semántica para BPEL4WS descrita en términos de dicho formalismo. En base a esta semántica, se puede automatizar la transformación de definiciones de procesos BPEL4WS a este formalismo, contribuyendo a la automatización del procedimiento global de verificación.

La semántica propuesta en este apartado es una de las contribuciones principales de esta tesis. Es también una de las tareas de mayor complejidad. Para intentar facilitar su comprensión, sin pérdida de rigurosidad, se ha estructurado la exposición con la intención de que el orden resulte lo más natural posible.

La exposición comienza proponiendo cómo representar las variables BPEL4WS como entidades de CFM. Para ello, es necesario poder representar también sus tipos de datos. Dado que la esencia de un proceso BPEL4WS es su capacidad para comunicarse con otras entidades externas, se discute a continuación cómo representar el comportamiento de dichas entidades.

Posteriormente, se presentan brevemente los principios generales de representación de actividades BPEL4WS mediante actividades CFM. Un aspecto esencial de dicha representación es el concepto de ciclo de vida, que modela los distintos estados internos que atraviesa una actividad durante la ejecución del proceso. El comportamiento de la actividad, esto es, su forma de evolucionar por dicho ciclo de vida e interactuar con el resto de actividades, se modela mediante transiciones funcionales. Dado que las transiciones funcionales dependen del tipo de actividad BPEL4WS, a continuación se exponen las características particulares de cada una de ellas. Para uniformar dicha exposición y facilitar su comprensión, se propone una notación que se utilizará en el resto del apartado.

Antes de comenzar con la exposición de las actividades, se expone cómo transformar expresiones XPath, lenguaje de expresiones utilizado por defecto en BPEL4WS, al lenguaje de expresiones de CFM. Es fundamental para comprender la semántica dada a algunas de las actividades.

El proceso BPEL4WS en sí mismo se modela también como una actividad en el modelo CFM. Se expone a continuación la forma de representarlo. Posteriormente, se comienza exponiendo la semántica dada a la actividad *assign*, así como a las actividades estructuradas *sequence*, *flow*, *switch* y *while*. A continuación, se presenta la semántica de las actividades utilizadas para proveer servicios Web a entidades externas: *receive*, *pick* y *reply*. Se finaliza esta parte del apartado exponiendo la semántica de las actividades *wait* y *empty*.

Las actividades *scope* e *invoke* tienen características especiales, junto con el propio proceso, que las diferencia del resto de las actividades de BPEL4WS: la posibilidad de contener manejadores de fallos, compensación y eventos. Se exponen, en primer lugar, las líneas básicas de representación de ambas actividades, excluyendo la funcionalidad asociada a los mecanismos de cancelación, manejo de fallos, compensación y manejo de eventos. Estos mecanismos se exponen, en este orden, a continuación, junto con las actividades *terminate*, *throw* y *compensate*, íntimamente ligadas a ellos.

A lo largo de la exposición se presentan algunas limitaciones en la capacidad de CFM para representar algunos aspectos concretos de la semántica de BPEL4WS. En el apartado siguiente se recopilan y discuten estas limitaciones.

Con el fin de facilitar la comprensión de la metodología de transformación, se ilustra la exposición mediante numerosos ejemplos. Dado su tamaño, estos ejemplos se presentan al final del capítulo. Durante la exposición, se citan los ejemplos pertinentes en cada caso.

7.2.1. Variables

Este apartado se basa en las líneas generales de representación de procesos de negocio, y más concretamente en los apartados 4.4.2 (representación de objetos manipulados) y 4.4.3 (representación de variables).

Un variable BPEL4WS se representa como una *entidad* en el modelo CFM. El *tipo de entidad* correspondiente se obtiene de transformar el tipo de datos de la variable BPEL4WS. Independientemente de que una variable se defina en el proceso principal o en un *ámbito* interno, su entidad correspondiente se define a nivel global. Véase en el apartado 4.5.2 la discusión acerca de la representación de variables de ámbito local. En cualquier caso, esto no supone ninguna limitación desde el punto de vista de la verificación de procesos BPEL4WS.

El tipo de una variable de BPEL4WS puede ser un tipo de datos XML Schema, un elemento definido mediante XML Schema o un mensaje WSDL. Se puede representar, con limitaciones, mediante un *tipo de entidad* en el formalismo CFM.

En general, el tipo de entidad contendrá tantos campos como elementos de información haya en el tipo de datos original. Los tipos de datos de BPEL4WS pueden ser simples o complejos. Los tipos simples dan lugar a un tipo de entidad con un único campo. Los tipos complejos, a un tipo de entidad con tantos campos como elementos finales de información pueda haber en el tipo de datos. En este último caso, el nombre de cada campo se escogerá de tal forma que refleje su posición jerárquica en el tipo de datos original. De esta forma se evita la utilización de nombres duplicados y se facilita la comprensión de la definición en CFM.

Representación de tipos de datos simples

Muchos datos simples de XML Schema admiten un rango infinito de valores. Como ya se ha expuesto en el apartado 4.5.1, los atributos en el formalismo CFM tienen siempre un rango finito de valores. Esto afecta a la representación de datos numéricos y, sobre todo, a tipos de datos basados en cadenas de texto. En este último caso, hay dos opciones: realizar una abstracción total del atributo o abstraer parcialmente sus valores en un tipo de rango enumerado.

Los tipos de datos simples de XML Schema se representan mediante los siguientes tipos de datos simples del formalismo CFM:

- Tipo *integer*: mediante el tipo *integer* de CFM. Opcionalmente, se puede optar por representarlo como *abstract* para reducir el espacio de estados del modelo.

- Tipo *boolean*: mediante el tipo *boolean* de CFM.

- Resto de tipos: se representan directamente mediante su codificación XML como cadenas de caracteres. Pueden ser representados como *abstract* (en general), o como tipos de rango enumerado. En este último caso, se analizan todos los valores que se asignan o con los cuales se comparan las variables de este tipo. El rango enumerado se construye como el conjunto de estos valores, además de un valor especial, que representa a cualquier valor distinto de los representados explícitamente.

Representación de tipos de datos complejos

El procedimiento general de representación de datos complejos es el siguiente, en función del tipo de datos de que se trate:

1. Mensaje WSDL: se representa cada una de sus partes de acuerdo al tipo de datos XML Schema que la defina. El nombre de la parte actúa como prefijo del nombre de cada campo generado a partir de datos de dicha parte.

2. Elemento XML: se representa de acuerdo al tipo de datos XML Schema que define al elemento.

3. Tipo de datos XML Schema: se representa recursivamente, comenzando por la definición raíz del tipo de datos. Se distinguen dos casos: tipos de datos simples y tipos de datos complejos.

    - Tipos de datos simples: se crea un campo para representar el tipo de datos.
    - Tipos de datos complejos: se representa recursivamente cada componente del tipo de datos complejo, ya sea *all*, *sequence* o *choice*. En caso de que este sea *choice*, es necesario añadir un campo adicional (de tipo enumerado) que indique cuál de las alternativas se representa.

Un tipo de datos XML Schema puede definir internamente un número infinito de tipos de datos simples. Por tanto, su transformación a CFM daría lugar a un número infinito de campos. Esto ocurre, por ejemplo, cuando un elemento puede contener a instancias de sí mismo. El formalismo CFM no es capaz de representar, por tanto, un tipo de datos general. Se analiza en profundidad esta limitación en el apartado 7.4.1.

Ejemplo

La figura 7.5 muestra un ejemplo de definición de variables y sus tipos mediante WSDL y BPEL4WS, y el resultado de su transformación al modelo de CFM, aplicando la metodología expuesta en este apartado.

7.2.2. Modelado de entidades externas

El lenguaje BPEL4WS está esencialmente orientado a la comunicación con otras entidades a través de servicios Web. Esta comunicación incluye la invocación de operaciones remotas provistas por entidades externas así como la provisión de operaciones locales que son invocadas por entidades externas. En ambos casos, el proceso recibe mensajes generados en las entidades externas, cuyo contenido puede afectar a su propia ejecución. Por tanto, es imprescindible, desde el punto de vista de la verificación de requisitos funcionales, tener en cuenta los distintos contenidos posibles de los mensajes recibidos desde entidades externas.

Normalmente no es posible disponer de un modelo completo del comportamiento de todas las entidades externas. Sin embargo, desde el punto de vista de la verificación, es suficiente disponer de una abstracción de dichas entidades, que modele únicamente el comportamiento de las entidades externas visible desde el punto de vista del proceso BPEL4WS. Para ello, se plantean varias alternativas:

- Modelar cada entidad externa como un proceso que se ejecuta concurrentemente con el resto de los procesos.

- En cada punto del proceso en que se reciban mensajes de una entidad externa, indicar un conjunto de mensajes alternativos que podrían ser recibidos. Para ello, también se plantean varias alternativas:

  - Definir cada uno de los mensajes como constantes.
  - Definir los mensajes mediante una actividad BPEL4WS (posiblemente estructurada) que modela la construcción del mensaje, basándose en el contenido de las variables del propio proceso.

Aunque lo ideal sería permitir todas las alternativas, la primera es considerablemente más compleja porque implica diseñar un mecanismo de modelado de las comunicaciones. En esta tesis se plantea como un posible trabajo futuro (apartado 9.3.2).

Se recurre, por tanto, a la segunda opción. Se permite proporcionar un conjunto de mensajes alternativos en cada actividad que deba recibir un mensaje. Cada uno de estos mensajes puede ser especificado tanto como una constante como mediante una actividad BPEL4WS. En este último caso, las actividades deben asignar el valor del mensaje a la variable de recepción del mensaje correspondiente, especificada en las actividades *receive*, *invoke* y *pick*, y en manejadores de eventos. Todos los mensajes alternativos son tenidos en cuenta para realizar la verificación.

Estos mensajes alternativos se definen en el propio documento BPEL4WS, mediante marcas añadidas específicamente para este entorno de verificación. En la figura 7.1 se muestra un ejemplo. En la descripción de las actividades *receive*, *invoke*, *pick* y del mecanismo de manejo de eventos se profundiza en esto.

En algunas ocasiones resulta conveniente abstraer el contenido del mensaje desde el punto de vista de la verificación, esto es, considerar relevante su llegada pero no su contenido. Este es el modo de representación utilizado cuando no se especifiquen mensajes alternativos.

```
<receive createInstance="yes" partnerLink="OliveOilPlnk"
         portType="tns:OliveOilService"
         operation="order" variable="request">
  <verbus:altMessages>
    <verbus:message>
      <request>
        <quantity>1</quantity>
        <maxPrice>3</maxPrice>
        <deadline>5</deadline>
        <customerId>c102000</customerId>
      </request>
    </verbus:message>
    <verbus:message activity="yes">
      <sequence>
        <assign>
          <copy>
            <from (...) />
            <to variable="request" />
          </copy>
        </assign>
        (...)
      </sequence>
    </verbus:message>
  </verbus:altMessages>
</receive>
```

Figura 7.1: Modelado de mensajes alternativos en una actividad *receive*.

7.2.3. Actividades

En este apartado se presentan los principios generales de representación de actividades BPEL4WS mediante el formalismo CFM. En siguientes apartados se expone cómo se transforma cada actividad concreta de BPEL4WS. Véanse las líneas generales de representación de actividades, en el apartado 4.4.4.

Cada actividad de BPEL4WS se transforma, en general, a una *actividad* en el formalismo CFM. El estado interno de la actividad dentro de su ciclo de vida se representa mediante una *entidad*. La evolución de la actividad por dicho ciclo de vida, y su efecto sobre las variables del proceso, se modela mediante las *transiciones funcionales* de la actividad CFM. Cada tipo de actividad BPEL4WS se modela con distintas transiciones funcionales.

Ciclo de vida

En este trabajo se han identificado dos tipos de ciclo de vida en actividades BPEL4WS. El *ciclo de vida general* permite representar el ciclo de vida de aquellas actividades que pueden contener manejadores de eventos, fallos o compensación. Esto incluye al propio elemento *process* y a las actividades *scope* e *invoke*. El *ciclo de vida simplificado* es una simplificación del anterior para el resto de actividades. Se representan ambos ciclos de vida en la figura 7.2. En ella se etiquetan los estados con mayúsculas y las transiciones con minúsculas. Los puntos negros indican estados finales del ciclo de vida.

A cada actividad de asocia una entidad que almacene su estado interno dentro de su ciclo de vida. Para ello, se declaran dos tipos de entidad, uno para cada uno de los tipos de ciclo de vida. La figura 7.3 muestra la declaración de ambos tipos de entidad. El estado se almacena

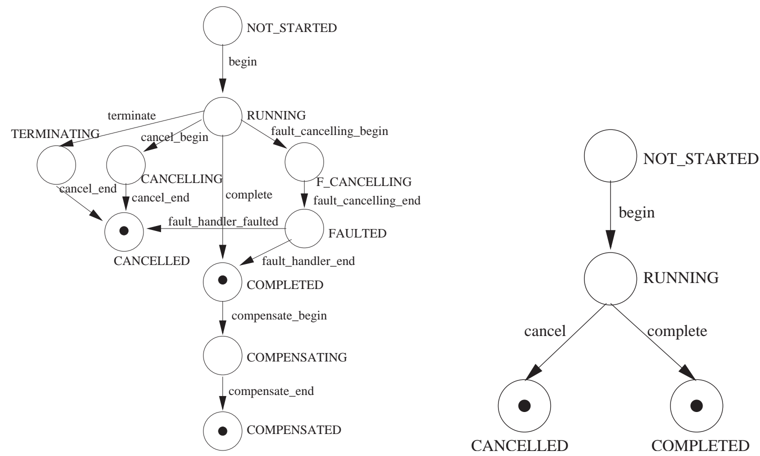

Figura 7.2: Ciclo de vida general (izda.) y simplificado (dcha.) de las actividades BPEL4WS.

```
enttype BPEL_lc_basic {
  state: enum (not_started, running, completed, cancelled);
}
enttype BPEL_lc {
  state: enum (not_started, running, completed, compensating,
               compensated, faulted, fault_cancelling,
               cancelling, terminating, cancelled);
  fault: enum (none);
  fault_handler: enum (none, unnamed_act_1);
  compensate: boolean;
}
```

Figura 7.3: Tipos de entidad que representan el ciclo de vida de actividades.

en el campo llamado *state*. Los campos *fault*, *fault_handler* y *compensate* se utilizan para implementar los mecanismos de manejo de fallos y de compensación. Su utilidad se describe en los apartados correspondientes a estos mecanismos.

Comportamiento

El comportamiento de las actividades BPEL4WS, y su evolución por el ciclo de vida, se modela mediante transiciones funcionales. En este apartado se exponen las dos transiciones funcionales con las que comúnmente se representa toda actividad BPEL4WS. Las transiciones concretas que modelan cada tipo de actividad BPEL4WS se presentan en los próximos apartados.

En general, son dos las transiciones funcionales comunes a las actividades BPEL4WS. Una de ellas representa el inicio de la ejecución de la actividad. La otra, su finalización:

- Transición funcional de inicio (*begin*):

  - Dominio: se compone, en principio, mediante una *o–lógica* de *y–lógicas*. La primera *y–lógica* es inicializada por la actividad padre de la propia actividad. Se establece dependiendo del tipo de actividad padre, y de la posición de la actividad

con respecto a otras relacionadas. Adicionalmente, se añade a esta primera *y–lógica* una condición que establece que el estado de la actividad sea *not_started*.

  - Acción: es una *y–lógica*. Establece el valor *running* en el estado de la actividad. Adicionalmente, en algunos tipos de actividades se añaden más expresiones a la acción.

- Transición funcional de finalización (*complete)*:

  - Dominio: establece como condición que el estado de la actividad debe ser *running*. Adicionalmente, determinados tipos de actividades introducen más condiciones.
  - Acción: establece estado *completed* en la actividad. Algunos tipos de actividades pueden añadir más expresiones de asignación.

Identificadores

Existen dos identificadores asociados a cada actividad definida en el modelo CFM: el identificador de la propia actividad y el de la entidad que almacena su estado dentro de su ciclo de vida. Estos identificadores se construyen de la siguiente forma:

- Identificador de actividad: se concatenan, en este orden, el nombre de la actividad BPEL4WS a representar (o el nombre del tipo de actividad, si no tiene un atributo *name* asociado), la cadena "_act_" y el número de orden de la actividad BPEL4WS (con respecto al orden seguido durante la conversión). Si el nombre de la actividad BPEL4WS contiene caracteres ilegales en un identificador CFM (véase la definición de *Id* en el apartado 4.3), se reemplaza cada uno de los mismos por el carácter "_" (*subrayado*).

- Identificador de la entidad: se concatena, a continuación del nombre de la actividad CFM, la cadena "_lc__".

Notación

En los próximos apartados se definen las acciones y dominios de distintas transiciones funcionales. Con el fin de facilitar la especificación y comprensión de los mismos, se utiliza la siguiente notación:

- $dom(a, t)$: dominio de la transición funcional t de la actividad a.

- $act(a, t)$: acción de la transición funcional t de la actividad a.

- $dom_i(a, t)$: condiciones intrínsecas a la propia actividad a a añadir al dominio de la transición funcional t.

- $dom_e(a, t)$: condiciones extrínsecas a la actividad a a añadir al dominio de la transición funcional t.

- $act_i(a, t)$: condiciones intrínsecas a la propia actividad a a añadir a la acción de la transición funcional t.

- $act_e(a,t)$: condiciones extrínsecas a la actividad a a añadir a la acción de la transición funcional t.

Las condiciones intrínsecas son aquellas debidas a la propia naturaleza de la actividad y, por tanto, dependientes del tipo de actividad BPEL4WS que se esté representando. Las condiciones extrínsecas son, por el contrario, debidas al contexto en el cual se ejecute la actividad. No dependen de la naturaleza de la propia actividad, sino de otras actividades de su entorno, normalmente la actividad inmediatamente superior en el árbol de actividades del proceso.

Utilizando la notación anterior, se define el dominio y la acción de una transición funcional t de una actividad a de la siguiente forma (se denota la igualdad de asignación como "::=" para evitar confusiones con las igualdades en las expresiones de dominios y acciones):

$$dom(a,t) ::= dom_i(a,t) \wedge dom_e(a,t) \quad (7.1)$$

$$act(a,t) ::= int_i(a,t) \wedge int_e(a,t) \quad (7.2)$$

Dada una actividad estructurada denotada como a, las subactividades contenidas directamente en esta actividad se denotarán normalmente como $a_0, a_1, \ldots, a_{n-1}$, donde n representa el número de subactividades. La actividad *scope* inmediatamente superior a la actividad a (o el propio proceso, si a no está dentro de ninguna actividad *scope*) se denota normalmente como s. La actividad CFM que representa al proceso global se denota como p.

7.2.4. Expresiones XPath

La especificación de BPEL4WS proporciona un mecanismo extensible para definir el lenguaje de expresiones utilizado en las definiciones de procesos. El lenguaje por defecto, obligatorio para toda aplicación que procese BPEL4WS, es XPath 1.0 [128]. En este apartado se expone cómo se transforman expresiones XPath al lenguaje de expresiones del formalismo CFM, cuya sintaxis se especifica en el apartado 4.3.

Expresiones generales XPath

Una expresión general XPath se puede representar como un árbol binario de expresiones, donde cada nodo del árbol representa un operador y cada rama un operando. Este árbol se puede transformar de forma recursiva a una expresión CFM: cada operador XPath se transforma al operador CFM equivalente, y cada uno de sus operandos, que es una expresión XPath, se transforma recursivamente a una expresión CFM. La recursión finaliza cuando se alcance una constante o una referencia a variable, que se transforma directamente a CFM.

Las constantes enteras y *Booleanas* de XPath se transforman a los mismos tipos de datos en CFM. Las constantes de tipo *cadena de texto* de XPath se transforman a valores enumerados en CFM. Estas últimas deben aparecer siempre en una comparación de igualdad o desigualdad. El otro extremo de esta comparación debe ser una referencia a variable de tipo enumerado.

Funciones específicas de BPEL4WS

La especificación de BPEL4WS define tres funciones XPATH que pueden ser utilizadas en definiciones de procesos:

- *bpws:getVariableProperty('variableName', 'propertyName')*: devuelve el contenido de un fragmento de una variable referenciado a través de una *propiedad*.
- *bpws:getVariableData('variableName', 'partName'?, 'locationPath'?)*: devuelve el contenido de una variable o fragmento definido por el nombre de una parte y una expresión XPath.
- *bpws:getLinkStatus('linkName')*: devuelve un valor *Booleano* indicando el valor al cual se ha evaluado el vínculo dado.

La transformación de las primeras dos funciones es, en general, bastante complejo. Dado un nombre de variable BPEL4WS, es necesario recuperar la *entidad* correspondiente, así como la definición de su *tipo de entidad*. De entre todos sus campos, se seleccionan aquellos que coincidan con el criterio de búsqueda. Si se trata de una consulta de la variable completa, se seleccionan todos los campos. Si se delimita la búsqueda por nombre de propiedad, o por nombre de parte y camino XPath, se seleccionan los campos correspondientes. Dependiendo de si se selecciona un único campo o más de uno, el mecanismo de transformación difiere.

- Si es una referencia a variable o fragmento que selecciona más de un campo: su uso está restringido a expresiones de comparación de igualdad o desigualdad. El otro extremo de la comparación debe ser también una variable o fragmento compatible, esto es, procedente del mismo tipo de datos XML Schema. En este caso, se genera una expresión *y–lógica* de CFM. Para cada campo de la variable izquierda de la comparación, se añade a la *y–lógica* una cláusula que lo compara con el campo correspondiente en la variable derecha.
- Si es una referencia a variable o fragmento que selecciona un único campo: se crea una referencia a variable (regla de producción *VarRef* según se define en la notación de CFM).

En el apartado 7.2.22 se expone cómo se transforma la función de evaluación de un vínculo, una vez explicado cómo se transforman los vínculos a CFM.

Ejemplos

En los ejemplos propuestos para las actividades *switch* (figura 7.9) y *while* (figura 7.10) se muestra algún ejemplo de transformación de expresiones XPath al formalismo CFM.

7.2.5. Procesos

Un proceso BPEL4WS se representa mediante el propio concepto de proceso de CFM (con el mismo nombre que el proceso BPEL4WS). Además, dado que el proceso también responde al ciclo de vida general de las actividades, se representa también mediante una actividad CFM. El identificador de la actividad es la concatenación de las cadenas "process_", el nombre del proceso y la cadena "_act__".

Una diferencia fundamental de esta actividad con respecto a las demás es que no tiene transición funcional de inicio. Esto se debe a que la instancia del proceso no comienza hasta

que se reciba un primer mensaje (véase la representación de la actividad *receive*, en el apartado 7.2.11). Es esta primera actividad de recepción de mensajes la que marca el proceso como iniciado.

La transición funcional de finalización del proceso se activa cuando su actividad principal, denotada a continuación como a_0, finalice.

$$\begin{array}{lll} dom_i(p, end) & ::= & (p.state = running) \wedge (a_0.state = completed) \\ act_i(p, end) & ::= & (p.state = completed) \end{array} \tag{7.3}$$

Con respecto a manejo de fallos, compensación, eventos y cancelación, las transiciones son similares a las de un ámbito (actividad *scope*). Véase el apartado 7.2.16.

7.2.6. Actividad *assign*

La actividad *assign* permite realizar una o más copias de datos desde una variable, datos literales o expresiones XPath a otra variable. No es necesario que las copias de datos pertenezcan a variables completas: se pueden realizar copias de fragmentos de variables. Las asignaciones están sujetas a comprobación estricta de tipos de los datos origen y de la variable o fragmento destino.

Cada copia a realizar dentro de una actividad *assign* se representa mediante un elemento *copy*. A su vez, este contiene un elemento *from* que especifica el origen de los datos y un elemento *to* que especifica el destino. Existen varias formas de especificar el origen y destino de los datos:

- Origen de los datos:
  - Variable y parte: se indica el nombre de una variable y, opcionalmente, una parte.
  - Variable y propiedad: se indica el nombre de una variable y el de una propiedad.
  - Enlace a socio: el origen de la copia no es una variable, sino una ligadura de servicios Web.
  - Expresión XPath: el origen de los datos es el resultado de evaluar la expresión XPath dada.
  - Literal: se especifican los datos de forma literal como contenido del elemento *to*.
- Destino de los datos:
  - Variable y parte: se indica el nombre de una variable y, opcionalmente, una parte.
  - Variable y propiedad: se indica el nombre de una variable y el nombre de una propiedad.
  - Enlace a socio: el destino de la copia no es una variable, sino una ligadura de servicios Web.

Representación de asignaciones

Una asignación de datos de un origen de datos hacia un destino se representa en CFM mediante una *y–lógica* de expresiones de asignación entre campos de entidades. Para ello, se sigue el siguiente procedimiento:

1. Se determina el origen de datos y el tipo de datos CFM con que se ha representado.
2. Se determina el conjunto de campos que han de ser asignados y sus valores representados en CFM (literales o expresiones *VarRef*).
3. Se comprueba si el destino tiene un tipo de datos compatible para la asignación.
4. Se determinan los campos a los cuales se realiza la copia en la variable destino.
5. Se construye la *y–lógica* de asignación, en la cual se asigna a cada campo de la variable destino el valor correspondiente del origen.

Transiciones funcionales

En la especificación de BPEL4WS se indica que se puede considerar despreciable la duración de una actividad *assign*. Por tanto, es suficiente modelar esta actividad con una única transición funcional que represente tanto su inicio como su finalización. Las asignaciones se incluyen en la acción de esta transición funcional.

Si se denota como a esta actividad y como $assign_0, \ldots, assign_{n-1}$ las *y–lógicas* de asignación correspondientes a cada uno de los elementos *copy* de esta actividad, se define esta transición funcional de la siguiente forma:

$$\begin{array}{lcl} dom_i(a, assign) & ::= & (a.state = not_started) \\ act_i(a, assign) & ::= & (a.state = running) \wedge assign_0 \wedge \ldots \wedge assign_{n-1} \end{array} \quad (7.4)$$

Es posible que alguna de las asignaciones entre en conflicto con las anteriores, provocando que se intente asignar más de un valor al mismo fragmento de una variable. Algunas posibilidades, cuando se detecte esto, son lanzar un fallo o hacer que prevalezca sólo una de las asignaciones, según el orden en que se especifiquen. Sin embargo, la especificación de BPEL4WS no indica qué se debe hacer en ese caso. En este entorno, se ha decidido que, mientras este aspecto de la especificación no sea aclarado, prevalecerá la última asignación. Para ello, se añade un procesado extra a la generación de la acción que compruebe la existencia de asignaciones cuya referencia a variable destino coincida. Debe eliminar estas asignaciones, exceptuando la correspondiente al elemento *assign* definido en último lugar.

Ejemplo

Para ilustrar los conceptos expuestos en este apartado, la figura 7.6 muestra un ejemplo de transformación de una actividad *assign* con dos elementos *copy*. Se basa en las variables y tipos de datos de la figura 7.5.

7.2.7. Actividad *sequence*

La representación de esta actividad se realiza siguiendo los principios expuestos en las líneas generales (apartado 4.4.6). La diferencia con dichas líneas está en que la propia actividad de secuencia se representa explícitamente con una actividad CFM. Cuando esta actividad comienza su ejecución, se permite el inicio de su primera actividad interna. Cuando su última actividad interna finaliza su ejecución, se finaliza la ejecución de la propia actividad. Las actividades internas se representan recursivamente.

Se modela el comportamiento de la actividad *sequence* mediante las transiciones funcionales *begin* y *end*. Si a representa una actividad *sequence*, sus transiciones funcionales de a vienen dadas por:

$$\begin{array}{lll} dom_i(a, begin) & ::= & (a.state = not_started) \\ act_i(a, begin) & ::= & (a.state = running) \\ dom_i(a, end) & ::= & (a.state = running) \wedge (a_{n-1}.state = completed) \\ act_i(a, end) & ::= & (a.state = completed) \end{array} \tag{7.5}$$

Por otra parte, se modifica la condición extrínseca del dominio de la transición funcional de inicio de las actividades internas (hijas directas de la actividad de secuencia) de la siguiente forma:

$$dom_e(a_i, begin) \quad ::= \quad \begin{cases} a.state = running, & i = 0 \\ a_{i-1}.state = completed, & 0 < i < n \end{cases} \tag{7.6}$$

Ejemplo

La figura 7.7 ilustra la transformación de actividades *sequence* mediante un ejemplo.

7.2.8. Actividad *flow*

La representación de esta actividad se realiza siguiendo los principios expuestos en las líneas generales (apartado 4.4.7). La diferencia con dichas líneas está en que la propia actividad de secuencia se representa explícitamente con una actividad CFM. Cuando esta actividad comienza su ejecución, se permite el inicio de todas sus actividades internas. Cuando todas ellas finalizan su ejecución, se finaliza la ejecución de la propia actividad. Las actividades internas se representan recursivamente.

Se modela el comportamiento de la actividad *flow* mediante las transiciones funcionales *begin* y *end*. Si a representa una actividad *flow*, las transiciones funcionales de a vienen dadas por:

$$\begin{array}{lll} dom_i(a, begin) & ::= & (a.state = not_started) \\ act_i(a, begin) & ::= & (a.state = running) \\ dom_i(a, end) & ::= & (a.state = running) \wedge \\ & & \wedge (a_0.state = completed) \wedge \ldots \wedge (a_{n-1}.state = completed) \\ act_i(a, end) & ::= & (a.state = completed) \end{array} \tag{7.7}$$

Por otra parte, se modifica la condición extrínseca del dominio de la transición funcional de inicio de las actividades internas (hijas directas de la actividad *flow*) de la siguiente forma:

$$dom_e(a_i, begin) \quad ::= \quad (a.state = running), \quad 0 \leq i < n \tag{7.8}$$

Ejemplo

La figura 7.8 ilustra la transformación de actividades *flow* mediante un ejemplo.

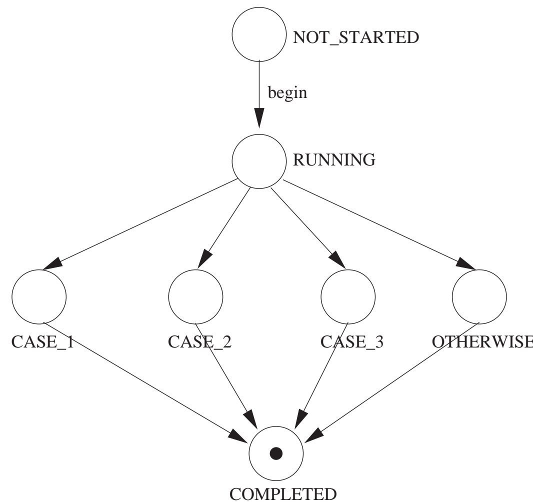

Figura 7.4: Evolución de la actividad *switch*

7.2.9. Actividad *switch*

La actividad *switch* es una de las actividades de BPEL4WS cuya representación en el formalismo CFM resulta más compleja, tanto en términos de número de transiciones funcionales como de entidades y tipos de entidades asociados.

Por una parte, además de la entidad de representación de su estado interno, se añade una nueva entidad para almacenar el caso seleccionado. Su identificador se construye a partir del identificador de la propia actividad, concatenándole la cadena "_switch_sel__". El tipo de entidad correspondiente tiene el mismo nombre que la entidad. Contiene un campo llamado *selected*, cuyo valor es el nombre asignado al caso seleccionado, o *none* mientras no se haya seleccionado ninguno. Los casos se nombran concatenando "case_" y el número de caso. El caso por defecto se nombra como "otherwise". A continuación se muestra un ejemplo para una actividad con tres casos, uno de ellos por defecto:

```
enttype my_switch_act_3_switch_sel__ {
  selected: enum (none, case_0, case_1, otherwise);
}
entity my_switch_act_3_switch_sel__: my_switch_act_3_switch_sel__;
```

En cuanto a transiciones funcionales, la actividad *switch* se representa mediante una de inicio y, para cada caso definido en la actividad, una de finalización y otra, intermedia, de selección. La figura 7.4 muestra un esquema de cómo evoluciona esta actividad. Cada transición funcional de selección evalúa la condición de un caso. Sólo se ejecuta una de las transiciones intermedias, la correspondiente al caso seleccionado. Una vez se ha ejecutado la transición de selección de un caso, la actividad se bloquea hasta que la actividad principal de dicho caso finaliza, momento en el cual se ejecuta la transición funcional de finalización correspondiente a dicho caso.

A continuación se define cada una de estas transiciones funcionales. Sea a la actividad *switch*, $a_0, a_1, \ldots, a_{n-1}$ las actividades asociadas a sus casos y a' la actividad del caso por defecto. Nótese que no todas las actividades *switch* tienen necesariamente un caso por defecto. Sean $c_0, c_1, \ldots, c_{n-1}$ las condiciones asociadas a cada uno de los casos. Sea e la entidad

que representa el caso seleccionado. La transición funcional de inicio (*begin*) se representa de la siguiente forma:

$$\begin{array}{lll} dom_i(a, begin) & ::= & (a.state = not_started) \\ act_i(a, begin) & ::= & (a.state = running) \wedge (e.selected = none) \end{array} \tag{7.9}$$

La representación de las transiciones funcionales asociadas a cada caso es la siguiente. Para el caso por defecto es similar, pero en la primera definición desaparece el término c_i y se incluye la *y–lógica* de las condiciones negadas de todos los casos.

$$\begin{array}{lll} dom_i(a, begin_case_i) & ::= & (a.state = running) \wedge (e.selected = none) \wedge \\ & & \wedge c_i \wedge \bigwedge_{k=0}^{i-1} \neg c_k \\ act_i(a, begin_case_i) & ::= & (e.selected = case_i) \\ dom_i(a, complete_case_i) & ::= & (a.state = running) \wedge \\ & & \wedge (e.selected = case_i) \wedge (a_i.state = completed) \\ act_i(a, complete_case_i) & ::= & (a.state = completed) \wedge (e.selected = none) \end{array} \tag{7.10}$$

La condición extrínseca del dominio de la transición funcional de inicio de la actividad principal de cada uno de los casos se define como:

$$dom_e(a_i, begin) \quad ::= \quad (e.selected = case_i), 0 \leq i < n \tag{7.11}$$

Las condiciones en una actividad *switch* se expresan mediante XPath. En el apartado 7.2.4 se explica cómo se transforman estas expresiones al formalismo CFM.

Ejemplo

La figura 7.9 muestra un ejemplo de transformación de una actividad *switch*. La actividad del ejemplo contiene dos alternativas: una cláusula *case* y una cláusula *otherwise*. La condición asociada a la primera cláusula establece que el precio recibido en la variable *response* sea menor o igual que el precio máximo establecido en la variable *request*, o bien el número de días de espera de la variable *response* sea igual al plazo establecido en la variable *request*. Por simplicidad, se representa la actividad interna de cada una de las alternativas mediante actividades *empty*.

7.2.10. Actividad *while*

La actividad *while* se modela mediante tres transiciones funcionales. Dos de ellas son las convencionales de inicio y finalización. Las tercera se ejecuta cada vez que se inicia una nueva iteración, e incluye la evaluación de la condición del bucle. Una característica que diferencia a esta actividad de las demás es que debe restablecer los valores iniciales en todas sus actividades internas, en cualquier nivel de profundidad, antes de comenzar cada iteración. Esto es así porque se reutiliza, de una iteración a otra, la misma actividad CFM, y este formalismo no permite la instanciación múltiple de una misma actividad (véase el apartado 4.5.4).

Si se denota como a la actividad *while*, como a_0 su actividad interna (en primer nivel de jerarquía), $a_1, a_2, \ldots, a_{n-1}$ el resto de actividades internas (en cualquier nivel de jerarquía),

$lnk_0, lnk_1, \ldots, lnk_{m-1}$ todos los vínculos declarados en actividades *flow* internas al bucle y c la condición del bucle, las transiciones funcionales se construyen de la siguiente forma:

$$\begin{array}{lll} dom_i(a, begin) & ::= & (a.state = not_started) \\ act_i(a, begin) & ::= & (a.state = running) \wedge reset \\ dom_i(a, complete) & ::= & (a.state = running) \wedge (a_0.state = completed) \wedge \neg c \\ act_i(a, complete) & ::= & (a.state = completed) \\ dom_i(a, continue) & ::= & (a.state = running) \wedge (a_0.state = completed) \wedge c \\ act_i(a, continue) & ::= & reset \\ reset & ::= & \bigwedge_{k=0}^{n-1}(a_k.state = not_started) \wedge \bigwedge_{i=0}^{m-1}(\neg lnk_i.evaluated) \end{array} \tag{7.12}$$

La actividad interna del bucle, a_0, se puede ejecutar siempre y cuando la condición de continuación sea cierta y el propio bucle se encuentre en ejecución:

$$dom_e(a_0, begin) \quad ::= \quad (a.state = running) \wedge c \tag{7.13}$$

Al igual que en el caso de la actividad *switch*, la condición de continuación de la actividad *while* se expresa mediante XPath. En el apartado 7.2.4 se explica cómo se transforman estas expresiones al formalismo CFM.

Ejemplo

En la figura 7.10 se muestra un ejemplo de transformación de la actividad *while*. Se ejecuta mientras el número de días de espera, almacenado en la variable *response*, sea menor que el plazo establecido en la variable *request*, y además el precio, contenido en la variable *oPrice*, sea mayor que el precio máximo establecido en la variable *request*. En cada iteración se ejecuta una secuencia de tres actividades.

7.2.11. Actividad *receive*

Una actividad *receive* modela la recepción de una invocación a una operación de un servicio proporcionado por el proceso. Una vez comienza su ejecución, se bloquea hasta que se reciba el mensaje de invocación. El instante de recepción del mensaje, así como el contenido del mismo, no dependen directamente del proceso, sino de la entidad externa que realiza la invocación.

Desde el punto de vista de la verificación, no resulta relevante el instante concreto de llegada de la invocación, sino su relación (anterior o posterior) con otros sucesos, ya sean internos del proceso o externos. Por tanto, es suficiente con permitir cualquier ordenación posible de la llegada del mensaje de invocación con respecto a otros sucesos que pudiesen ocurrir concurrentemente en el proceso.

Por otra parte, es necesario delimitar un rango de valores para el mensaje de entrada de la invocación. En el apartado 7.2.2 se discute el modelado de las entidades externas desde el punto de vista de los mensajes que envían al proceso, mediante la declaración de *mensajes alternativos*.

Cuando no se especifiquen mensajes alternativos de entrada, se representa esta actividad mediante dos transiciones funcionales, una de inicio y otra de finalización. En este caso, el

instante de ejecución de la transición de finalización representa el instante de llegada del mensaje. Se definen las transiciones de la siguiente forma:

$$\begin{array}{lll} dom_i(a, begin) & ::= & (a.state = not_started) \\ act_i(a, begin) & ::= & (a.state = running) \\ dom_i(a, complete) & ::= & (a.state = running) \\ act_i(a, complete) & ::= & (a.state = completed) \end{array} \tag{7.14}$$

Representación de mensajes alternativos

Si se definen mensajes alternativos en la actividad *receive*, se representa cada una de las alternativas mediante una transición funcional de finalización distinta. La representación depende de si la alternativa se modela mediante mensajes constantes o mediante código BPEL4WS.

En cualquier caso, es necesario definir una entidad y un tipo de entidad adicionales para representar la alternativa seleccionada. Esto permite controlar eficazmente la ejecución de las actividades que definen cada mensaje alternativo. El tipo de entidad contiene un campo de tipo enumerado con valores *none* (ningún mensaje, de entre los definidos mediante actividad, seleccionado) y un valor para cada uno de los mensajes alternativos definidos mediante actividades, tal y como se muestra en este ejemplo:

```
enttype receive_act_3_altmsg__ {
  selected: enum (none, alternative_0, alternative_1);
}
entity receive_act_3_altmsg__: receive_act_3_altmsg__;
```

En la transición de inicio de la actividad se inicializa esta entidad con valor *none* (e denota a la entidad):

$$\begin{array}{lll} dom_i(a, begin) & ::= & (a.state = not_started) \\ act_i(a, begin) & ::= & (a.state = running) \wedge (e.selected = none) \end{array} \tag{7.15}$$

Si el mensaje se especifica como una constante, se construye una expresión de asignación de dicho mensaje a la variable en la cual debe ser almacenado. En el apartado 7.2.6, que expone cómo se representa la actividad *assign*, se describe en detalle cómo se genera esta expresión. Si se denota a la expresión de asignación del mensaje alternativo i–ésimo como $assign_i$, la transición funcional de finalización correspondiente a dicho mensaje es:

$$\begin{array}{lll} dom_i(a, complete_i) & ::= & (a.state = running) \wedge (e.selected = none) \\ act_i(a, complete_i) & ::= & (a.state = completed) \wedge assign_i \end{array} \tag{7.16}$$

Si el mensaje se especifica mediante una actividad BPEL4WS, es necesario definir una transición funcional intermedia para cada mensaje alternativo. Esta transición funcional representa la selección de uno de los mensajes. Bloquea la selección de cualquier otro, y permite la ejecución de la actividad interna. Si la actividad interna para la alternativa i–ésima

se denota como a_i:

$$\begin{array}{lll} dom_i(a, select_i) & ::= & (a.state = running) \wedge (e.selected = none) \\ act_i(a, select_i) & ::= & (e.selected = alternative_i) \\ dom_i(a, complete_i) & ::= & (a.state = running) \wedge (e.selected = alternative_i) \wedge \\ & & \wedge (a_i.state = completed) \\ act_i(a, complete_i) & ::= & (a.state = completed) \wedge (e.selected = none) \end{array} \tag{7.17}$$

En este caso, la actividad que construye el valor de este mensaje se puede ejecutar sólo si ha sido seleccionado el mensaje alternativo correspondiente:

$$dom_e(a_i, begin) \quad ::= \quad (a.state = running) \wedge (e.selected = alternative_i) \tag{7.18}$$

Creación de instancias del proceso

Las actividades de recepción de mensajes (*pick* y *receive*) son las únicas capaces de crear instancias del proceso. Para ello, deben contener un atributo *createInstance* con valor “*yes*”. De hecho, todo proceso debe comenzar obligatoriamente con una de estas dos actividades con dicho atributo activado.

La creación de la instancia del proceso se representa cambiando el estado del proceso de *not_started* a *running*, en el instante de recepción del mensaje. Por tanto, cuando sea necesario en una actividad *receive* crear una nueva instancia del proceso, a todas sus transiciones de finalización se añade la asignación $p.state = running$ mediante una *y–lógica*.

Ejemplo

En la figura 7.11 se muestra un ejemplo de transformación de una actividad *receive*. En esta actividad se especifican dos posibles mensajes de entrada alternativos. Por otra parte, la recepción del mensaje debe crear una nueva instancia del proceso, dado que el atributo *createInstance* toma valor “*yes*”.

7.2.12. Actividad *pick*

Aunque una actividad *pick* permite, al igual que *receive*, recibir invocaciones a operaciones desde entidades remotas, su representación es considerablemente más compleja debido, fundamentalmente, a tres factores. Por una parte, se pueden recibir invocaciones para distintas operaciones. Por otra parte, también es posible que se active la actividad debido al vencimiento de algún temporizador. Además, se asocia la ejecución de una actividad interna a la recepción de cada mensaje o vencimiento de cada temporizador.

La representación de esta actividad mediante transiciones funcionales es relativamente similar al modelado de la actividad *switch*, presentado en el apartado 7.2.9. Se declara una transición funcional de inicio. Para cada una de las operaciones que pueden ser invocadas o alarmas que pueden vencer, se declaran una transición funcional intermedia y otra de finalización. Por otra parte, la posibilidad de especificación de mensajes alternativos (véanse los apartados 7.2.2 y 7.2.11) introduce más transiciones funcionales.

Adicionalmente, se representa el suceso ocurrido mediante una entidad y un tipo de entidad. En el resto de este apartado se denota como e la entidad de selección asociada a la actividad.

```
enttype pick_act_5_pick_sel__ {
  activated: enum (none, operation_0, operation_1, ..., onAlarm_0, onAlarm1, ...);
}
entity pick_act_5_pick_sel__: pick_act_5_pick_sel__;
```

Si alguna de las cláusulas *onMessage* especifica algún mensaje alternativo definido mediante una actividad, es necesario declarar una nueva entidad y tipo de entidad para representar el mensaje alternativo seleccionado. Esta entidad puede ser compartida por todas las cláusulas *onMessage* de una misma actividad *pick*. El primer índice de cada valor de la enumeración indica de qué *onMessage* se trata, y el segundo de qué mensaje alternativo. Sólo es necesario introducir en el rango de valores los mensajes alternativos especificados mediante actividad.

```
enttype pick_act_5_altmsg__ {
  activated: enum (none, alternative_0_0, alternative_0_1, ...);
}
entity pick_act_5_altmsg__: pick_act_5_altmsg__;
```

La transición funcional de inicio se define de la forma habitual. En ella se inicializa el valor de la entidad de selección.

$$\begin{array}{lll} dom_i(a, begin) & ::= & (a.state = not_started) \\ act_i(a, begin) & ::= & (a.state = running) \wedge (e.activated = none) \end{array} \tag{7.19}$$

Cada cláusula *onAlarm* se modela mediante dos transiciones funcionales. Nótese que, desde el punto de vista de la verificación, no es relevante el tiempo transcurrido. Por tanto, se ignora este tiempo en la representación[3]. Se denota como a_i la actividad contenida en la i–ésima cláusula *onAlarm*.

$$\begin{array}{lll} dom_i(a, begin_operation_i) & ::= & (a.state = running) \wedge (e.activated = none) \wedge \\ & & \wedge (m.selected = none) \\ act_i(a, begin_onAlarm_i) & ::= & (e.activated = onAlarm_i) \\ dom_i(a, complete_onAlarm_i) & ::= & (a.state = running) \wedge (e.activated = onAlarm_i) \wedge \\ & & \wedge (a_i.state = completed) \\ act_i(a, complete_onAlarm_i) & ::= & (a.state = completed \wedge e.activated = none) \\ dom_e(a_i, begin) & ::= & (a.state = running) \wedge (e.activated = onAlarm_i) \end{array} \tag{7.20}$$

Una cláusula *onMessage* para la cual no se propongan mensajes alternativos se representa de forma similar a una cláusula *onAlarm*:

$$\begin{array}{lll} dom_i(a, begin_onAlarm_i) & ::= & (a.state = running) \wedge (e.activated = none) \wedge \\ & & \wedge (m.selected = none) \\ act_i(a, begin_operation_i) & ::= & (e.activated = operation_i) \\ dom_i(a, complete_operation_i) & ::= & (a.state = running) \wedge (e.activated = operation_i) \wedge \\ & & \wedge (a_i.state = completed) \\ act_i(a, complete_operation_i) & ::= & (a.state = completed) \wedge (e.activated = none) \\ dom_e(a_i, begin) & ::= & (a.state = running) \wedge (e.activated = operation_i) \end{array} \tag{7.21}$$

[3] En realidad, puede ser relevante si se especifican dos o más eventos *onAlarm*. En ese caso, es posible que se sepa, en tiempo de diseño, que algunas de estas alarmas no pueden ser ejecutadas nunca. Estas alarmas deben ser descartadas en la transformación a CFM.

Representación de mensajes alternativos

La representación de una cláusula *onMessage* con mensajes alternativos asociados es considerablemente más compleja. La transición que representa la finalización de la cláusula *onMessage* es igual a la presentada para casos en que no se especifique mensajes alternativos. La condición extrínseca añadida al dominio de la transición funcional de inicio de la actividad asociada a la actividad interna de la cláusula es también igual. Lo que difiere es la transición de selección de cláusula y mensaje alternativo.

Para cada caso alternativo en que se especifique el mensaje como constante, se define la transición funcional intermedia de la siguiente forma. Se denota como i el índice de la cláusula *onMessage* y como j el número de mensaje alternativo dentro de dicha cláusula. Se denota como $assign_{i,j}$ la expresión que representa la asignación del mensaje constante a la variable correspondiente. Se denota como m la entidad que representa el mensaje alternativo seleccionado.

$$\begin{array}{lcl} dom_i(a, begin_operation_{i,j}) & ::= & (a.state = running) \wedge (e.activated = none) \wedge \\ & & \wedge (m.selected = none) \\ act_i(a, begin_operation_{i,j}) & ::= & assign_{i,j} \wedge (e.activated = operation_i) \end{array} \tag{7.22}$$

Para cada caso alternativo en que se especifique el mensaje mediante una actividad, se definen dos transiciones funcionales intermedias. Se denota como $b_{i,j}$ la actividad que define dicho mensaje alternativo.

$$\begin{array}{lcl} dom_i(a, begin_operation_{i,j}) & ::= & (a.state = running) \wedge (e.activated = none) \wedge \\ & & \wedge (m.selected = none) \\ act_i(a, begin_operation_{i,j}) & ::= & (m.selected = alternative_{i,j}) \\ dom_i(a, interm_operation_{i,j}) & ::= & (a.state = running) \wedge (m.selected = alternative_{i,j}) \wedge \\ & & \wedge (b_{i,j}.state = completed) \\ act_i(a, interm_operation_{i,j}) & ::= & (e.activated = operation_i) \wedge (m.selected = none) \\ dom_e(b_{i,j}, begin) & ::= & (a.state = running) \wedge (m.selected = alternative_{i,j}) \\ dom_e(a_i, begin) & ::= & (a.state = running) \wedge (e.activated = operation_i) \end{array} \tag{7.23}$$

Ejemplo

Las figuras 7.12 y 7.13 muestran un ejemplo de representación de la actividad *pick*. La primera muestra el fragmento BPEL4WS que define la actividad, mientras que la segunda muestra su representación mediante CFM. Esta actividad consta de una cláusula *onMessage*, en la cual se especifican dos mensajes alternativos, y una cláusula *onAlarm*. Las actividades internas de ambas cláusulas son, por mantener el ejemplo lo más simple posible, actividades *empty*.

7.2.13. Actividad *reply*

La actividad *reply* envía un mensaje de respuesta a una invocación recibida previamente por el proceso. La representación mediante CFM de esta actividad es sencilla, dado que se

abstraen los mecanismos de comunicación. Es suficiente modelar esta actividad con las dos transiciones funcionales habituales de inicio y finalización.

$$\begin{array}{lll} dom_i(a, begin) & ::= & (a.state = not_started) \\ act_i(a, begin) & ::= & (a.state = running) \\ dom_i(a, complete) & ::= & (a.state = running) \\ act_i(a, complete) & ::= & (a.state = completed) \end{array} \tag{7.24}$$

Si fuese necesario reducir el espacio de estados para reducir el tiempo de verificación, se podría recurrir a modelar esta actividad con una única transición funcional que inicie e, inmediatamente, finalice la actividad. Esto es equivalente a considerar que el tiempo de envío de mensajes es despreciable. Desde el punto de vista de la verificación, la única diferencia está en que, con una única transición funcional, no se contemplaría el caso en que la actividad fuese cancelada durante su ejecución.

7.2.14. Actividad *empty*

Una actividad *empty* representa una actividad nula, esto es, que no hace nada. Se modela únicamente con una transición funcional que representa su inicio y finalización instantánea. A esta transición se le llama *empty*.

$$\begin{array}{lll} dom_i(a, empty) & ::= & (a.state = not_started) \\ act_i(a, empty) & ::= & (a.state = completed) \end{array} \tag{7.25}$$

7.2.15. Actividad *wait*

La actividad *wait* bloquea su hilo de control durante un intervalo de tiempo o hasta una fecha y hora dados. Desde el punto de vista de la verificación se abstrae el concepto de tiempo, y sólo resulta relevante la ordenación parcial de sucesos. Por tanto, basta con modelar la actividad con las dos transiciones funcionales habituales de inicio y finalización. El tiempo transcurrido desde que se activa la transición funcional de inicio hasta que se activa la de finalización representa el tiempo durante el cual el hilo de control está bloqueado.

$$\begin{array}{lll} dom_i(a, begin) & ::= & (a.state = not_started) \\ act_i(a, begin) & ::= & (a.state = running) \\ dom_i(a, complete) & ::= & (a.state = running) \\ act_i(a, complete) & ::= & (a.state = completed) \end{array} \tag{7.26}$$

7.2.16. Actividad *scope*

Un ámbito se modela mediante la actividad *scope*. Esta es una actividad cuya representación es especialmente compleja debido a los mecanismos de manejo de fallos, compensación y eventos, así como su papel especial durante la cancelación de un proceso. En este apartado se profundiza en el modelado básico de esta actividad. En próximos apartados se profundiza en los cambios que es necesario introducir debido a los mecanismos citados anteriormente.

Variables

Las variables de un ámbito se declaran también como entidades del proceso, dado que el formalismo CFM no admite la declaración de entidades de ámbito local (véase el apartado 4.5.2). Para evitar colisiones con los nombres de entidades correspondientes a otros ámbitos, al nombre CFM de la entidad se le antepone la concatenación de los nombres de los ámbitos existentes entre el nivel superior del proceso y el ámbito en el cual se declara la variable. Por otra parte, para toda referencia a variable en el proceso BPEL4WS se determina cuál de las entidades globales le corresponde, teniendo en cuenta los ámbitos de declaración de cada una y el ámbito en que aparezca la referencia.

Transiciones funcionales

Un ámbito se modela con dos transiciones funcionales fundamentales: una que modela su inicio y otra que modela su finalización. Esta última es ejecutable sólo cuando su actividad interna finalice. La definición de estas transiciones funcionales es la siguiente:

$$\begin{array}{lll} dom_i(a, begin) & ::= & (a.state = not_started) \\ act_i(a, begin) & ::= & (a.state = running) \\ dom_i(a, complete) & ::= & (a.state = running) \wedge (a_0.state = completed) \\ act_i(a, complete) & ::= & (a.state = completed) \end{array} \quad (7.27)$$

Por otra parte, es necesario modelar el hecho de que la actividad interna no puede comenzar su ejecución hasta que haya comenzado la ejecución de la actividad *scope*:

$$dom_e(a_0, begin) \quad ::= \quad (a.state = running) \quad (7.28)$$

7.2.17. Actividad *invoke*

Una actividad *invoke* puede contener, al igual que un ámbito, manejadores de fallos y compensación. En este apartado se representa sólo el funcionamiento básico de la actividad. En próximos apartados se indican la modificaciones necesarias para los mecanismos de manejo de fallos y compensación.

La operación invocada puede ser síncrona o asíncrona. En el primer caso, la actividad debe bloquearse hasta recibir una respuesta desde la entidad que provee la operación. Al igual que en el resto de actividades que pueden recibir mensajes, el diseñador puede incluir una lista de mensajes alternativos que podrían ser recibidos.

Al igual que casi todas las actividades, se modela esta actividad, en principio, mediante dos transiciones funcionales: una de inicio y otra de finalización. Los mecanismos de gestión de excepciones y compensación pueden añadir a la actividad transiciones funcionales adicionales, así como el mecanismo de representación de mensajes alternativos.

$$\begin{array}{lll} dom_i(a, begin) & ::= & (a.state = not_started) \\ act_i(a, begin) & ::= & (a.state = running) \\ dom_i(a, complete) & ::= & (a.state = running) \\ act_i(a, complete) & ::= & (a.state = completed) \end{array} \quad (7.29)$$

Mensajes alternativos

Si se especifican mensajes alternativos para esta actividad, su representación es similar a la presentada en el apartado 7.2.11 para la actividad *receive*. Al igual que en el caso de dicha actividad, se añade una nueva transición funcional para cada mensaje alternativo especificado mediante actividad BPEL4WS.

Ejemplo

En la figura 7.14 se muestra un ejemplo de representación de una actividad *invoke*. La operación invocada es síncrona, y puede devolver dos posibles mensajes alternativos. A pesar de que puede ocurrir algún fallo en la invocación, no se representa, por simplicidad, la transición funcional que representa el lanzamiento del fallo. Se proporcionan ejemplos de lanzamiento de fallos en el apartado 7.2.20.

7.2.18. Cancelación

Tanto antes de ejecutar un manejador de fallos de un ámbito, como cuando se ejecuta una actividad *terminate*, es necesario cancelar todas las actividades que estén en ejecución en ese momento, respectivamente, en el proceso o en el ámbito. El ciclo de vida general (véanse las figuras 7.2 y 7.3) define, para ello, cuatro estados especiales para gestionar la cancelación de actividades: *cancelling*, *fault_cancelling*, *terminating* y *cancelled*.

Un ámbito (o proceso) cuya ejecución deba ser cancelada, se sitúa en estado *fault_cancelling* si la cancelación se debe a un fallo que él mismo debe manejar, en estado *cancelling* si se debe a un fallo a manejar en otro ámbito de nivel superior, o *terminating* si se debe a una terminación explícita del proceso mediante la actividad *terminate*. Se añade a todas las actividades una transición funcional que, cuando detecta que su ámbito inmediatamente superior está en uno de estos tres estados, inicia su propia cancelación, siempre y cuando esté en ese momento en ejecución.

En la actividad *scope*, se activa recursivamente el procedimiento de cancelación, situándose en estado *cancelling* y esperando a que hayan sido canceladas todas las actividades internas que estuviesen en ejecución. Supongamos que a es una actividad *scope*. Supongamos también que las actividades controladas por el ciclo de vida general cuyo ámbito inmediatamente superior es a son $b_0, \ldots, b_{m-1}$, y que las actividades controladas por el ciclo de vida simplificado cuyo ámbito inmediatamente superior es a son $c_0, \ldots, c_{k-1}$. El ámbito inmediatamente superior de a se denota como s. Las transiciones funcionales de cancelación

en ámbitos se construyen de la siguiente forma:

$$\begin{array}{lcl}
dom_i(a, cancel_begin) & ::= & ((a.state = running) \vee (a.state = faulted) \vee \\
 & & \quad \vee (a.state = compensating)) \wedge \\
 & & \wedge((s.state = cancelling) \vee (s.state = fault_cancelling)) \\
act_i(a, cancel_begin) & ::= & (a.state = cancelling) \\
dom_i(a, terminate) & ::= & ((a.state = running) \vee (a.state = faulted) \vee \\
 & & \quad \vee (a.state = compensating)) \wedge \\
 & & \wedge(s.state = terminating) \\
act_i(a, terminate) & ::= & (a.state = terminating) \\
dom_i(a, cancel_end) & ::= & ((a.state = cancelling) \vee (a.state = terminating)) \wedge \\
 & & \wedge \bigwedge_{i=0}^{m-1} (\quad (b_i.state = not_started) \vee \\
 & & \quad \vee (b_i.state = completed) \vee \\
 & & \quad \vee (b_i.state = compensated) \vee \\
 & & \quad \vee (b_i.state = cancelled)) \wedge \\
 & & \wedge \bigwedge_{i=0}^{k-1} \quad (c_i.state \neq running) \\
act_i(a, cancel_end) & ::= & (a.state = cancelled)
\end{array} \tag{7.30}$$

Las transiciones funcionales asociadas a la cancelación en caso de un fallo a ser manejado por el propio ámbito se presentan en el apartado 7.2.20.

En caso de actividades normales, la cancelación consiste simplemente en avanzar a estado *cancelled*. La especificación de BPEL4WS indica que no tiene sentido cancelar las actividades *empty*, *terminate* ni *throw*. También indica que, dada su corta duración, se puede permitir que la actividad *assign* finalice normalmente. Para el resto de actividades, la transición funcional de cancelación es la siguiente:

$$\begin{array}{lcl}
dom_i(a, cancel) & ::= & (a.state = running) \wedge ((s.state = cancelling) \vee \\
 & & \vee (s.state = fault_cancelling) \vee (s.state = terminating)) \\
act_i(a, cancel) & ::= & (a.state = cancelled)
\end{array} \tag{7.31}$$

Por otra parte, para acelerar la cancelación, se impide el comienzo de nuevas actividades cuando el proceso principal haya iniciado un procedimiento de cancelación. Para ello, se completa el dominio extrínseco de todas las actividades, a partir de aquella actividad de recepción de mensajes que cree una instancia del proceso, con la siguiente condición:

$$dom_e(a, begin) = \ldots \wedge p.state = running \tag{7.32}$$

Ejemplo

En la figura 7.15 se muestra un ejemplo de representación del procedimiento de *cancelación*, desde el punto de vista de una actividad *scope* (la que se llama *scope1*) y de sus actividades internas. Por sencillez, se omiten todas las transiciones funcionales excepto aquellas relacionadas directamente con el mecanismo de cancelación.

7.2.19. Actividad *terminate*

La implementación de la actividad *terminate* se basa en el mecanismo de cancelación. Es suficiente con una transición funcional que establezca el estado *terminating* en la actividad

CFM que representa al proceso (denotada como p a continuación). En ese momento se activa el mecanismo de cancelación en el resto de actividades mediante las transiciones funcionales definidas al efecto.

$$\begin{array}{lll} dom_i(a, terminate) & ::= & (a.state = not_started) \\ act_i(a, terminate) & ::= & (a.state = completed) \wedge (p.state = terminating) \end{array} \quad (7.33)$$

El proceso debe esperar a que finalice el procedimiento de cancelación en todas sus actividades internas. Si p representa el proceso, $b_0, b_1, \ldots, b_{m-1}$ son aquellas actividades, controladas por el ciclo de vida general, cuyo *ámbito* inmediatamente superior es el proceso, y $c_0, c_1, \ldots, c_{k-1}$ son aquellas actividades, controladas por el ciclo de vida simplificado, cuyo *ámbito* inmediatamente superior es el proceso, se construye la transición funcional que realiza esta espera de la siguiente forma:

$$\begin{array}{lll} dom_i(a, terminate_end) & ::= & (p.state = terminating) \wedge \\ & & \wedge \bigwedge_{i=0}^{m-1} \quad (b_i.state = not_started) \vee \\ & & \qquad \vee (b_i.state = completed) \vee \\ & & \qquad \vee (b_i.state = compensated) \vee \\ & & \qquad \vee (b_i.state = cancelled)) \wedge \\ & & \wedge \bigwedge_{i=0}^{k-1} \quad (c_i.state \neq running) \\ act_i(a, terminate_end) & ::= & (p.state = cancelled) \end{array} \quad (7.34)$$

Ejemplo

En la figura 7.16 se muestra un ejemplo de representación de la actividad *terminate*. Por otra parte, se muestra la transición funcional *terminate_end* de la actividad que representa al proceso.

7.2.20. Manejo de fallos

El mecanismo de manejo de fallos se activa cuando alguna de las actividades del proceso genera un *fallo*. La secuencia de eventos que ocurren en ese momento es la siguiente:

1. Se genera un fallo en una actividad.
2. El fallo fluye de ámbito en ámbito hasta que alguno de ellos posea un manejador adecuado para capturarlo.
3. Cuando un ámbito capture el fallo:
   a) Se cancelan todas las actividades internas de dicho ámbito, en cualquier nivel de profundidad.
   b) Se ejecuta el manejador de fallos seleccionado.
   c) Se finaliza normalmente el ámbito.
4. Si el fallo alcanza el nivel del proceso sin ser capturado en ningún ámbito, incluido el propio proceso, se cancelan todas las actividades del proceso y se finaliza (anormalmente) su ejecución.

A continuación se expone una posible forma de modelar el mecanismo de manejo de fallos en el formalismo CFM.

Campos específicos en la entidad de ciclo de vida

La entidad que modela el ciclo de vida de una actividad con ciclo de vida complejo incorpora dos campos de tipo enumerado directamente relacionados con el manejo de fallos.

El primero de ellos, llamado *fault*, se utiliza para indicar qué fallo ha ocurrido. En la declaración de su rango de valores enumerado se incluye el nombre de todos los posibles fallos que pueden ocurrir en el proceso, además del valor *none* para indicar la no ocurrencia de fallo alguno. Cuando ocurre un fallo, se debe establecer el valor adecuado de este campo en la entidad correspondiente a los ámbitos a través de los cuales fluye el fallo.

El otro, llamado *fault_handler*, se utiliza para indicar el manejador de fallos seleccionado. Su rango de valores enumerado se construye mediante el nombre de la actividad principal de cada uno de los manejadores de fallos definidos en todo el proceso. Cuando ocurre un fallo, se determina cuál es el manejador de fallos adecuado para el mismo, y se establece el valor de este campo en la entidad correspondiente al ámbito en el cual se encuentre dicho manejador.

Transiciones funcionales

A cada ámbito en el que puedan ocurrir fallos se añaden varias transiciones funcionales para modelar el manejo de fallos. Si se revisa el ciclo de vida general (figura 7.2), se observa que la actividad debe evolucionar, en este orden, por los siguientes estados:

1. *fault_cancelling*: el ámbito cambia a este estado cuando se produce un fallo, y alguno de sus manejadores de fallos ha sido seleccionado. Permanece en este estado en espera de que todas sus actividades internas sean canceladas.

2. *faulted*: el ámbito evoluciona a este estado desde el estado anterior, una vez todas sus actividades internas han sido canceladas. En este momento, se permite la ejecución del manejador de fallos seleccionado.

3. *completed*: la actividad evoluciona a este estado cuando finaliza la actividad principal del manejador de fallos seleccionado. En este momento se da por finalizada la ejecución del ámbito. Desde el punto de vista de las actividades externas al ámbito, su finalización es normal. A pesar de ello, no puede ser compensado.

La evolución del ámbito a través de estos estados se modela con tres transiciones funcionales. Se denominan, en este orden, *fault_cancel_begin*, *fault_cancel_end* y *catch_completion*.

Las primera de estas transiciones funcionales simplemente cambia el estado del ámbito. La segunda debe esperar a que ninguna actividad interna esté en ejecución. Esto es, a que todas ellas alcancen el estado *not_started*, *completed*, *compensated* o *cancelled*. No es necesario hacer esta comprobación sobre actividades internas que dependan de otro ámbito de menor nivel, dado que dicho ámbito no alcanzará ninguno de estos estados mientras alguna de sus actividades internas esté en ejecución.

Si se denotan estas actividades internas, cuando están controladas por el ciclo de vida general, como $b_0, \ldots, b_{m-1}$ y, cuando están controladas por el ciclo de vida simplificado, como $d_0, \ldots, d_{k-1}$, estas dos transiciones funcionales se construyen de la siguiente forma:

$$
\begin{array}{lcl}
dom_i(a, fault_cancel_begin) & ::= & (a.fault \neq none) \wedge (a.state = running) \\
act_i(a, fault_cancel_begin) & ::= & (a.state = fault_cancelling) \\
dom_i(a, fault_cancel_end) & ::= & (a.state = fault_cancelling) \wedge \\
 & & \wedge \bigwedge_{i=0}^{m-1} (\quad (b_i.state = not_started) \vee \\
 & & \qquad \vee (b_i.state = completed) \vee \\
 & & \qquad \vee (b_i.state = compensated) \vee \\
 & & \qquad \vee (b_i.state = cancelled)) \wedge \\
 & & \wedge \bigwedge_{i=0}^{k-1} \quad (d_i.state \neq running) \\
act_i(a, fault_cancel_end) & ::= & (a.state = faulted)
\end{array}
\tag{7.35}
$$

Una vez ejecutada la transición funcional *fault_cancel_end*, puede comenzar la ejecución de la actividad asociada a la cláusula *catch* o *catchAll* seleccionada. La transición funcional *catch_end* finaliza el ámbito, una vez esta actividad finalice. Si las actividades principales de cada *catch* de la actividad a se denotan como $c_0, \ldots, c_{n-1}$, se representa esto como:

$$
\begin{array}{lcl}
dom_i(a, catch_completion) & ::= & (a.state = faulted) \wedge \\
 & & \wedge (\bigvee_{i=0}^{n-1} (c_i.state = completed)) \\
act_i(a, catch_completion) & ::= & (a.state = completed)
\end{array}
\tag{7.36}
$$

Cada actividad principal de un manejador de fallos se representa con una pre–condición extrínseca que establece que su ámbito debe estar en estado *faulted* y que el manejador de fallos al cual pertenece debe ser el seleccionado para ser ejecutado:

$$
dom_e(c_i, begin) \quad ::= \quad (a.state = faulted) \wedge (a.fault_handler = c_i), \ 0 <= i < n
\tag{7.37}
$$

Un fallo ocurrido en una actividad contenida dentro de un manejador de fallos debe ser reenviado al ámbito inmediatamente superior. Esto es, dados un ámbito A, y un ámbito B inmediatamente anidado en él, se puede considerar, desde el punto de vista de la representación de las actividades incluidas en los manejadores de fallos de B, que su ámbito inmediatamente superior es A.

Señalización de un fallo

Para señalizar un fallo en una actividad, se sigue un procedimiento ligeramente distinto al propuesto en la especificación de BPEL4WS, pero equivalente en resultados. En lugar de determinar en tiempo de ejecución (o *tiempo de verificación*, en este caso) cuál es el manejador de fallos adecuado, proponemos calcularlo *a priori* en el momento de representación del proceso BPEL4WS mediante CFM. Cada vez que sea necesario representar un punto de ocurrencia de fallos, se siguen los siguientes pasos:

1. Se determina el nombre del fallo y, si procede, el tipo de datos de la variable asociada.

2. Se determina cuál es el manejador de fallos más adecuado para dicho fallo, siguiendo las instrucciones de la especificación de BPEL4WS.

3. Si se encuentra un manejador de fallos adecuado, se modifica la variable de estado del ámbito que lo contenga, estableciendo el nombre de fallo y actividad principal del manejador adecuados. A partir de este momento las propias transiciones funcionales de manejo de fallos del ámbito toman el control. Si el fallo tiene una variable asociada, entonces se copia el contenido de dicha variable a la variable referenciada en la cláusula *catch*.

4. Si no se encuentra un manejador de fallos adecuado, se fuerza la cancelación del proceso, estableciendo su estado en *terminating*.

Supongamos que en el paso 2 se determina que el manejador de fallos adecuado para un fallo llamado f está en el *scope* s y que la actividad principal de dicho manejador es c. Supongamos también que la expresión $assign$ representa las asignaciones necesarias para copiar la variable del fallo a la variable de la cláusula *catch*. Si no hay ninguna variable asociada al fallo, simplemente se omite esta expresión. La expresión de acción que señaliza la ocurrencia de dicho fallo es:

$$assign \wedge (s.fault = f) \wedge (s.fault_handler = c) \tag{7.38}$$

Si, por el contrario, no se encontrase un manejador de fallos adecuado, se inicia la terminación del proceso con una expresión de acción como la siguiente. Se denota como p a la actividad CFM que representa al proceso:

$$(p.estado = terminating) \tag{7.39}$$

En toda actividad en que se pueda producir algún tipo de fallo durante su ejecución, se crea una transición funcional para representar la posible ocurrencia de dicho fallo. Es el caso de los fallos predefinidos para *procesos ejecutables* en la especificación de BPEL4WS y de las actividades `invoke` con operaciones que, según su declaración WSDL, puedan lanzar fallos. Estas transiciones se construirían de la siguiente forma.

$$\begin{array}{lll} dom_i(a, throw_f) & ::= & (a.state = running) \\ act_i(a, throw) & ::= & (a.state = cancelled) \wedge assign \wedge \\ & & \wedge (s.fault = f) \wedge (s.fault_handler = c) \end{array} \tag{7.40}$$

Para fallos no capturados:

$$\begin{array}{lll} dom_i(a, throw_f) & ::= & (a.state = running) \\ act_i(a, throw) & ::= & (a.state = cancelled) \wedge \\ & & \wedge (p.state = terminating) \end{array} \tag{7.41}$$

Ejemplo

En la figura 7.17 se muestra un proceso BPEL4WS de ejemplo, con varias actividades *scope*, manejadores de fallos y actividades *throw*. En la figura 7.18 se representa mediante CFM el *scope* llamado *scope0* en este ejemplo, así como su manejador para fallos *tns:fault1*.

7.2.21. Actividad *throw*

La implementación de la actividad `throw` es bastante sencilla. Puede ser representada con una única transición funcional que inicie y finalice la propia actividad. La acción de dicha transición funcional contendrá el código necesario para señalizar la ocurrencia del fallo, tal y como se ha planteado en el apartado 7.2.20. Si se utiliza la misma notación que en dicho apartado, esta transición funcional se definiría como:

$$\begin{array}{lll} dom_i(a, throw) & ::= & (a.state = not_started) \\ act_i(a, throw) & ::= & (a.state = completed) \wedge assign \wedge \\ & & \wedge (s.fault = f) \wedge (s.fault_handler = c) \end{array} \tag{7.42}$$

Si no hubiese ningún manejador de fallos adecuado para el fallo, la acción de esta transición funcional sería distinta:

$$act_i(a, throw) \quad ::= \quad (a.state = completed) \wedge (p.state = terminating) \tag{7.43}$$

Ejemplo

En la figura 7.19 se muestra la representación CFM de cada una de las actividades *throw* del ejemplo de la figura 7.17.

7.2.22. Modelado de vínculos (*links*)

Los vínculos (*links*) de BPEL4WS son un mecanismo de especificación del flujo de control de un proceso, complementario al mecanismo de estructuración jerárquica de actividades. Un vínculo está asociado a una actividad origen y una actividad destino. Contiene una variable *Booleana*, cuyo valor es establecido por la actividad origen, que controla la ejecución o no ejecución de la actividad destino.

Cada vínculo se representa en CFM mediante una entidad, instancia del tipo de entidad `BPEL_lnk`. Este tipo de entidad consta de dos campos de tipo *Booleano*. El primero de ellos indica si el vínculo ha sido evaluado o no. El otro, el resultado de la evaluación del mismo. El valor inicial de ambos atributos debe ser *falso*.

```
enttype BPEL_lnk {
  evaluated: boolean;
  evaluation: boolean;
}
```

En realidad, se podría haber modelado esta entidad con un campo de rango enumerado con tres valores: *no evaluado*, *evaluado con valor cierto* y *evaluado con valor falso*. Aunque reduce el espacio de estados total del proceso, no se ha optado por esta alternativa debido a que dificultaría la asignación del resultado de la evaluación, obligando a duplicar las transiciones funcionales de finalización de la actividad origen de cada vínculo.

La función XPath *bpws:getLinkStatus* se representa como una referencia al campo *evaluation* del vínculo que reciba como parámetro.

Actividad origen de un vínculo

Las transiciones funcionales de la actividad origen de cada vínculo deben ser modificadas, en su parte extrínseca, añadiendo predicados de acción que realicen lo siguiente:

- Las transiciones funcionales de finalización normal, y finalización de ejecución del manejador de fallos de un ámbito (véase el mecanismo de manejo de fallos en el apartado 7.2.20) de la actividad deben establecer valor *cierto* en el campo *evaluated* de la entidad que representa el vínculo, y el valor resultante de evaluar su condición de transición en el campo *evaluation*.

- La transición funcional que cancela la actividad (véase el mecanismo de cancelación en el apartado 7.2.18) debe establecer valor *cierto* en el campo *evaluated* de la entidad que representa el vínculo, y valor *falso* en el campo *evaluation*.

A continuación se muestra la acción extrínseca asociada a cada una de estas transiciones funcionales. La entidad del vínculo se denota como lnk. La condición de transición asociada a la origen del vínculo se denota como $cond$.

$$\begin{array}{lll} act_e(a, complete) & ::= & lnk.evaluated \wedge lnk.evaluation = cond \\ act_e(a, cancel) & ::= & lnk.evaluated \wedge \neg lnk.evaluation \\ act_e(a, catch_completion) & ::= & lnk.evaluated \wedge lnk.evaluation = cond \\ act_e(a, cancel_end) & ::= & lnk.evaluated \wedge \neg lnk.evaluation \end{array} \qquad (7.44)$$

La de la transición funcional *complete* es extensiva a las transiciones funcionales de finalización de cada actividad: si alguna actividad consta de varias transiciones funcionales de finalización, o de alguna cuyo nombre no sea *complete*, se aplica de la misma forma. La transición funcional *cancel* aplica a las actividades con ciclo de vida simple. Las transiciones funcionales *cancel_end* y *catch_completion* aplican al resto de actividades.

Nótese que, según se indica en la especificación de BPEL4WS, si ocurre un fallo en un ámbito, y este es adecuadamente capturado por uno de sus manejadores, que finaliza correctamente, se interpreta que el ámbito finaliza correctamente. En base a lo anterior, se considera que en esta situación, aunque la especificación no lo indique explícitamente, los vínculos de los cuales el propio ámbito sea origen se evalúan normalmente, como si no se hubiese producido ningún fallo.

Actividad destino de un vínculo

Las transiciones funcionales de la actividad destino de cada vínculo deben ser modificadas:

- Las transiciones funcionales de inicio deben contar, en su parte extrínseca, con una condición adicional. Esta condición establece que todos los vínculos de los cuales la actividad es destino deben haber sido evaluados, combinada mediante una *y–lógica* con la *condición de unión* asociada a la actividad.

- Si el atributo *suppressJoinFailure* que aplica a la actividad toma valor *falso*, se añade una transición funcional que lance un fallo *bpws:joinFailure*. Su dominio consta de la

misma condición que el de la transición funcional de inicio, con la diferencia de que la *condición de unión* se presenta negada. Su acción cancela la actividad y lanza el fallo (véase el mecanismo de gestión de fallos en el apartado 7.2.20).

- Si el atributo *suppressJoinFailure* que aplica a la actividad toma valor *cierto*, y la actividad es también origen de algún otro vínculo, se añade una transición funcional que implementa el mecanismo de *eliminación de caminos muertos* (*dead–path–elimination, DPE*). Su dominio consta de la misma condición que el de la transición funcional de inicio, con la diferencia de que la *condición de unión* se presenta negada. Su acción cancela la actividad y evalúa todos los vínculos de los cuales la actividad sea origen con valor *falso*.

Si $cond$ denota la *condición de unión* de la actividad, $lnki_0, lnki_1, \ldots, lnki_{n-1}$ todos los vínculos de los cuales la actividad es destino, y $dom'_e(a, begin)$ el dominio extrínseco de su transición funcional de inicio, excluyendo el mecanismo de gestión de vínculos, se construye el dominio extrínseco de su transición funcional de inicio la siguiente forma:

$$dom_e(a, begin) ::= dom'_e(a, begin) \wedge \bigwedge_{i=0}^{n-1} lnki_i.evaluated \wedge cond \qquad (7.45)$$

La definición anterior afecta a cualquier transición funcional que inicie una actividad, independientemente de que su nombre sea o no "*begin*".

La transición funcional que genera un fallo *bpws:joinFailure* se construye de forma similar. La expresión de acción $throw$ denota las acciones necesarias para generar el fallo, mediante el mecanismo de *señalización de fallos*. Los vínculos cuya origen es la propia actividad se denotan como $lnko_0, lnko_1, \ldots, lnko_{m-1}$

$$\begin{array}{lcl} dom(a, join_failure) & ::= & dom_i(a, begin) \wedge dom'_e(a, begin) \wedge \\ & & \wedge \bigwedge_{i=0}^{n-1} lnki_i.evaluated \wedge \neg cond \\ act(a, join_failure) & ::= & (a.state = cancelled) \wedge \bigwedge_{i=0}^{m-1} lnko_i.evaluated \wedge \\ & & \wedge \bigwedge_{i=0}^{m-1} \neg lnko_i.evaluation \wedge throw \end{array} \qquad (7.46)$$

La única diferencia de la transición funcional que implementa el mecanismo DPE con respecto a la anterior es el hecho de no generar ningún fallo. Esta transición funcional sólo se define en caso de que la actividad sea origen de algún vínculo. Su definición se construye de la siguiente forma:

$$\begin{array}{lcl} dom(a, dpe) & ::= & dom_i(a, begin) \wedge dom'_e(a, begin) \wedge \\ & & \wedge \bigwedge_{i=0}^{n-1} lnki_i.evaluated \wedge \neg cond \\ act(a, dpe) & ::= & (a.state = cancelled) \wedge \bigwedge_{i=0}^{m-1} lnko_i.evaluated \wedge \\ & & \wedge \bigwedge_{i=0}^{m-1} \neg lnko_i.evaluation \end{array} \qquad (7.47)$$

Ejemplo

En la figura 7.20 se propone un ejemplo con cuatro actividades, dentro de una actividad *flow* con interdependencias modeladas mediante vínculos.

7.2.23. Compensación

El mecanismo de *compensación* de BPEL4WS es, con diferencia, la funcionalidad más compleja que presenta el lenguaje. Su principal fuente de complejidad son los siguientes aspectos:

- El manejador de compensación de un ámbito, cuando se ejecuta, debe ver todas las variables con el mismo valor que estas tuviesen en el momento de finalizar la ejecución normal de dicho ámbito. Esto obliga a guardar una copia de las variables para cada uno de los ámbitos finalizados que pudiesen ser compensados en el futuro.

- El manejador de compensación por defecto de un ámbito debe forzar la ejecución de los manejadores de compensación de todos los ámbitos anidados inmediatamente en él, en orden inverso al de finalización de los mismos. Esto obliga a guardar el orden en que finalizan los distintos ámbitos del proceso.

A los aspectos anteriores, hay que añadir el hecho de que cabe la posibilidad de que haya ámbitos anidados en el interior de un bucle *while*. Esto contribuye a hacer más complejo el mecanismo de compensación, porque en este caso podría ser imposible conocer, en tiempo de diseño, el número máximo de iteraciones de un bucle y, por tanto, el número máximo de instancias de los ámbitos que se ejecuten dentro de dicho bucle.

Como consecuencia de lo anterior, cabe decir que no es posible representar por completo el mecanismo de compensación de BPEL4WS mediante el formalismo CFM.

En cualquier caso, sí es posible representar este mecanismo en un subconjunto de procesos BPEL4WS, aquellos que *no contengan ningún ámbito anidado dentro de un bucle*. Bajo esta premisa, el número máximo de instancias de un mismo ámbito que pueden ser creadas es *uno*. Por tanto, es posible, para cada uno de los ámbitos, declarar una copia de todos los atributos visibles a dicho ámbito. En las transiciones funcionales de finalización normal del ámbito se incluye, en su expresión de *acción*, el código necesario para realizar una asignación de los valores de los atributos a su copia correspondiente. El manejador de compensación del ámbito, incluyendo todas sus actividades internas, no accede a los atributos originales, sino a sus copias.

No cabe duda de que el hecho de duplicar atributos puede aumentar considerablemente la complejidad de la verificación. Una optimización obvia que reduce su crecimiento consiste en duplicar sólo aquellos atributos a los cuales el manejador de compensación acceda en modo de lectura, ignorando operaciones sobre atributos a los que sólo acceda en modo de escritura.

El problema de la compensación en orden inverso se resuelve creando, dentro de cada ámbito que internamente contenga otros ámbitos, una pila implementada mediante una *lista enlazada de ámbitos internos finalizados*. Cada vez que finaliza un ámbito, se apila. En el momento de realizar la compensación, el ámbito que esté en cabeza de la pila ejecuta su manejador de compensación y se desapila. Se continúa ejecutando estos manejadores sucesivamente hasta que la pila se vacíe.

Esta pila puede ser implementada añadiendo los siguientes tipos de entidades y entidades:

- Tipo de entidad *bpel_scopes__*: contiene un campo de rango enumerado, llamado *scope*, cuya enumeración de valores se compone de los nombres de todas las acti-

vidades CFM que representen ámbitos declarados en el proceso BPEL4WS, y el valor especial *null*.

- En cada *scope* que contenga internamente al menos un *scope* anidado, una entidad llamada *completion_stack__*, instancia del tipo de entidad anterior. El valor de esta entidad debe ser en todo momento el nombre de la actividad correspondiente a la cabeza de la pila. Al nombre de la entidad se concatena el nombre de la actividad CFM que represente el *scope*.

- En cada *scope*, una entidad llamada *completion_stack_next__*, instancia del tipo de entidad anterior. Su valor debe ser el nombre de la actividad correspondiente al ámbito, de entre los anidados directamente en el mismo ámbito superior, que haya finalizado inmediatamente antes. Al nombre de la entidad se concatena el nombre de la actividad CFM que represente cada uno de estos *scopes*.

Con estos atributos, la operación de *apilar* un ámbito recién finalizado consiste en asignar a su entidad *completion_stack_next__* el valor de *completion_stack__* de su ámbito inmediatamente superior, y asignar a esta última entidad el nombre del propio ámbito.

La operación de *desapilar* un ámbito consiste en asignar el valor de su entidad *completion_stack_next__* a la entidad *completion_stack__* de su ámbito inmediatamente superior.

A modo de resumen, aunque no es posible representar este mecanismo para cualquier proceso BPEL4WS, sí lo es para un amplio subconjunto de ellos. En cualquier caso, esta representación podría suponer un aumento considerable en la complejidad de la verificación.

Definición de las transiciones funcionales asociadas al manejo de compensación

En los siguientes apartados se define cómo se construyen las actividades y transiciones funcionales asociadas a manejadores de compensación. La notación empleada es la siguiente:

- s: el ámbito en el cual se define el manejador de compensación.

- $s_0, \ldots, s_{n-1}$: todos aquellos ámbitos cuyo ámbito inmediatamente superior es s.

- a_0: actividad principal del manejador de compensación.

- $s.ctop$: campo *scope* de la entidad *completion_stack__* del ámbito s.

- $s_i.cnext$: campo *scope* de la entidad *completion_stack_next__* del ámbito s_i.

Un ámbito puede definir un manejador de compensación de forma explícita, indicando una actividad, posiblemente estructurada, cuyo comportamiento realiza las acciones de compensación necesarias. Si no lo define, entonces cuenta con un manejador implícito, cuyo cometido es compensar todos los ámbitos inmediatamente inferiores, en orden inverso a su finalización.

Es posible también ejecutar el comportamiento del manejador implícito en algún punto concreto de la ejecución del manejador explícito, mediante una actividad *compensate* que no posea atributo *scope*.

Manejador explícito

En caso de que se defina explícitamente un manejador de fallos para el ámbito, la actividad principal del manejador de fallos, a_0, puede comenzar su ejecución cuando en el ámbito se produzca un cambio de estado de *completed* a *compensating*. Por otra parte, una transición funcional en el ámbito fuerza el cambio a estado *compensated* cuando finalice esta actividad.

$$\begin{array}{lcl} dom_e(a_0, begin) & ::= & (s.state = compensating) \\ dom(s, compensate_end) & ::= & (s.state = compensating) \wedge (a_0.state = completed) \\ act(s, compensate_end) & ::= & (s.state = compensated) \end{array} \tag{7.48}$$

La actividad a_0, y todas sus actividades internas, se transforman de forma recursiva, siguiendo el procedimiento normal, exceptuando el hecho de que se ignoran las escrituras en atributos a los que se acceda sólo en modo de escritura, y los accesos al resto de atributos se realizan a sus copias almacenadas a la finalización del ámbito. El mecanismo de copia se expone posteriormente.

Es posible que internamente aparezca una actividad *compensate* sin atributo *scope*. Si es el caso, es necesario también definir las entidades y transiciones funcionales descritos en el siguiente apartado.

Manejador implícito

En caso de que no se defina el manejador de compensación explícitamente, o que se defina pero este incluya una actividad *compensate* sin atributo *scope*, es necesario representar el comportamiento del manejador implícito. Para ello, y sólo en este caso, se definen las siguientes entidades y transiciones funcionales.

El mecanismo de compensación implícita se pone en marcha cuando el ámbito esté en estado *compensating* y el atributo *compensate* de su entidad del ciclo de vida tome valor *cierto*. Nótese que este último permanece con valor falso durante la ejecución del manejador explícito, salvo que desde este se invoque al manejador implícito.

Para gestionar la pila de ámbitos finalizados, se define una entidad *completion_stack__* para el ámbito s, y una entidad *completion_stack_next__* para cada uno de los ámbitos s_i.

Pos otra parte, se definen las transiciones funcionales necesarias para realizar la compensación, para cada uno de los ámbitos s_i, con $0 \leqslant i < n$:

$$act_e(s_i, complete) \quad ::= \quad (s_i.cnext = s.ctop) \wedge (s.ctop = s_i) \tag{7.49}$$

$$\begin{array}{lcl} dom(s_i, compensate_default) & ::= & (s.state = compensating) \wedge \\ & & \wedge s.compensate \wedge \\ & & \wedge (s_i.state = completed) \wedge \\ & & \wedge (s.ctop = s_i) \\ act(s_i, compensate_default) & ::= & (s_i.state = compensating) \end{array} \tag{7.50}$$

$$\begin{array}{lcl} dom(s_i, no_compensate) & ::= & (s.state = compensating) \wedge \\ & & \wedge s.compensate \wedge \\ & & \wedge (s_i.state \neq completed) \wedge \\ & & \wedge (s.ctop = s_i) \\ act(s_i, no_compensate) & ::= & (s.ctop = s_i.cnext) \end{array} \tag{7.51}$$

La transición funcional adicional, *no_compensate*, sirve para no bloquear la compensación de posteriores ámbitos almacenados en la pila cuando el ámbito a su cabeza no esté en un estado compensable.

Por otra parte, la transición funcional *compensate_end* de cada ámbito s_i, en el caso de que el ámbito s tenga manejador implícito, debe añadir en su acción la expresión adecuada para desapilar el ámbito: $s.ctop = s_i.cnext$.

En el propio ámbito a ser compensado, se define la siguiente transición funcional:

$$\begin{array}{lcl} dom(s, compensate_end) & ::= & (s.state = compensating) \wedge (s.ctop = null) \\ act(s, compensate_end) & ::= & (s.state = compensated) \end{array} \tag{7.52}$$

Duplicación de atributos

Tal y como se ha mencionado anteriormente, es necesario almacenar, en el instante de finalización de un ámbito, una copia de todos aquellos atributos a los que se acceda en modo de lectura dentro su manejador explícito de compensación. No es necesario realizar esta copia en aquellos ámbitos que no definan un manejador explícito.

Dada la complejidad que supondría en CFM duplicar sólo una parte de los atributos definidos por una entidad, se duplica la entidad completa si al menos uno de sus atributos debe ser duplicado.

Para duplicar una entidad, se define una nueva entidad, del mismo tipo, y con un nombre formado por la concatenación de la entidad original, la cadena "*_copy__*" y el nombre CFM de la actividad que representa el ámbito.

A la transición de finalización del ámbito se le añade, mediante una *y–lógica*, el código necesario para almacenar el valor de las entidades originales en las copias de las mismas. Si los atributos a duplicar son $e_0, e_1, \ldots, e_{n-1}$, y sus copias $e'_0, e'_1, \ldots, e'_{n-1}$, entonces esta expresión es la siguiente:

$$\bigwedge_{i=0}^{n-1} (e'_i = e_i) \tag{7.53}$$

7.2.24. Actividad *compensate*

La actividad *compensate* fuerza la ejecución del manejador de compensación de un ámbito. Si consta de un atributo *scope*, este indica el nombre del ámbito a ser compensado. Si no, la actividad debe estar en el manejador de compensación de un ámbito, y fuerza la ejecución de su manejador de compensación implícito.

Si el manejador de compensación no puede ser ejecutado, el comportamiento de esta actividad es similar al de la actividad *empty*. Esto ocurre cuando el ámbito objeto de la compensación no se encuentra en estado *completed* o cuando se invoca al manejador implícito sin estar este disponible. Un manejador implícito no está disponible si existe un manejador explícito en que se invoca explícitamente un manejador de compensación de algún ámbito inmediatamente anidado.

Se modela la actividad *compensate* con dos transiciones funcionales, ambas de inicio y alternativas. Una de ellas activa el manejador cuando el estado del ámbito objeto es el

correcto. La otra, representa el comportamiento, similar a *empty*, cuando no se puede ejecutar este manejador.

Cuando una actividad *compensate* (denotada como a) invoca explícitamente el manejador de compensación de un ámbito s', se modela con las siguientes transiciones funcionales:

$$\begin{aligned} dom_i(a, begin) &::= (a.state = not_started) \wedge \\ &\quad \wedge (s'.state = completed) \wedge \\ &\quad \wedge (s'.fault = none) \\ act_i(a, begin) &::= (a.state = running) \wedge \\ &\quad \wedge (s'.state = compensating) \end{aligned} \tag{7.54}$$

$$\begin{aligned} dom_i(a, not_compensate) &::= (a.state = not_started) \wedge \\ &\quad \wedge ((s'.state \neq completed) \vee \\ &\quad \vee (s'.fault \neq none)) \\ act_i(a, not_compensate) &::= (a.state = completed) \end{aligned} \tag{7.55}$$

$$\begin{aligned} dom_i(a, complete) &::= (a.state = running) \wedge \\ &\quad \wedge (s'.state = compensated) \\ act_i(a, complete) &::= (a.state = completed) \end{aligned} \tag{7.56}$$

Si el manejador se invoca desde un manejador de compensación de un ámbito s inmediatamente superior de s', entonces es necesario deshabilitar el manejador de compensación implícito de s, añadiendo a la acción de ambas transiciones funcionales, mediante *y–lógica*, la expresión $s.ctop = null$. De esta forma, se vacía la pila de ámbitos finalizados de s. Si se ejecutase el manejador implícito, este no tendría efecto alguno.

Cuando la actividad *compensate* (denotada como a) invoca explícitamente su manejador de compensación implícito, se representaría como:

$$\begin{aligned} dom_i(a, compensate) &::= (a.state = not_started) \wedge \\ &\quad \wedge (s'.state = completed) \wedge \\ &\quad \wedge (s'.fault = none) \\ &\quad \wedge (s'.ctop \neq null) \\ act_i(a, compensate) &::= (a.state = completed) \wedge \\ &\quad \wedge (s'.state = compensating) \wedge \\ &\quad \wedge s'.compensate \end{aligned} \tag{7.57}$$

$$\begin{aligned} dom_i(a, not_compensate) &::= (a.state = not_started) \wedge \\ &\quad \wedge ((s'.state \neq completed) \vee \\ &\quad \vee (s'.fault \neq none) \\ &\quad \vee (s'.ctop = null)) \\ act_i(a, not_compensate) &::= (a.state = completed) \end{aligned} \tag{7.58}$$

$$\begin{aligned} dom_i(a, complete) &::= (a.state = running) \wedge \\ &\quad \wedge (s'.state = compensated) \\ act_i(a, complete) &::= (a.state = completed) \end{aligned} \tag{7.59}$$

7.2.25. Manejo de eventos

Tanto el proceso como cada actividad *scope* pueden tener manejadores de eventos asociados. BPEL4WS define dos tipos de eventos: mensajes entrantes y alarmas. Los manejadores de eventos se instalan cuando comienza la ejecución del proceso o *scope* al cual están asociados, y finalizan cuando este finaliza su ejecución normal.

Los manejadores de eventos se ejecutan de forma concurrente con las actividades normales de su proceso o *scope*, así como con otros manejadores de eventos. Por otra parte, un manejador de eventos asociado a un mensaje puede ser activado más de una vez durante la ejecución de su proceso o *scope*. Es más, es posible la ejecución concurrente de varias instancias del mismo manejador de eventos.

Desde el punto de vista del mecanismo de manejo de fallos, los manejadores de eventos son equivalentes a cualquier otra actividad que se ejecute en el flujo normal del proceso o *scope*.

Representación de un manejador de eventos

Cada manejador de eventos se representa mediante una actividad especial en el modelo CFM. El inicio en la ejecución de esta actividad permite que comience su ejecución la actividad interna del manejador. El fin de la ejecución de dicha actividad interna provoca el fin de la ejecución del manejador.

Un mismo manejador puede ser instanciado varias veces. Para permitir esto, es necesario, una vez finalizada la ejecución de una instancia del mismo, restablecer el estado inicial en todas sus actividades internas. De esta forma, el manejador de fallos puede volver a ser ejecutado en el futuro.

A continuación se especifica cómo se define esta actividad. La notación empleada para ello es la siguiente:

- a: la actividad que representa el manejador de eventos.
- a_0: la actividad de primer nivel contenida en el manejador de eventos.
- $a_1, \ldots, a_{n-1}$: las actividades contenidas en el manejador de eventos, en cualquier nivel de profundidad, excluida a_0.
- $lnk_0, \ldots, lnk_{m-1}$: los vínculos definidos en actividades *flow* en el interior, en cualquier nivel de profundidad, del manejador de eventos.

Las transiciones funcionales de inicio y finalización que definen esta actividad se construyen de la siguiente forma:

$$\begin{array}{lll} dom(a, begin) & ::= & (a.state = not_started) \wedge (s.state = running) \\ act(a, begin) & ::= & (a.state = running) \wedge reset \\ dom(a, complete) & ::= & (a.state = running) \wedge (a_0.state = completed) \\ act(a, complete) & ::= & (a.state = not_started) \\ reset & ::= & \bigwedge_{i=0}^{n-1} (a_k.state = not_started) \wedge \bigwedge_{i=0}^{m-1} (\neg lnk_i.evaluated) \end{array} \quad (7.60)$$

Por otra parte, esta actividad consta de una transición funcional de cancelación convencional, tal y como se describe en el apartado 7.2.18.

Si se tratase de un manejador de eventos de tipo alarma, entonces sólo se puede instanciar una vez. En este caso, se simplifica considerablemente la definición:

$$\begin{array}{lll} dom(a, begin) & ::= & (a.state = not_started) \wedge (s.state = running) \\ act(a, begin) & ::= & (a.state = running) \\ dom(a, complete) & ::= & (a.state = running) \wedge (a_0.state = completed) \\ act(a, complete) & ::= & (a.state = completed) \end{array} \quad (7.61)$$

A primera vista, puede resultar curioso el hecho de que no haya ninguna pre–condición para el inicio de la actividad, más allá de la condición de que la actividad *scope* o proceso deben estar en ejecución y la propia actividad del manejador debe estar parada. Sin embargo, esto es correcto, dado que, durante la verificación, podría ocurrir que nunca llegase a activarse el manejador antes de finalizar el *scope* o, por el contrario, que se activase.

Desde el punto de vista de la verificación, el hecho de que se pueda activar el manejador de recepción de mensaje tantas veces como se quiera, en paralelo con la ejecución normal del *scope*, da lugar a que puedan ocurrir ciclos de duración infinita, debidos a la llegada de un número infinito de mensaje de entrada durante la ejecución del *scope*. Para solucionar este problema, puede ser necesario imponer una condición de *fairness* o una etiqueta *progress*, para que el verificador excluya la posibilidad de llegada de infinitos mensajes.

El dominio extrínseco de la transición funcional de inicio de la actividad interna de primer nivel del manejador de eventos se define como:

$$dom_e(a_0, begin) \quad ::= \quad (a.state = running) \quad (7.62)$$

Instancias concurrentes del mismo manejador

Nótese que, aplicando esta metodología de transformación, se permite que un mismo manejador de eventos se instancie y ejecute múltiples veces. Sin embargo, no es posible la ejecución concurrente de dos instancias del mismo manejador. En el apartado 7.4.2 se discute esta limitación.

Como solución parcial a este problema, se propone ofrecer la posibilidad al diseñador de acotar el número máximo de ejecuciones concurrentes. Dada esta cota, se replican en consonancia las actividades CFM asociadas al manejador y a todas sus actividades internas, incluyendo sus entidades de ciclo de vida.

Normalmente, es suficiente con dos instancias del manejador para verificar una gran mayoría de los posibles caminos. Por tanto, esta técnica puede resultar suficiente en una mayoría de los casos para detectar errores asociados con la ejecución concurrente de los manejadores.

Manejador de eventos con mensajes alternativos

Los manejadores de eventos de recepción de mensajes permiten también la especificación de mensajes de entrada alternativos. Si se especifican mensajes alternativos, se crea una transición funcional de inicio distinta para cada uno de estos mensajes. Todas ellas se

definen con el mismo dominio. Cada una realiza, en su predicado de acción, la asignación correspondiente al mensaje de entrada que representa.

Si se representa mediante un índice j el número de mensaje alternativo, y mediante $assign_j$ el predicado de acción que realiza la asignación del contenido de dicho mensaje a las variables del proceso, se representa cada una de estas transiciones funcionales de inicio como:

$$\begin{array}{lll} dom(a, begin_j) & ::= & (a.state = not_started) \wedge (s.state = running) \\ act(a, begin_j) & ::= & (a.state = running) \wedge reset \wedge assign_j \end{array} \quad (7.63)$$

Ejemplo

En la figura 7.21 se presenta un ejemplo de representación de un manejador de eventos de una actividad *scope*. Por sencillez, en este ejemplo no se proponen mensajes alternativos de entrada.

7.3. Especificaciones

El diseñador de un proceso debe ser capaz no sólo de definir el proceso en sí mismo, sino también los requisitos que componen su especificación. Estos requisitos deben poder ser expresados en un lenguaje de expresiones al cual dicho diseñador esté habituado, y convertidos, junto a la definición del proceso, a requisitos en el ámbito del formalismo CFM. En este apartado se propone una sintaxis para la definición de requisitos, y una metodología de transformación de los mismos a requisitos CFM.

7.3.1. Definición de requisitos

Se propone añadir un elemento de extensión adicional a las definiciones de procesos BPEL4WS, definido dentro de un espacio de nombres propio, llamado *property*. Este elemento se puede incluir como contenido del elemento *process* de BPEL4WS.

Este elemento tiene un atributo obligatorio, llamado *type*, para especificar el tipo de requisito. Adicionalmente, aquellos requisitos que hagan referencia a una actividad concreta, pueden especificar su nombre mediante el atributo opcional *activity*. Los requisitos suelen tener un predicado asociado, que se especifica mediante el atributo *expression*. Opcionalmente, se puede asignar un nombre al requisito mediante el atributo *name*.

El contenido del predicado del atributo *expression* es una expresión con sintaxis XPath. El principal motivo para seleccionar este lenguaje es que se trata del lenguaje de expresiones por defecto en BPEL4WS, y toda aplicación que procese BPEL4WS debe dar soporte a este lenguaje.

Dado que el nivel de abstracción es distinto en BPEL4WS con respecto al formalismo CFM, no todos los tipos de requisitos de este último, definidos en el apartado 6.2.3, son aplicables a BPEL4WS. Concretamente, se definen los siguientes tipos de requisitos para BPEL4WS (entre paréntesis se especifica el valor que toma el atributo *tipo* en cada caso):

- Invariante (*invariant*): se corresponde con el mismo concepto presentado en el formalismo CFM. La única diferencia está en el lenguaje de expresiones, y en los símbolos referenciables desde dichas expresiones.

- Objetivo (*goal*): se corresponde con el mismo concepto presentado en el formalismo CFM. Al igual que las invariantes, se diferencian en el lenguaje de expresiones.

- Pre–requisito de actividad (*pre-req*)): define un predicado *Booleano* que debe ser evaluado con valor cierto inmediatamente antes del inicio de la ejecución de la actividad. Se corresponde, en nivel CFM, con un pre–requisito sobre las transiciones funcionales de inicio de la actividad CFM resultante de la transformación de la actividad BPEL4WS.

- Post–requisito de actividad (*post-req*): define un predicado *Booleano* que debe ser evaluado con valor cierto inmediatamente después de de la *finalización normal* de ejecución de la actividad. No aplica en caso de finalización debida a errores o cancelación. Se corresponde, en nivel CFM, con un post–requisito sobre las transiciones funcionales de finalización normal de la actividad CFM resultante de la transformación de la actividad BPEL4WS.

- Ejecutabilidad de actividad (*exe*): impone que la actividad dada debe ser ejecutable, esto es, debe ser ejecutada en al menos una posible ejecución del proceso. Se corresponde en el nivel CFM con un requisito de ejecutabilidad sobre las transiciones funcionales de inicio de de la actividad CFM resultante de la transformación de la actividad BPEL4WS.

Por otra parte, desde el punto de vista del análisis del requisito general RG–1, el usuario debe poder marcar aquellas actividades que suponen un progreso del proceso para que. Si se detecta un ciclo de duración infinita en que no se ejecute ninguna actividad con marca de *progreso*, se notificará como un error. Se establece esta marca introduciendo el elemento *verbus:progress* (vacío y sin atributos) en el interior del elemento XML correspondiente a la actividad BPEL4WS. En la conversión a CFM se marcan las transiciones funcionales de inicio de esta actividad mediante el atributo *progress*.

7.3.2. Funciones añadidas al lenguaje de expresiones

Con el objetivo de enriquecer la expresividad de los requisitos, y aprovechando el mecanismo de extensibilidad de funciones del lenguaje XPath, se añade las siguiente función:

- Función *string verbus:activityState(string)*: devuelve una cadena de texto que representa el estado, en su ciclo de vida, de la actividad CFM correspondiente a la actividad BPEL4WS cuyo nombre se especifique como parámetro. Si se especifica el nombre del proceso BPEL4WS, devuelve el estado de la actividad CFM correspondiente al proceso.

- Funciones de lógica temporal: los operadores de lógica temporal se definen en XPath como funciones extendidas.

  - *boolean verbus:tla(boolean)*
  - *boolean verbus:tle(boolean)*
  - *boolean verbus:tlx(boolean)*

- *boolean verbus:tlf(boolean)*
- *boolean verbus:tlg(boolean)*
- *boolean verbus:tlu(boolean, boolean)*
- *boolean verbus:tlr(boolean, boolean)*

7.3.3. Requisitos específicos de esta metodología de transformación

No se plantea ningún requisito específico de la metodología de transformación de procesos BPEL4WS presentada en este capítulo. Por tanto, resultan aplicables los requisitos generales y los específicos que defina el diseñador del proceso a verificar.

7.3.4. Ejemplos

En la figura 7.22, al final de este capítulo, se muestra, a modo de ejemplo, la representación de varios requisitos de distintos tipos, así como su transformación a requisitos en el modelo CFM.

7.4. Limitaciones

En la exposición de la metodología de representación de BPEL4WS mediante el formalismo CFM, se han reseñado algunas limitaciones de CFM que impiden que sea capaz de representar de manera totalmente automática cualquier proceso BPEL4WS. En este apartado se recopilan y analizan, justificando hasta qué punto puede cada una de ellas limitar la aplicabilidad práctica de CFM.

7.4.1. Limitaciones en la representación de variables

La semántica presentada en este capítulo no es capaz de representar variables cuyo tipo de datos tenga un rango infinito. Este es el caso, por ejemplo, de datos de tipo textual. Estos casos se pueden dar con datos XML de tipos como, por ejemplo, los siguientes:

- Cadenas de texto que admiten un número ilimitado de caracteres. Por ejemplo, el tipo *string* y todos sus tipos derivados.
- Números enteros que admiten un número ilimitado de dígitos, así como números decimales con precisión ilimitada. Por ejemplo, los tipos *integer*, *decimal* y todos sus derivados.
- Tipos de datos complejos que, internamente, contengan recursivamente instancias de ellos mismos.

En estos casos, es necesario realizar una *abstracción* del rango de valores del tipo de datos, esto es, agrupar en subconjuntos el conjunto total de valores posibles, y codificar a qué subconjunto pertenece el valor de cada variable, en vez de codificar su valor concreto. Esta técnica no sólo es aplicable a casos en que el rango de valores del tipo de datos sea infinito, sino también a casos en que sea finito pero excesivamente elevado. Se propone un ejemplo de esta técnica en el apartado 4.5.1.

En determinados procesos, algunas de las variables de rango infinito de valores no afectan a la ejecución del proceso en sí mismo. En estos casos, es suficiente con transformar dichas variables a tipo *abstract* del formalismo CFM. En otras ocasiones, sin embargo, estas variables pueden influir en el desarrollo del proceso y no pueden, por tanto, ser totalmente abstraídas.

En el caso de datos de tipo textual, se puede realizar abstracción de forma automática, convirtiendo el tipo de datos a uno de rango enumerado. Este rango se construye de la siguiente forma:

- Un valor literal por cada una de las cadenas literales con las cuales se compare o realice asignación a las instancias del tipo de datos.
- Mezcla con los valores de tipos de datos de otras variables, cuando se realice una asignación o comparación entre dos variables textuales en este caso.
- Un valor que represente el caso en que el valor sea distinto a todos los anteriores.

La representación de tipos de datos numéricos también sufre el mismo problema. Por una parte, los tipos de datos numéricos de XML tienen un rango infinito de valores. Por otra, aun en procesos en que se conozca una cota al rango de variables numéricas, estas variables pueden elevar exponencialmente el espacio de estados del proceso, con el consiguiente aumento en los recursos necesarios para llevar a cabo la verificación.

Los tipos de datos numéricos pueden ser abstraídos de forma automática de la misma forma que los de tipo textual, siempre y cuando no se realicen operaciones aritméticas con ellos. Si se realizan operaciones aritméticas, se puede acotar su rango de forma automática. Sin embargo, esta última solución no siempre resulta adecuada, y en ocasiones es necesaria la intervención previa de una persona para realizar la abstracción que crea conveniente de forma manual. Por ejemplo, en procesos en que haya variables que represente una cantidad de dinero es necesario, típicamente, realizar una abstracción de los valores posibles de dichas variables. En el caso de estudio propuesto en el capítulo 8 se propone una situación en que resulta necesario hacer esto.

Los tipos de datos en que exista recursión requieren siempre intervención manual de una persona que realice abstracción de sus valores.

Conclusiones

Algunos tipos de datos deben ser abstraídos para poder ser representados mediante el formalismo CFM. Algunas de estas abstracciones pueden ser realizadas automáticamente en el proceso de transformación a CFM. Sin embargo, en algunas circunstancias, como es el caso de tipos de datos numéricos de rango excesivamente grande, con los cuales se realizan operaciones, o de tipos de datos definidos de forma recursiva, es necesaria la intervención de una persona para realizar de forma manual estas abstracciones.

7.4.2. Limitaciones en la representación de manejadores de eventos

El sistema de manejo de eventos de BPEL4WS permite la ejecución de varias instancias concurrentes del mismo manejador. La limitación del formalismo CFM relativa a la creación

dinámica de entidades, sin embargo, no permite modelar este comportamiento de forma razonable. Nótese que esta limitación no afecta a la ejecución concurrente de instancias de manejadores distintos.

La ejecución de dos instancias del mismo manejador de eventos de forma concurrente ocurre cuando se reciben dos mensajes asociados a la misma instancia del proceso, *socio*, *tipo de puerto* y *operación*, si el manejador del primer mensaje no ha finalizado antes de la llegada del segundo.

Si el tiempo de ejecución asociado a uno de estos manejadores no es excesivamente grande, se puede asumir que siempre dará tiempo a finalizar el manejo de un mensaje antes de la llegada del siguiente. Por otra parte, el hecho de que se ejecuten en secuencia dos instancias del manejador que deberían ejecutarse de forma concurrente no cambia, en la mayoría de los casos, los resultados visibles desde el punto de vista de la verificación del proceso.

La solución propuesta permite asumir un número acotado de manejadores concurrentes, pero no un número arbitrariamente alto. Esto puede provocar que sean indetectables los errores debidos a la ejecución concurrente de un número de instancias del manejador superior a la cota seleccionada. A pesar de ello, salvo en situaciones extrañas, es suficiente con dos manejadores concurrentes para detectar este tipo de errores.

Conclusiones

Se puede concluir que, aunque esta limitación puede provocar que algunos errores no sean detectables durante la verificación, resulta razonable asumir que esto sólo ocurre en casos extraordinarios.

7.4.3. Limitaciones en el sistema de gestión de compensación

Como se ha visto, no es posible representar adecuadamente el mecanismo de manejo de compensación de ámbitos anidados dentro de bucles *while*.

Una posible solución parcial consiste en añadir un contador al bucle, duplicar el código asociado al ámbito y a sus manejadores de compensación (no es necesario duplicar el código de sus actividades internas), y asociar la ejecución de cada una de las copias del ámbito a un valor del contador. Por supuesto, es necesario asumir una cota para el número de iteraciones del bucle, que podría configurar manualmente el usuario. Si esta cota es correcta, entonces la verificación es completa. Si no lo fuese, no serían verificados aquellos caminos en que el bucle sobrepasase la cota.

Conclusiones

La técnica presentada en este apartado consigue ampliar el rango de procesos BPEL4WS verificables, pero a costa de aumentar la complejidad de las verificaciones considerablemente, en especial en el caso de que haya ámbitos en el interior de varios bucles anidados. Por otra parte, la verificación no es completa en aquellos bucles en que no se pueda establecer una cota correcta.

7.4.4. Limitaciones en la representación de *scopes* serializables

Para representar *scopes* serializables, es necesario implementar un mecanismo de cerrojos mediante CFM. Dado que las transiciones funcionales se ejecutan de forma atómica, es posible implementar estos cerrojos, y las operaciones sobre los mismos, mediante atributos CFM. Sin embargo, dada la complejidad asociada a la implementación, se plantea como línea futura de trabajo.

7.5. Conclusiones

En este capítulo se ha expuesto una posible metodología de representación mediante el formalismo CFM de procesos BPEL4WS, una sintaxis de representación de especificaciones de requisitos de procesos BPEL4WS mediante XPath y una metodología de transformación de dichos requisitos a requisitos expresados mediante el formalismo CFM. Con ello, es posible, con algunas limitaciones, transformar de forma automática definiciones de procesos y especificaciones BPEL4WS a procesos y especificaciones CFM. Desde este último formalismo, a su vez, se pueden realizar verificaciones tal y como se expone en el capítulo 6.

La metodología de transformación presentada en este capítulo permite transformar correctamente todas las actividades de BPEL4WS, el sistema de vínculos (*links*) e incluso los mecanismos de manejo de fallos y manejo de eventos. Permite, con limitaciones, representar el mecanismo de manejo de compensación.

La metodología no es aplicable a cualquier proceso BPEL4WS, sino que está sujeta a algunas restricciones:

- No es posible representar mediante CFM tipos de datos XML Schema con un rango infinito de valores. Es el caso, por ejemplo, del tipo primitivo *xsd:string*, o de definiciones recursivas de elementos que incluyen, en su contenido, instancias del propio elemento. Estas limitaciones se solucionan mediante técnicas de abstracción, que en algunas ocasiones son aplicadas de forma automática, mientras que en otras requieren la intervención de una persona. Se ha discutido en mayor detalle esta limitación en el apartado 7.4.1.

- No es posible modelar la ocurrencia de un número arbitrario de instancias concurrentes de un mismo manejador de eventos asociado a recepción de un mensaje. Sin embargo, se puede modelar, a costa de duplicación de actividades, un número acotado de ocurrencias concurrentes. Como se ha discutido en el apartado 7.4.2, esta limitación no afecta a la verificación de requisitos en una gran mayoría de los casos.

- No es posible modelar el mecanismo de gestión de compensación de *scopes* definidos en el interior de una actividad *while*, salvo que se conozca, en tiempo de diseño, una cota al número de iteraciones de dicho bucle. En cualquier caso, aunque sea posible conocer esta cota, el excesivo número de variables que deben ser introducidas puede incrementar el espacio de estados del sistema drásticamente, limitando de esta forma las posibilidades de verificación. Por otra parte, este incremento en el espacio de estados afecta también a situaciones con ámbitos fuera de bucles *while*. Se profundiza en ello en el apartado 7.4.3.

- Esta metodología no especifica cómo representar *scopes* marcados con la propiedad *serializable*. Aunque es factible implementar mecanismos de cerrojos en el formalismo CFM, la representación de este tipo de *scopes* puede ser compleja. Se profundiza en ello en el apartado 7.4.4.

Sin embargo, se han propuesto mecanismos para reducir el impacto de estos casos, gracias a los cuales se puede, con una interacción manual reducida, realizar las verificaciones de estos procesos. Esta interacción puede consistir en abstraer los valores de determinados tipos de datos o proporcionar una cota al número de instancias concurrentes de un manejador de eventos asociado a mensajes. Aún en el caso de que no se pudiese proporcionar esta cota, se podrían asumir figuras razonables para la misma, que permitiesen realizar la verificación abarcando un número elevado, aunque no exhaustivo, de caminos. En la gran mayoría de los casos esto supone una restricción razonable.

Por tanto, a pesar de las limitaciones anteriores, se puede concluir que la metodología propuesta resulta aplicable a una amplia mayoría de procesos BPEL4WS, normalmente de forma automática, aunque en algunos casos con intervención manual de una persona.

Por otra parte, cabe destacar el hecho de que la complejidad del lenguaje BPEL4WS, desde el punto de vista de control de flujo, es superior a la de la mayoría de los lenguajes de definición de procesos de negocio utilizados en herramientas comerciales. La limitación del formalismo CFM en cuanto a la instanciación múltiple afecta también, a la vista de los resultados presentados en el capítulo 5, a estos lenguajes. Por tanto, el formalismo CFM podrá, con mucha probabilidad, representar completamente estos lenguajes.

7.6. Código de los ejemplos

En las páginas siguientes se muestran distintos ejemplos de transformación de procesos BPEL4WS al modelo CFM, aplicando la metodología propuesta en este capítulo. Cada uno de los ejemplos se centra en la transformación de un aspecto concreto del lenguaje BPEL4WS.

```
<xsd:complexType name="OliveOilRequestT">            <!-- fichero WSDL -->
  <xsd:sequence>
    <xsd:element name="quantity" type="xsd:integer" />
    <xsd:element name="maxPrice" type="xsd:integer" />
    <xsd:element name="deadline" type="xsd:integer" />
    <xsd:element name="customerId" type="xsd:string" />
  </xsd:sequence>
</xsd:complexType>
<xsd:simpleType name="OilTypeT">
  <xsd:restriction base="xsd:string">
    <xsd:enumeration value="olive"/>
    <xsd:enumeration value="soja"/>
    <xsd:enumeration value="sunflower"/>
  </xsd:restriction>
</xsd:simpleType>
<message name="OliveOilRequest">
  <part name="request" type="tns:OliveOilRequestT"/>
</message>
<message name="OliveOilResponse">
  <part name="request" type="tns:OliveOilRequestT"/>
  <part name="ordered" type="xsd:boolean"/>
  <part name="price" type="xsd:integer"/>
  <part name="totalPrice" type="xsd:integer"/>
  <part name="waitingDays" type="xsd:integer"/>
</message>
<message name="OilOrder">
  <part name="quantity" type="xsd:integer" />
  <part name="customerId" type="xsd:string" />
  <part name="oilType" type="tns:OilTypeT"/>
</message>
<variables>                                          <!-- fichero BPEL4WS -->
  <variable name="request" messageType="tns:OliveOilRequest" />
  <variable name="response" messageType="tns:OliveOilResponse" />
  <variable name="oOrder" messageType="tns:OilOrder" />
</variables>
```

```
enttype OliveOilRequest {
    request__quantity: integer;
    request__maxPrice: integer;
    request__deadline: integer;
    request__customerId: enum (_abstract__none, c102000);
}
enttype OliveOilResponse {
    totalPrice: integer;
    ordered: boolean;
    waitingDays: integer;
    price: integer;
    request__quantity: integer;
    request__maxPrice: integer;
    request__deadline: integer;
    request__customerId: enum (_abstract__none, c102000);
}
enttype OilOrder {
    oilType: enum (olive, soja, sunflower);
    quantity: integer;
    customerId: enum (_abstract__none, c102000);
}
entity request: OliveOilRequest;
entity response: OliveOilResponse;
entity oOrder: OilOrder;
```

Figura 7.5: Ejemplo de transformación de variables BPEL4WS.

```
<assign name="prepare_neg_response">
  <copy>
    <from variable="request" part="request" />
    <to variable="response" part="request" />
  </copy>
  <copy>
    <from><ordered>false</ordered></from>
    <to variable="response" part="ordered" />
  </copy>
</assign>
```

```
activity prepare_neg_response_act_2 {
    transition assign {
        domain: {(unnamed_act_1_lc__.state=running
                 & prepare_neg_response_act_2_lc__.state=not_started)}
        action: {(prepare_neg_response_act_2_lc__.state=completed
                 & (response.request__quantity=request.request__quantity
                   & response.request__maxPrice=request.request__maxPrice
                   & response.request__deadline=request.request__deadline
                   & response.request__customerId=request.request__customerId)
                 & !response.ordered)}
    }
}
```

Figura 7.6: Ejemplo de transformación de una actividad *assign*.

```
<sequence name="initial">
  <receive name="receive" ... />
  <pick name="my_pick">...</pick>
</sequence>
```

```
activity initial_act_1 {
    transition begin {
        domain: {initial_act_1_lc__.state=not_started}
        action: {initial_act_1_lc__.state=running}
    }
    transition complete {
        domain: {(initial_act_1_lc__.state=running
                 & my_pick_act_3_lc__.state=completed)}
        action: {initial_act_1_lc__.state=completed}
    }
}
activity receive_act_2 {
    transition begin {
        domain: {(initial_act_1_lc__.state=running
                  & receive_act_2_lc__.state=not_started)}
        action: {...}
    }
    (...)
}
activity my_pick_act_3 {
    transition begin {
        domain: {(receive_act_2_lc__.state=completed
                  & my_pick_act_3_lc__.state=not_started)}
        action: {...}
    }
    (...)
}
```

Figura 7.7: Ejemplo de transformación de una actividad *sequence*.

```
<flow name="initial">
  <receive name="receive" ... />
  <pick name="my_pick">...</pick>
</flow>
```

```
activity initial_act_1 {
    transition begin {
        domain: {initial_act_1_lc__.state=not_started}
        action: {initial_act_1_lc__.state=running}
    }
    transition complete {
        domain: {(initial_act_1_lc__.state=running
                 & initial_act_1_lc__.state=completed
                 & my_pick_act_3_lc__.state=completed)}
        action: {initial_act_1_lc__.state=completed}
    }
}
activity receive_act_2 {
    transition begin {
        domain: {(initial_act_1_lc__.state=running
                  & receive_act_2_lc__.state=not_started)}
        action: {...}
    }
    (...)
}
activity my_pick_act_3 {
    transition begin {
        domain: {(initial_act_1_lc__.state=running
                  & my_pick_act_3_lc__.state=not_started)}
        action: {...}
    }
    (...)
}
```

Figura 7.8: Ejemplo de transformación de una actividad *flow*.

```
<switch name="select">
  <case condition="bpws:getVariableData('response', 'price') &lt;=
                   bpws:getVariableData('request', 'request', 'maxPrice')
                   or
                   bpws:getVariableData('response', 'waitingDays') =
                   bpws:getVariableData('request', 'request', 'deadline')" >
    <empty name="case1" />
  </case>
  <otherwise>
    <empty name="case2" />
  </otherwise>
</switch>
```

```
enttype select_act_16_switch_sel__ {
   selected: enum (none, case_0, otherwise);
}
entity select_act_16_switch_sel__: select_act_16_switch_sel__;
activity select_act_16 {
    transition begin {
        domain: {(response_act_15_lc__.state=completed
                  & select_act_16_lc__.state=not_started)}
        action: {(select_act_16_lc__.state=running
                  & select_act_16_switch_sel__.selected=none)}
    }
    transition begin_case_0 {
        domain: {(select_act_16_lc__.state=running
                  & select_act_16_switch_sel__.selected=none
                  & ((response.price <= request.request__maxPrice)
                     | response.waitingDays=request.request__deadline))}
        action: {select_act_16_switch_sel__.selected=case_0}
    }
    transition complete_case_0 {
        domain: {(select_act_16_lc__.state=running
                  & case1_act_17_lc__.state=completed
                  & select_act_16_switch_sel__.selected=case_0)}
        action: {(select_act_16_lc__.state=completed
                  & select_act_16_switch_sel__.selected=none)}
    }
    transition begin_otherwise {
        domain: {(select_act_16_lc__.state=running
                  & select_act_16_switch_sel__.selected=none
                  & !((response.price <= request.request__maxPrice)
                      | response.waitingDays=request.request__deadline))}
        action: {select_act_16_switch_sel__.selected=otherwise}
    }
    transition complete_otherwise {
        domain: {(select_act_16_lc__.state=running
                  & case2_act_18_lc__.state=completed
                  & select_act_16_switch_sel__.selected=otherwise)}
        action: {(select_act_16_lc__.state=completed
                  & select_act_16_switch_sel__.selected=none)}
    }
}
activity case1_act_17 {
    transition empty {
        domain: {(select_act_16_switch_sel__.selected=case_0
                  & case1_act_17_lc__.state=not_started)} (...)
} (...)
```

Figura 7.9: Ejemplo de transformación de una actividad *switch*.

```
<!-- while the price is too high... -->
<while name="wait"
       condition="bpws:getVariableData('response', 'waitingDays')
                  &lt; bpws:getVariableData('request', 'request', 'deadline')
                  and bpws:getVariableData('oPrice', 'price') >
                  bpws:getVariableData('request', 'request', 'maxPrice')">
  <sequence name="loop">
      <wait name="wait_a_day" for="1D" />
      <assign name="incdays">...</assign>
      <invoke name="askPrice2" ... />
  </sequence>
</while>
```

```
activity wait_act_8 {
    transition begin {
        domain: {(askPrice1_act_7_lc__.state=completed
                  & wait_act_8_lc__.state=not_started)}
        action: {(wait_act_8_lc__.state=running
                  & wait_a_day_act_10_lc__.state=not_started
                  & incdays_act_11_lc__.state=not_started
                  & askPrice2_act_12_lc__.state=not_started
                  & loop_act_9_lc__.state=not_started)}
    }
    transition continue {
        domain: {(((response.waitingDays < request.request__deadline)
                   & (oPrice.price > request.request__maxPrice))
                 & wait_act_8_lc__.state=running
                 & loop_act_9_lc__.state=completed)}
        action: {(wait_a_day_act_10_lc__.state=not_started
                  & incdays_act_11_lc__.state=not_started
                  & askPrice2_act_12_lc__.state=not_started
                  & loop_act_9_lc__.state=not_started)}
    }
    transition complete {
        domain: {((loop_act_9_lc__.state=completed
                   | loop_act_9_lc__.state=not_started)
                  & !((response.waitingDays < request.request__deadline)
                  & (oPrice.price > request.request__maxPrice))
                  & wait_act_8_lc__.state=running)}
        action: {wait_act_8_lc__.state=completed}
    }
}
activity loop_act_9 {
    transition begin {
        domain: {(wait_act_8_lc__.state=running
                  & ((response.waitingDays < request.request__deadline)
                      & (oPrice.price > request.request__maxPrice))
                  & loop_act_9_lc__.state=not_started)}
            action: {loop_act_9_lc__.state=running}
        }
        (...)
}
```

Figura 7.10: Ejemplo de transformación de una actividad *while*.

```
<receive createInstance="yes" partnerLink="OliveOilPlnk"
         portType="tns:OliveOilService"
         operation="order" variable="request">
  <verbus:altMessages>
    <verbus:message>
      <request>
        <quantity>1</quantity>
        <maxPrice>3</maxPrice>
        <deadline>5</deadline>
        <customerId>c102000</customerId>
      </request>
    </verbus:message>
    <verbus:message>
      <request>
        <quantity>2</quantity>
        <maxPrice>6</maxPrice>
        <deadline>2</deadline>
        <customerId>c102000</customerId>
      </request>
    </verbus:message>
  </verbus:altMessages>
</receive>
```

```
activity receive_act_5 {
    transition begin {
        domain: {(main_act_4_lc__.state=running
                  & receive_act_5_lc__.state=not_started)}
        action: {receive_act_5_lc__.state=running}
    }
    transition complete_0 {
        domain: {receive_act_5_lc__.state=running}
        action: {(receive_act_5_lc__.state=completed
                  & process_oliveProcess_lc__.state=running
                  & (request.request__customerId=c102000
                     & request.request__maxPrice=3
                     & request.request__deadline=5
                     & request.request__quantity=1))}
    }
    transition complete_1 {
        domain: {receive_act_5_lc__.state=running}
        action: {(receive_act_5_lc__.state=completed
                  & process_oliveProcess_lc__.state=running
                  & (request.request__customerId=c102000
                     & request.request__maxPrice=6
                     & request.request__deadline=2
                     & request.request__quantity=2))}
    }
}
```

Figura 7.11: Ejemplo de transformación de una actividad *receive*.

```
<pick name="my_pick" createInstance="yes">
  <onMessage partnerLink="caller" portType="pt"
             operation="orderComplete" variable="request" >
    <empty name="message" />
    <verbus:altMessages>
      <verbus:message>
        <color><msg_colors>white</msg_colors></color>
        <isResponse>false</isResponse>
      </verbus:message>
      <verbus:message>
        <color><msg_colors>red</msg_colors></color>
        <isResponse>false</isResponse>
      </verbus:message>
    </verbus:altMessages>
  </onMessage>
  <onAlarm for="4D">
    <empty name="alarm" />
  </onAlarm>
</pick>
```

Figura 7.12: Fragmento BPEL4WS del ejemplo de transformación de una actividad *pick*.

```
activity my_pick_act_3 {
    transition begin {
        domain: {(receive_act_2_lc__.state=completed
                  & my_pick_act_3_lc__.state=not_started)}
        action: {(my_pick_act_3_lc__.state=running
                  & my_pick_act_3_pick_sel__.activated=none)}
    }
    transition begin_orderComplete_0_0 {
        domain: {(my_pick_act_3_lc__.state=running
                  & my_pick_act_3_pick_sel__.activated=none)}
        action: {(my_pick_act_3_pick_sel__.activated=orderComplete_0
                  & request.color__msg_colors=white & !request.isResponse)}
        }
    transition begin_orderComplete_0_1 {
        domain: {(my_pick_act_3_lc__.state=running
                  & my_pick_act_3_pick_sel__.activated=none)}
        action: {(my_pick_act_3_pick_sel__.activated=orderComplete_0
                  & request.color__msg_colors=red & !request.isResponse)}
    }
    transition complete_orderComplete_0 {
        domain: {(my_pick_act_3_lc__.state=running
                  & message_act_4_lc__.state=completed
                  & my_pick_act_3_pick_sel__.activated=orderComplete_0)}
        action: {(my_pick_act_3_lc__.state=completed
                  & my_pick_act_3_pick_sel__.activated=none)}
    }
    transition begin_onAlarm_0 {
        domain: {(my_pick_act_3_lc__.state=running
                  & my_pick_act_3_pick_sel__.activated=none)}
        action: {my_pick_act_3_pick_sel__.activated=onAlarm_0}
    }
    transition complete_onAlarm_0 {
        domain: {(my_pick_act_3_lc__.state=running
                  & alarm_act_5_lc__.state=completed
                  & my_pick_act_3_pick_sel__.activated=onAlarm_0)}
        action: {(my_pick_act_3_lc__.state=completed
                  & my_pick_act_3_pick_sel__.activated=none)}
    }
activity message_act_4 {
    transition empty {
        domain: {(my_pick_act_3_pick_sel__.activated=orderComplete_0
                  & message_act_4_lc__.state=not_started)}
    (...)
activity alarm_act_5 {
    transition empty {
        domain: {(my_pick_act_3_pick_sel__.activated=onAlarm_0
                  & alarm_act_5_lc__.state=not_started)}
    (...)
```

Figura 7.13: Representación mediante CFM del ejemplo de transformación de una actividad *pick*.

```
<invoke partnerLink="OilProviderPlnk" portType="tns:OilProviderService"
        operation="order" inputVariable="oOrder"
        outputVariable="oOrderConfirm">
  <verbus:altMessages>
    <verbus:message>
      <accepted>true</accepted>
    </verbus:message>
    <verbus:message>
      <accepted>false</accepted>
    </verbus:message>
  </verbus:altMessages>
</invoke>
```

```
activity invoke_act_13 {
    transition begin {
        domain: {(wait_act_8_lc__.state=completed &
                  invoke_act_13_lc__.state=not_started)}
        action: {invoke_act_13_lc__.state=running}
    }
    transition complete_0 {
        domain: {invoke_act_13_lc__.state=running}
        action: {(invoke_act_13_lc__.state=completed
                  & oOrderConfirm.accepted)}
    }
    transition complete_1 {
        domain: {invoke_act_13_lc__.state=running}
        action: {(invoke_act_13_lc__.state=completed
                  & !oOrderConfirm.accepted)}
    }
}
```

Figura 7.14: Ejemplo de transformación de una actividad *invoke*.

```
<scope name="scope0">
  <sequence>
    <receive ... />
    <scope name="scope1">
      <flow name="flow">
        <wait name="wait" for="4D" />
        <receive name="receive2" ... />
      </flow>
    </scope>
  </sequence>
</scope>
```

```
activity scope1_act_4 {
    transition cancel_begin {
        domain: {((scope1_act_4_lc__.state=running
                   | scope1_act_4_lc__.state=faulted
                   | scope1_act_4_lc__.state=compensating)
                  & (scope0_act_1_lc__.state=fault_cancelling
                     | scope0_act_1_lc__.state=cancelling))}
        action: {scope1_act_4_lc__.state=cancelling}
    }
    transition cancel_end {
        domain: {((scope1_act_4_lc__.state=cancelling
                   | scope1_act_4_lc__.state=terminating)
                  & (!(wait_act_6_lc__.state=running))
                  & (!(receive2_act_7_lc__.state=running))
                  & (!(flow_act_5_lc__.state=completed))
        action: {scope1_act_4_lc__.state=cancelled}
    }
    transition terminate {
        domain: {((scope1_act_4_lc__.state=running
                   | scope1_act_4_lc__.state=faulted
                   | scope1_act_4_lc__.state=compensating)
                  & scope0_act_1_lc__.state=terminating)}
        action: {scope1_act_4_lc__.state=terminating}
    } (...)
}
activity flow_act_5 {
    transition cancel {
        domain: {(flow_act_5_lc__.state=running
                  & (scope1_act_4_lc__.state=fault_cancelling
                     | scope1_act_4_lc__.state=cancelling
                     | scope1_act_4_lc__.state=terminating))}
        action: {flow_act_5_lc__.state=cancelled}
    } (...)
activity wait_act_6 {
    transition cancel {
        domain: {(wait_act_6_lc__.state=running
                  & (scope1_act_4_lc__.state=fault_cancelling
                     | scope1_act_4_lc__.state=cancelling
                     | scope1_act_4_lc__.state=terminating))}
        action: {wait_act_6_lc__.state=cancelled}
    } (...)
activity receive2_act_7 {
    transition cancel {
        domain: {(receive2_act_7_lc__.state=running
                  & (scope1_act_4_lc__.state=fault_cancelling
                     | scope1_act_4_lc__.state=cancelling
                     | scope1_act_4_lc__.state=terminating))}
        action: {receive2_act_7_lc__.state=cancelled}
    }
}
```

Figura 7.15: Procedimiento de cancelación.

```
<process name="echoString" ...>
  <scope name="scope0">
    <sequence name="sequence">
      <receive .../>
      <assign name="assign1">...</assign>
      <terminate name="terminate" />
    </sequence>
  </scope>
</process>
```

```
activity process_echoString_act__ {
    transition terminate_end {
        domain: {(process_echoString_lc__.state=terminating
                  & (scope0_act_1_lc__.state=completed
                     | scope0_act_1_lc__.state=not_started
                     | scope0_act_1_lc__.state=compensated
                     | scope0_act_1_lc__.state=cancelled))}
        action: {process_echoString_lc__.state=cancelled}
    }
    (...)
}
activity terminate_act_5 {
    transition terminate {
        domain: {(assign1_act_4_lc__.state=completed
                  & scope0_act_1_lc__.state=running
                  & terminate_act_5_lc__.state=not_started)}
        action: {(terminate_act_5_lc__.state=completed
                  & process_echoString_lc__.state=terminating)}
    }
}
```

Figura 7.16: Representación de la actividad *terminate*.

```
<process name="echoString" ...>
  <faultHandlers>
    <catch faultName="tns:fault1">
      <empty name="catch_1" />
    </catch>
    <catch faultName="tns:fault2">
      <empty name="catch_2" />
    </catch>
  </faultHandlers>

  <scope name="scope0">
    <faultHandlers>
      <catch faultName="tns:fault3">
        <throw name="throw_again" faultName="tns:fault2" />
      </catch>
      <catch faultName="tns:fault1">
        <empty name="catch_scope0_fault1" />
      </catch>
      <catch faultName="tns:fault4">
        <scope name="scope_in">
          <faultHandlers>
            <catch faultName="tns:fault2">
              <empty name="catch_scope0_fault2"/>
            </catch>
          </faultHandlers>
          <flow name="flow">
            <throw name="throw_1" faultName="tns:fault1" />
            <throw name="throw_2" faultName="tns:fault2" />
          </flow>
        </scope>
      </catch>
    </faultHandlers>

    <sequence name="sequence">
      <receive ... />
      <assign name="assign1"> ... </assign>
      <throw name="throw3" faultName="tns:fault3" />
    </sequence>
  </scope>
</process>
```

Figura 7.17: Proceso BPEL4WS tomado como ejemplo para ilustrar la representación del mecanismo de manejo de fallos.

```
activity scope0_act_3 {
    transition catch_completion {
        domain: {(scope0_act_3_lc__.state=faulted
                  & (throw_act_4_lc__.state=completed
                     | catch_scope0_fault1_act_5_lc__.state=completed
                     | scope_in_act_6_lc__.state=completed))}
        action: {scope0_act_3_lc__.state=completed}
    }
    transition fault_cancel_begin {
        domain: {(scope0_act_3_lc__.fault!=none
                  & scope0_act_3_lc__.state=running)}
        action: {scope0_act_3_lc__.state=fault_cancelling}
    }
    transition fault_cancel_end {
        domain: {(scope0_act_3_lc__.state=fault_cancelling
                  & (!(receive1_act_12_lc__.state=running))
                  & (!(assign1_act_13_lc__.state=running))
                  & (!(throw_act_14_lc__.state=running))
                  & (!(sequence_act_11_lc__.state=running)))}
        action: {scope0_act_3_lc__.state=faulted}
    }
    transition cancel_begin {
        domain: {((scope0_act_3_lc__.state=running
                   | scope0_act_3_lc__.state=faulted
                   | scope0_act_3_lc__.state=compensating)
                  & (process_echoString_lc__.state=fault_cancelling
                     | process_echoString_lc__.state=cancelling))}
        action: {scope0_act_3_lc__.state=cancelling}
    }
    transition cancel_end {
        domain: {((scope0_act_3_lc__.state=cancelling
                   | scope0_act_3_lc__.state=terminating)
                  & (!(receive1_act_12_lc__.state=running))
                  & (!(assign1_act_13_lc__.state=running))
                  & (!(throw_act_14_lc__.state=running))
                  & (!(sequence_act_11_lc__.state=running)))}
        action: {scope0_act_3_lc__.state=cancelled}
    }
    transition terminate {
        domain: {((scope0_act_3_lc__.state=running
                   | scope0_act_3_lc__.state=faulted
                   | scope0_act_3_lc__.state=compensating)
                  & process_echoString_lc__.state=terminating)}
        action: {scope0_act_3_lc__.state=terminating}
    }
    transition begin {
        domain: {scope0_act_3_lc__.state=not_started}
        action: {scope0_act_3_lc__.state=running}
    }
    transition complete {
        domain: {(sequence_act_11_lc__.state=completed
                  & scope0_act_3_lc__.state=running)}
        action: {scope0_act_3_lc__.state=completed}
    }
}
activity catch_scope0_fault1_act_5 {
    transition empty {
        domain: {(scope0_act_3_lc__.state=faulted
                & catch_scope0_fault1_act_5_lc__.state=not_started
                & scope0_act_3_lc__.fault_handler=catch_scope0_fault1_act_5)}
        action: {catch_scope0_fault1_act_5_lc__.state=completed}
}   }
```

Figura 7.18: Ejemplo de manejo de fallos en CFM.

```
/* catch fault urn_echo_echoService_fault3 activity scope0 */
activity throw_again_act_4 {
    transition throw {
        domain: {(scope0_act_3_lc__.state=faulted
                  & throw_again_act_4_lc__.state=not_started
                  & scope0_act_3_lc__.fault_handler=throw_again_act_4)}
        action: {(throw_again_act_4_lc__.state=completed
                  & process_echoString_lc__.fault_handler=catch_2_act_2)}
    }
}

activity throw_1_act_9 {
    transition throw {
        domain: {(flow_act_8_lc__.state=running
                  & scope_in_act_6_lc__.state=running
                  & throw_1_act_9_lc__.state=not_started)}
        action: {(throw_1_act_9_lc__.state=completed
                  & process_echoString_lc__.fault_handler=catch_1_act_1)}
    }
}

activity throw_2_act_10 {
    transition throw {
        domain: {(flow_act_8_lc__.state=running
                  & scope_in_act_6_lc__.state=running
                  & throw_2_act_10_lc__.state=not_started)}
        action: {(throw_2_act_10_lc__.state=completed
              & scope_in_act_6_lc__.fault_handler=catch_scope0_fault2_act_7)}
    }
}

activity throw3_act_14 {
    transition throw {
        domain: {(assign1_act_13_lc__.state=completed
                  & scope0_act_3_lc__.state=running
                  & throw3_act_14_lc__.state=not_started)}
        action: {(throw3_act_14_lc__.state=completed
                  & scope0_act_3_lc__.fault_handler=throw_again_act_4)}
    }
}
```

Figura 7.19: Representación de las cuatro actividades *throw* del proceso representado en la figura 7.17.

```
<flow name="flow" suppressJoinFailure="yes">
  <links>
    <link name="a-c" />
    <link name="a-d" />
    <link name="b-c" />
  </links>
  <empty name="a" for="3H">
    <source linkName="a-c"
      transitionCondition="bpws:getVariableData('request', 'color',
                                'msg_colors')=string('red')" />
    <source linkName="a-d"
      transitionCondition="bpws:getVariableData('request', 'color',
                                'msg_colors')!=string('red')" />
  </empty>
  <wait name="b" for="1H">
    <source linkName="b-c" />
  </wait>
  <wait name="c" for="1H">
    <target linkName="a-c" />
    <target linkName="b-c" />
  </wait>
  <empty name="d">
    <target linkName="a-d" />
  </empty>
</flow>
```

```
entity flow_a-c_lnk__: BPEL_lnk;
entity flow_a-d_lnk__: BPEL_lnk;
entity flow_b-c_lnk__: BPEL_lnk;
activity a_act_4 {
    transition empty {
        domain: {...}
        action: {(a_act_4_lc__.state=completed
                 & flow_a-c_lnk__.evaluated & flow_a-d_lnk__.evaluated
                 & flow_a-c_lnk__.evaluation=(request.color__msg_colors=red)
               & flow_a-d_lnk__.evaluation=(request.color__msg_colors!=red))}
    }
}
activity b_act_5 {
    transition begin {...}
    transition complete {
        domain: {...}
        action: {(b_act_5_lc__.state=completed
                 & flow_b-c_lnk__.evaluated & flow_b-c_lnk__.evaluation)}
    }
    transition cancel {
        domain: {...}
        action: {(b_act_5_lc__.state=cancelled
                 & flow_b-c_lnk__.evaluated & !flow_b-c_lnk__.evaluation)}
    }
}
activity c_act_6 {
    transition begin {
        domain: {(flow_act_3_lc__.state=running
                  & process_echoString_lc__.state=running
                  & c_act_6_lc__.state=not_started)
                  & flow_a-c_lnk__.evaluated & flow_b-c_lnk__.evaluated
                  & flow_a-c_lnk__.evaluation & flow_b-c_lnk__.evaluation)}
        action: {c_act_6_lc__.state=running}
    } (...)
```

Figura 7.20: Representación de vínculos (*links*).

```
<scope name="scope">
  <eventHandlers>
    <onMessage partner="callerPrice" portType="tns:pricePT"
               operation="getPrice" variable="price" >
      <empty name="handler" />
    </onMessage>
  </eventHandlers>
  <receive partner="caller" portType="tns:echoPT"
           operation="echo" variable="request"
           createInstance="yes" name="receive">
  </receive>
</scope>
```

```
activity scope_ehandler_act_2 {
    transition begin {
        domain: {(scope_act_1_lc__.state=running
                  & scope_ehandler_act_2_lc__.state=not_started)}
        action: {(scope_ehandler_act_2_lc__.state=running
                  & handler_act_3_lc__.state=not_started)}
    }
    transition complete {
        domain: {(scope_ehandler_act_2_lc__.state=running
                  & handler_act_3_lc__.state=completed)}
        action: {scope_ehandler_act_2_lc__.state=not_started}
    }
}
activity handler_act_3 {
    transition empty {
        domain: {(scope_ehandler_act_2_lc__.state=running
                  && handler_act_3_lc__.state=not_started)}
        action: {handler_act_3_lc__.state=completed}
    }
}
```

Figura 7.21: Representación de manejadores de eventos.

```
<verbus:property type="invariant"
  expression="bpws:getVariableData('response', 'waitingDays') &lt;=
            bpws:getVariableData('request', 'request', 'deadline')" />
<verbus:property type="goal"
  expression="verbus:activityState('response') = 'completed'
            or verbus:activityState('response_neg') = 'completed'" />
<verbus:property type="pre-req" activity="response"
  expression="verbus:activityState('response_neg') = 'not_running'" />
<verbus:property type="exe" activity="response_neg" />
```

```
invariant invariant_0 {
    (response.waitingDays <= request.request__deadline)
}
goal goal_1 {
    (response_act_19_lc__.state=completed
     | response_neg_act_3_lc__.state=completed)
}
prereq prereq_2 (response_act_19, begin) {
    (response_neg_act_20_lc__.state=not_started)
}
exe exe_3 (response_neg_act_20, begin);
```

Figura 7.22: Ejemplo de transformación de requisitos BPEL4WS a requisitos CFM.

Capítulo 8

Caso de estudio

Este capítulo propone un escenario de aplicación de la arquitectura propuesta en esta tesis doctoral, mediante un caso de estudio. Los objetivos son varios. Por una parte, clarificar el contenido de las propuestas realizadas mediante su aplicación a un ejemplo. Por otra, mostrar cómo la utilización de estas propuestas podría ayudar a una empresa en un entorno B2B. Finalmente, contribuir a la validación de las propuestas.

En primer lugar se plantea el caso de estudio, basado en una empresa de embotellado y distribución de aceite de oliva. Se plantea el diseño de un proceso BPEL4WS para dicho caso. A continuación, se plantean diversos requisitos y se lleva a cabo su verificación mediante Spin. Entre estos requisitos se recogen tanto generales como específicos, incluyendo requisitos expresados mediante lógicas temporales. Finalmente, se concluye analizando la adecuación del modelo planteado en esta tesis al caso de estudio.

8.1. Planteamiento del caso de estudio

Supongamos una empresa de embotellado de aceite de oliva. Esta empresa obtiene el aceite de un proveedor que, dependiendo de las condiciones puntuales del mercado, puede variar los precios cada día. La empresa no está sujeta a requisitos temporales estrictos para comprar aceite de su proveedor porque cuenta con sus propias reservas. Por tanto, le interesa realizar compras sólo cuando el precio sea lo suficientemente bajo.

El proveedor permite a la empresa consultar precios y realizar pedidos de forma electrónica a través de un servicio Web. La empresa desea desarrollar un proceso BPEL4WS que automatice las interacciones con el proveedor, con el fin de reducir la carga de trabajo dedicada por sus empleados a las interacciones con este.

El programa de gestión de los operarios de la empresa permite, cuando las reservas decrezcan y necesiten realizar un pedido, cursar una solicitud a este proceso, indicando la cantidad de aceite a comprar, el precio máximo admisible y el plazo máximo de espera. El proceso se encarga de consultar el precio todos los días, y realizar el pedido cuando este sea menor o igual que el precio máximo admisible. Si finaliza el plazo, entonces se realiza el pedido inmediatamente, aunque este sea superior al máximo admisible. El proceso proporciona también un servicio que permite cancelar un pedido cursado, por si ocurriese cualquier circunstancia en que fuese necesario, como cambios abruptos en la situación del mercado que hiciesen variar los precios, cambios en los plazos, etc.

Dadas las pérdidas económicas que un error en este proceso podría causar a la empresa, los directivos no permiten su implantación sin que se realice un análisis exhaustivo del mismo. Plantean como requisitos más importantes los siguientes:

1. El proceso debe contestar siempre al usuario que realice una solicitud, incluso en el caso de que se produzca un fallo en alguna de las invocaciones al proveedor de aceite.

2. El proceso debe contestar al usuario que realiza una solicitud en el plazo máximo que este establezca en dicha solicitud.

3. El precio de compra del aceite de oliva debe ser aceptable. Esto significa que debe ser menor o igual que el precio máximo indicado en la solicitud, salvo que se supere el plazo, en cuyo caso resulta aceptable que sea mayor.

4. Una vez recibida una oferta del proveedor por un precio menor o igual que el precio máximo, en algún momento futuro se realizará el pedido. A partir de la recepción de dicha oferta, en ningún caso se volverá a realizar una consulta de precio ni se realizará una nueva espera de un día.

8.2. Diseño inicial del proceso

En este apartado se presenta el diseño inicial del proceso BPEL4WS de gestión de pedidos de aceite de oliva.

8.2.1. Interfaces WSDL y tipos de datos

En la figura 8.1 se muestra la interfaz WSDL del servicio Web proporcionado por el proveedor de aceite a sus clientes. Consta de dos operaciones, una para realizar una consulta de precio y otra para realizar un pedido. Ambas operaciones son síncronas y pueden generar un fallo. La cantidad de aceite del pedido se expresa como un número entero de litros. El precio por litro se expresa como un entero que representa céntimos de euro.

Por otra parte, el proceso BPEL4WS ofrece una interfaz, descrita también mediante WSDL, para las aplicaciones cliente de los operarios de la empresa de embotellamiento. Esta interfaz consta de una única operación, de tipo síncrono. En la figura 8.2 se muestra la definición de la interfaz. Aunque una operación síncrona no resulta apropiada cuando la respuesta a la invocación puede tardar mucho tiempo, se modela de esta forma por motivos de sencillez. El hecho de que la respuesta sea síncrona, y no asíncrona, no afecta a la verificación del proceso.

Adicionalmente, los operarios disponen de una interfaz de control para interactuar con una instancia del proceso en ejecución. Concretamente, esta interfaz permite a un operario cancelar la instancia del proceso. Se muestra en la figura 8.3.

8.2.2. El proceso BPEL4WS

En el apéndice A.1 se muestra la definición del proceso BPEL4WS de gestión de pedidos de aceite de oliva. Este proceso realiza lo siguiente:

```
<types>
  <xsd:schema ...>
    <xsd:simpleType name="OilTypeT">
      <xsd:restriction base="xsd:string">
        <xsd:enumeration value="olive"/>
        <xsd:enumeration value="soja"/>
        <xsd:enumeration value="sunflower"/>
      </xsd:restriction>
    </xsd:simpleType>
  </xsd:schema>
  (...)
</types>
<message name="OilOrder">
  <part name="quantity" type="xsd:integer" />
  <part name="customerId" type="xsd:string" />
  <part name="oilType" type="tns:OilTypeT"/>
</message>
<message name="OilConfirmation">
  <part name="accepted" type="xsd:boolean"/>
</message>
<message name="OilPriceResponse">
  <part name="price" type="xsd:integer"/>
</message>
<portType name="OilProviderService">
  <operation name="order">
    <input message="tns:OilOrder" />
    <output message="tns:OilConfirmation" />
    <fault name="tns:OilFault" />
  </operation>
  <operation name="getPrice">
    <input message="tns:OilOrder" />
    <output message="tns:OilPriceResponse" />
    <fault name="tns:OilFault" />
  </operation>
</portType>
```

Figura 8.1: Fragmento de la interfaz WSDL del proveedor de aceite.

1. Espera la recepción de una solicitud al servicio *OliveOilService*. Este mensaje crea una nueva instancia del proceso.

2. Se establece el valor inicial de las variables del proceso.

3. Se realiza una consulta de precio en el servicio *OilProviderService* del proveedor.

4. Mientras el precio ofertado sea superior al precio máximo solicitado por el usuario:

   a) Se bloquea la ejecución durante un día. Se puede interrumpir este bloqueo si se recibe una solicitud de cancelación por parte del usuario.
   b) Se solicita de nuevo el precio al proveedor.

5. Si no se ha recibido previamente ningún mensaje de cancelación del usuario, se realiza el pedido en el servicio *OilProviderService* del proveedor. Esta invocación es síncrona: espera una confirmación (positiva o negativa) por parte del proveedor.

6. Se responde al cliente confirmando el pedido. La confirmación puede ser positiva o negativa.

```
<types>
  <xsd:schema ...>
    <xsd:complexType name="OliveOilRequestT">
      <xsd:sequence>
        <xsd:element name="quantity" type="xsd:integer" />
        <xsd:element name="maxPrice" type="xsd:integer" />
        <xsd:element name="deadline" type="xsd:integer" />
        <xsd:element name="customerId" type="xsd:string" />
      </xsd:sequence>
    </xsd:complexType>
  </xsd:schema>
  (...)
</types>
<message name="OliveOilRequest">
  <part name="request" type="tns:OliveOilRequestT"/>
</message>
<message name="OliveOilResponse">
  <part name="request" type="tns:OliveOilRequestT"/>
  <part name="ordered" type="xsd:boolean"/>
  <part name="price" type="xsd:integer"/>
  <part name="totalPrice" type="xsd:integer"/>
  <part name="waitingDays" type="xsd:integer"/>
</message>
<portType name="OliveOilService">
  <operation name="order">
    <input message="tns:OliveOilRequest" />
    <output message="tns:OliveOilResponse" />
    <fault name="tns:OliveOilFault" />
  </operation>
</portType>
```

Figura 8.2: Fragmento de la interfaz WSDL del proceso de gestión de pedidos de aceite de oliva.

8.2.3. Abstracción de datos

Las variables de tipo entero son una de las causas principales de las explosiones de estados en los modelos de procesos, dado su gran rango de valores. Esto resulta especialmente crítico en herramientas que apliquen *model checking simbólico*, debido a que las variables enteras incrementan drásticamente el tamaño de los OBDDs. Esto no resulta crítico en la herramienta Spin, dado que no aplica esta técnica de *model checking*. En cualquier caso, para evitar esta explosión de estados, es recomendable realizar una abstracción de las variables enteras de forma manual, reduciendo los posibles valores a un rango más pequeño. En este caso, se hace lo siguiente:

- Cantidad de aceite a comprar: dado que no influye en el proceso en absoluto, se puede modelar como un tipo de datos abstracto en el modelo CFM. Si se realiza la verificación con Spin no es necesario realizar esta abstracción: basta con fijar un valor inicial que no varíe a lo largo de la verificación. Si el verificador utilizase *model checking simbólico*, sí sería necesario abstraer el tipo de datos para no incrementar inútilmente la complejidad de los OBDDs.

- Plazo: no es necesario abstraer su valor si se fija un valor no demasiado alto en la inicialización del mensaje de entrada. El hecho de fijarlo en un valor superior a 1 es

```
<types>
  <xsd:schema ...>
    <xsd:simpleType name="CommandT">
      <xsd:restriction base="xsd:string">
        <xsd:enumeration value="cancel"/>
      </xsd:restriction>
    </xsd:simpleType>
  </xsd:schema>
  (...)
</types>
<message name="CommandMessage">
  <part name="request" type="tns:CommandT"/>
</message>
<portType name="OliveOilServiceControl">
  <operation name="command">
    <input message="tns:CommandMessage" />
  </operation>
</portType>
```

Figura 8.3: Fragmento de la interfaz WSDL de control del proceso de gestión de pedidos de aceite de oliva.

suficiente para que se analicen todos los posibles caminos durante la verificación. Se reduce la complejidad por el hecho de no ser excesivamente elevado este valor.

- Precio: en lugar de utilizar un precio en céntimos de euro, que podría dar lugar a un rango excesivo en las variables de este tipo, se pueden utilizar valores simbólicos bajos. Por ejemplo, $1, 2, \ldots, 6$. Si se fija un precio máximo en un valor intermedio, se consigue que se verifiquen todas las posibles alternativas. Aunque resultaría más conveniente definir estos precios mediante una variable de rango enumerado, se decide, por el hecho de utilizar Spin, mantenerla como tipo entero para poder utilizar los operadores de comparación *menor–que* y *mayor–que*.

Esta abstracción en los rangos de valores se refleja en los distintos mensajes alternativos que pueden ser recibidos por el proceso, expuestos en el siguiente apartado.

8.2.4. Modelado de las entidades externas

Desde el punto de vista del proceso, existen dos entidades externas con las cuales interactúa: el usuario y el proveedor de aceite. El usuario invoca servicios proporcionados por el proceso, correspondientes a los tipos de puerto *OliveOilService* y *OliveOilControlService*. El proceso invoca servicios proporcionados por el proveedor de aceite, en el tipo de puerto *OilProviderService*.

Tal y como se expone en el apartado 7.2.2, es necesario proporcionar un modelo abstracto de estas entidades. En este ejemplo, es suficiente con modelar las recepciones de mensajes desde estas entidades con varios mensajes alternativos que representen los distintos tipos de casos que pueden ocurrir en el proceso. Concretamente, se definen los siguientes:

- Recepción de la petición inicial del usuario. Se recibe un identificador de usuario, la cantidad de aceite, el precio máximo por litro y el número máximo de días en que el pedido debe ser confirmado. Se puede especificar un único mensaje alternativo,

para mantener estos valores constantes: permitir que el precio proporcionado por el proveedor fluctúe en valores superiores e inferiores al precio mínimo es suficiente para forzar la ocurrencia de todas las posibles alternativas.

- Consulta de precio al proveedor de aceite. El proveedor contesta con su oferta para el pedido especificado en el mensaje de entrada. Se proporcionan varios mensajes con precios alternativos, con precios menores, iguales y mayores al precio máximo fijado en el punto anterior. Se realiza esto tanto en la actividad *invoke* anterior al bucle como en la interna del bucle.

- Recepción de una solicitud de cancelación por parte del usuario. Sólo hay un contenido válido para este mensaje (el comando de cancelación). Por tanto, se especifica un único mensaje alternativo.

- Confirmación del pedido con el proveedor. El proveedor puede devolver dos tipos de respuestas a esta petición: confirmación o cancelación del pedido realizado.

En la figura 8.4 se muestra cómo se declararían estos mensajes alternativos en el interior de la propia definición del proceso BPEL4WS.

8.3. Análisis de requisitos generales

El primer análisis del proceso BPEL4WS puede consistir en comprobar simplemente si el proceso es conforme a los requisitos generales expuestos en el apartado 6.2.4. Estos requisitos permiten comprobar que la estructura del proceso es correcta, así como detectar fácilmente algunos tipos de errores en las definiciones.

8.3.1. Planteamiento de los requisitos

Para realizar este análisis, es necesario, en primer lugar, generar una especificación con los requisitos generales. No es necesario especificar los requisitos en el modelo BPEL4WS, dado que el conversor de BPEL4WS a CFM es capaz de generarlos de forma automática.

- Requisito RG–1: la verificación de este requisito exige dos tipos de verificación distintos en Spin. Por una parte, es necesario verificar un requisito de tipo *objetivo* que imponga que el proceso debe estar en un estado final correcto (*completed, cancelled* o *compensated*). Por otra, es necesario realizar una verificación de *viveza*, para comprobar que no hay ningún ciclo de duración infinita *sin progreso*. El primer requisito *objetivo* de la figura 8.5 se corresponde con la primera parte del requisito RG–1.

- Requisito RG–2: este requisito impone que ninguna transición funcional puede ser ejecutable una vez el proceso haya finalizado. Se puede expresar como un pre–requisito, asociado a cada una de las transiciones funcionales, que imponga que el proceso no puede estar en un estado final (*completed, cancelled* o *compensated*). En la figura 8.5 se propone la aplicación de este requisito a dos actividades distintas. Debe ser aplicado, de la misma forma, al resto de actividades del proceso.

- Requisito RG–3: este requisito impone que, cuando finaliza el proceso, todas sus actividades, o bien no han comenzado su ejecución, o bien están en un estado final válido. La figura 8.5 muestra un fragmento del enunciado de este requisito, aplicado al proceso ejemplo.

- Requisito RG–4: dada la metodología de transformación presentada en el capítulo 7 para BPEL4WS, es posible que algunas transiciones funcionales, relacionadas con manejo de fallos, cancelación o compensación, no sean ejecutables. Por tanto, en general, no resulta conveniente aplicar este requisito cuando se aplica esta metodología de conversión.

- Requisito RG–5: al contrario que el requisito RG–4, este requisito sí resulta interesante para procesos BPEL4WS transformados según la metodología expuesta en esta tesis. Se muestra en la figura 8.5 su aplicación al ejemplo.

8.3.2. Verificación de los requisitos generales en el ejemplo

Una vez generado el proceso Promela, incluyendo los requisitos generales, se puede utilizar Spin para comprobar que el proceso cumpla dichos requisitos. En primer lugar, se pueden comprobar los requisitos 1 (parcialmente), 2, 3 y 5 mediante un análisis de tipo *seguridad*[1]). Con el ejemplo propuesto, el mensaje de resultado de Spin indica que no hay errores en invariantes, objetivos, pre–requisitos ni post–requisitos:

```
Full statespace search for:
        never claim              - (not selected)
        assertion violations     +
        cycle checks            - (disabled by -DSAFETY)
        invalid end states     - (disabled by -E flag)

State-vector 132 byte, depth reached 325, errors: 0
   50470 states, stored
    5667 states, matched
   56137 transitions (= stored+matched)
     741 atomic steps
hash conflicts: 180 (resolved)
```

El hecho de que los requisitos anteriores se verifiquen es debido, fundamentalmente, a la robustez de la metodología de conversión del proceso BPEL4WS a CFM. Cualquier error en esta metodología provocaría, con mucha probabilidad, la violación de alguno de los requisitos anteriores.

En su análisis de ejecutabilidad, Spin detecta que varias líneas de la definición Promela no son ejecutables. Cada una de estas líneas se corresponde con una transición funcional de la definición CFM del proceso. Sin embargo, no se viola el requisito RG–5, dado que en todas las actividades se ejecuta, al menos, su transición funcional de inicio. Es necesario analizar el motivo de la no ejecución de estas transiciones funcionales. En ninguno de los casos se debe a un error, sino una situación normal dada la definición del proceso:

[1] Para realizar un análisis de tipo seguridad con XSpin, es necesario seleccionar la propiedad de verificación *Safety (state properties)/Assertions*. Para verificar requisitos de ejecutabilidad, es necesario activar también la opción *Report unreachable code*.

- La transición funcional *terminate_end* del proceso global no se ejecuta nunca porque no se ha incluido ninguna actividad *terminate* en la definición del proceso.
- Las transiciones funcionales de cancelación de algunas actividades no son ejecutables. En todas las ocurrencias de este caso se debe a que no existe ninguna actividad *terminate* en el proceso y a que, dada la posición de las actividades en el proceso, es imposible que ocurra un *fallo* que afecte a dichas actividades mientras están en ejecución.
- La transición funcional de finalización por defecto de la actividad *switch* no es ejecutable. Se debe a que las condiciones de los dos casos definidos en dicha actividad cubren todas las alternativas posibles, siendo imposible que ambas se evalúen, a la vez, con valor *falso*.

Tras el análisis anterior, se han verificado todos los requisitos generales excepto RG–1, que sólo ha sido verificado parcialmente, y RG–4, que no debe ser verificado. Para completar la verificación, es necesario realizar un análisis de *viveza*[2], con el objetivo de detectar posibles ciclos *sin progreso*. De nuevo, Spin indica que no hay errores, con lo cual se concluye que el requisito RG–1 también es cierto para este proceso:

```
Full statespace search for:
        never claim               +
        assertion violations      + (if within scope of claim)
        non-progress cycles       + (fairness disabled)
        invalid end states      - (disabled by never claim)

State-vector 136 byte, depth reached 648, errors: 0
  100939 states, stored (151408 visited)
  124344 states, matched
  275752 transitions (= visited+matched)
    2964 atomic steps
hash conflicts: 7870 (resolved)
```

8.3.3. Ciclos sin progreso

Con el fin de ilustrar la capacidad de Spin para detectar incumplimientos de los requisitos generales, se propone una pequeña modificación al proceso BPEL4WS. Se elimina el plazo para realizar el pedido. Para ello, se elimina la actividad de incremento del contador de días dentro del bucle *while*, se eliminan las invariantes y objetivos que hacen referencia a este contador, y se modifica la condición del bucle *while* eliminando la comprobación de plazo:

```
<while name="wait" condition="bpws:getVariableData('oPrice', 'price') >
  bpws:getVariableData('request', 'request', 'maxPrice')
  and not(bpws:getVariableData('cancelled'))">
```

Si se ejecuta de nuevo un análisis de *viveza*, Spin detecta un *ciclo sin progreso*, y muestra una traza en que se produce dicho ciclo:

```
[process_oliveProcess_lc__.fault_handler = 1]        <merge 256 now @256>
<<<<<START OF CYCLE>>>>>
 69:        proc  1 (oliveProcess) line 295 "pan_in" (state 141)
```

[2]Para realizar un análisis de viveza, es necesario activar la opción *Liveness (cycles/sequences)/Non–progress cycles* de XSpin.

```
[(((loop_act_11_lc__.state==1)&&(askPrice2_act_12_lc__.state==2)))]
 71:        proc  1 (oliveProcess) line 297 "pan_in" (state 142)
[loop_act_11_lc__.state = 2]
(...)
105:        proc  1 (oliveProcess) line 305 "pan_in" (state 146)
[askPrice2_act_12_lc__.state = 2]        <merge 256 now @147>
105:        proc  1 (oliveProcess) line 305 "pan_in" (state 147)
[process_oliveProcess_lc__.fault_handler = 1]        <merge 256 now @256>
spin: trail ends after 105 steps
```

Este ciclo se debe a que existe la posibilidad de que, infinitas veces consecutivas, el proveedor de aceite ofrezca un precio superior al máximo aceptable, de tal forma que el proceso no puede abandonar nunca el bucle *while*.

Cabe la posibilidad de que el diseñador del proceso considere que este comportamiento es correcto, asumiendo que, más tarde o más temprano, el usuario de una instancia del proceso que esté en este bucle acabaría cancelándola. Si fuese el caso, puede marcar la actividad de consulta de precio interna al bucle mediante una cláusula *progress*. De esta forma, la herramienta de verificación no considerará este bucle de duración infinita como un error.

8.4. Análisis de requisitos específicos

En una fase de análisis más avanzada, se puede definir una especificación con requisitos específicos para el proceso. El diseñador, o la persona que se encargue de realizar la verificación, debe especificar los requisitos que crea convenientes, pertenecientes a cualquiera de los tipos presentados en el apartado 6.2.3.

A continuación se formalizan, para la definición del proceso, los tres primeros requisitos propuestos en el apartado 8.1. En la figura 8.6 se codifican estos requisitos, tanto con sintaxis XPath como mediante el formalismo CFM. Son los siguientes:

- Objetivo *response_sent*: cuando finaliza el proceso, debe haber enviado una respuesta al cliente. Dado que se responde al cliente en las actividades *response* y *response_neg*, al menos una de ellas debe haber sido ejecutada.

- Invariante *deadline*: el número de días transcurridos desde la recepción de la solicitud del usuario debe ser siempre menor o igual que el plazo establecido en dicha solicitud.

- Invariante *acceptable_price*: el precio al que se solicita el aceite de oliva debe ser aceptable. Esto implica que, si no se ha cumplido el plazo, debe ser menor o igual que el precio máximo fijado en la solicitud del cliente.

Si se realiza una nueva verificación de seguridad, en este caso aplicando estos requisitos específicos, Spin es capaz de demostrar que se cumplen todos ellos:

```
Full statespace search for:
        never claim                 - (not selected)
        assertion violations        +
        cycle checks               - (disabled by -DSAFETY)
        invalid end states        +

State-vector 132 byte, depth reached 292, errors: 0
   48073 states, stored
    5667 states, matched
```

```
   53740 transitions (= stored+matched)
     741 atomic steps
hash conflicts: 224 (resolved)
```

8.4.1. Requisitos que no se cumplen

Para mostrar la capacidad de Spin para detectar errores, se analiza un caso típico en programación: la gestión errónea de un caso límite. Supóngase que el diseñador del proceso espera que el tiempo de respuesta sea *estrictamente menor* al plazo especificado por el cliente, esto es, que el requisito al que se ha llamado *deadline* cambia el signo "$\leq$" por "$<$":

```
<verbus:property type="invariant" name="deadline"
  expression="bpws:getVariableData('response', 'waitingDays') &lt;
             bpws:getVariableData('request', 'request', 'deadline')
             or verbus:activityState('oliveProcess') = 'not_started'" />
```

Al hacer el cambio en el requisito, ha sido necesario también especificar que la invariante aplica una vez recibido el mensaje del usuario. Esto es debido a que, hasta que no se reciba, tanto el plazo como el número de días transcurridos toman el valor inicial *cero*. Obviamente, esta situación, anterior a la llegada de la petición del usuario, no debe ser tenida en cuenta en la verificación.

Si se realiza de nuevo la verificación, Spin detecta una violación de este requisito, y muestra una traza en la cual el número de días de espera es igual al plazo. El final de la traza mostrada es el siguiente:

```
186:        proc  1 (oliveProcess) line 418 "pan_in" (state 245)
     [assert(((response.price<=request.request__maxPrice)||
      (response.waitingDays==request.request__deadline)))]
187:        proc  1 (oliveProcess) line 293 "pan_in" (state 136)
     [((((loop_act_11_lc__.state==1)&&(process_oliveProcess_lc__.state==1))
       &&(incdays_act_12_lc__.state==0)))]
188:        proc  1 (oliveProcess) line 295 "pan_in" (state 137)
     [incdays_act_12_lc__.state = 2]        <merge 244 now @138>
188:        proc  1 (oliveProcess) line 295 "pan_in" (state 138)
     [response.waitingDays = (response.waitingDays+1)]        <merge 244 now @244>
spin: line 416 "pan_in", Error: assertion violated
spin: text of failed assertion: assert(((response.waitingDays
                                    <request.request__deadline)
                                    ||(process_oliveProcess_lc__.state==0)))
```

Si en ese punto de la traza se analiza el valor de las variables, se puede comprobar que el mensaje del cliente ha sido recibido y, por otra parte, las variables que almacenan el plazo y el número de días transcurridos toman el mismo valor: 5.

Con ayuda de la traza asociada a este error, el diseñador será capaz de localizar el error en la condición del bucle *while* y corregirlo.

8.5. Análisis de requisitos de lógica temporal

Los requisitos expresados mediante lógicas temporales permiten expresar causalidades y relaciones temporales entre propiedades. El cuarto requisito expresado en el planteamiento del caso de estudio, en el apartado 8.1 pertenece a este tipo de requisitos. Dado que se trata de un enunciado complejo, se puede dividir en los siguientes requisitos:

1. Una vez recibida una oferta por un precio menor o igual que el precio máximo, en algún momento futuro se ejecutará la actividad de realización del pedido.
2. Una vez recibida una oferta por un precio menor o igual que el precio máximo, en ningún caso se volverá a ejecutar ninguna de las actividades de consulta de precio o de espera de un día.

8.5.1. Un requisito temporal que se cumple

En este apartado se muestra cómo enunciar y verificar el primero de los requisitos anteriores. Se puede formular mediante un requisito LTL conforme al patrón $\mathbf{AG}(p \implies \mathbf{F}q)$. Esto es, si se alcanza un estado que verifique el predicado p, entonces eventualmente se alcanzará un estado que verifique el predicado q. El predicado p modela la ocurrencia de la recepción de una oferta menor que el precio máximo. El predicado q modela la ejecución de la actividad de envío de la solicitud de pedido.

El predicado p debe ser cierto únicamente si se alcanza un estado, en algún momento después de la ejecución de la primera consulta de precio, en el cual el precio ofertado sea menor o igual que el precio máximo. Adicionalmente, es necesario exigir que todavía no se haya iniciado la ejecución de la actividad de realización del pedido, para evitar que esta parte del requisito sea considerada tras hacer realizado el pedido. El predicado q debe ser cierto durante la ejecución de la actividad de realización del pedido. En la figura 8.7 se muestra este requisito expresado mediante XPath, CFM y Promela. Nótese que XPath no define el operador $\implies$, por lo que se reemplaza por la expresión equivalente $p \implies q \equiv \neg p \vee q$.

La verificación de este requisito demuestra que, efectivamente, la definición del proceso cumple este requisito. Para realizar esta verificación, es necesario activar las opciones *verificar cláusula "never"* y *propiedades de viveza / ciclos de aceptación*. La salida que muestra Spin (abreviada) es la siguiente:

```
warning: for p.o. reduction to be valid the never claim must be stutter-invariant
(never claims generated from LTL formulae are stutter-invariant)
depth 93: Claim reached state 9 (line 445)
(Spin Version 4.2.5 -- 2 April 2005)
        + Partial Order Reduction

Full statespace search for:
        never claim             +
        assertion violations    + (if within scope of claim)
        acceptance   cycles     + (fairness disabled)
        invalid end states    - (disabled by never claim)

State-vector 136 byte, depth reached 557, errors: 0
    7374 states, stored (7866 visited)
    1444 states, matched
    9310 transitions (= visited+matched)
      95 atomic steps
hash conflicts: 7 (resolved)
```

8.5.2. Un requisito temporal que no se cumple

Supongamos que el diseñador desease imponer el siguiente requisito: *"Una vez recibida una oferta por un precio menor o igual que el precio máximo, en algún momento futuro se contestará al cliente confirmando la realización del pedido."*

Este requisito es relativamente similar al presentado en el apartado anterior, pero se diferencia en el enunciado del predicado q. En este caso, el predicado exige que el campo *ordered* de la contestación al cliente indique que el pedido ha sido realizado. Con sintaxis Promela, el único cambio con respecto a la figura 8.7 sería la definición de q:

```
#define q (((response_act_19_lc__.state==BPEL_lc_state_running)
             ||(response_neg_act_3_lc__.state==BPEL_lc_state_running))
           && response.ordered)
```

Esto es, se exige que en algún momento futuro alguna de las dos actividades de contestación al cliente (la normal y la del manejador de fallos) esté en ejecución, así como que, en dicho instante, el valor del campo *ordered* del mensaje de contestación tenga valor *cierto*.

Si se realiza una verificación, Spin detecta que este requisito no se cumple, y muestra un contraejemplo al mismo. Analizando el contraejemplo, el diseñador puede comprobar que, una vez obtenida una oferta aceptable por parte del proveedor, una invocación al servicio de solicitud de pedidos puede lanzar un fallo, que provoca que se ejecute el manejador de fallos del proceso y se conteste al cliente con el campo *ordered* con valor *falso*. Dado que este es un comportamiento deseado en el proceso, el diseñador concluye que el requisito, en realidad, no es correcto.

Supongamos que el diseñador refina este requisito para excluir los casos en que ocurra un fallo. Puede hacer esto cambiando ligeramente el requisito: $\mathbf{AG}(p \implies \mathbf{F}(q \vee t))$. Se introduce un nuevo predicado t, estableciendo que, eventualmente en el futuro, o bien ocurre q o bien ocurre un fallo (predicado t):

```
#define t ((process_oliveProcess_lc__.fault_handler !=
            BPEL_lc_fault_handler_sequence_act_1))
```

Si se repite la verificación, Spin indicará, de nuevo, que no se cumple el requisito. En este caso, encuentra un contraejemplo distinto: no se produce un *fallo* en la invocación al servicio de solicitud de pedido, sino que este envía una respuesta normal, pero indicando que el pedido no ha sido aceptado (campo *accepted* con valor *falso*). Como consecuencia, se contesta al cliente indicando que el pedido no ha podido ser realizado.

De nuevo, el diseñador puede observar que este es un comportamiento deseado en el proceso. Por tanto, el requisito no se cumple debido a que, en realidad, no formaliza correctamente los deseos del diseñador.

8.6. Conclusiones

En este capítulo se han aplicado los conceptos desarrollados en esta tesis a un ejemplo sencillo de proceso de negocio, definido mediante BPEL4WS. Con ello, se ha mostrado cómo la arquitectura propuesta resulta adecuada para la verificación de procesos, y qué tipos de requisitos puede plantear un usuario de esta arquitectura.

A la vista de los resultados obtenidos de este caso de estudio, se puede concluir que la arquitectura propuesta resulta útil para la verificación de procesos de negocio. Por una parte, el modelo CFM es capaz de representar correctamente el proceso BPEL4WS planteado. Por otra, la variedad de requisitos que se han planteado muestra que esta arquitectura es considerablemente flexible con respecto a la definición de especificaciones.

Aunque no se ha realizado un análisis formal de tiempo de computación y cantidad de memoria requeridos para las verificaciones mediante Spin, sí cabe resaltar que todas las verificaciones planteadas en este capítulo han sido verificadas con un ordenador personal convencional[3] en tiempos no superiores a los $10s$ y consumo de memoria menor de $10MB$. Si bien es cierto que el proceso planteado no tiene una complejidad elevada, esto sugiere que en esta arquitectura probablemente se podrían verificar procesos considerablemente más complejos.

[3]Procesador AMD Athlon XP de 32 bits a 1,4 Mhz, con 768 MB de memoria RAM y sistema operativo Linux 2.4.

```
<receive createInstance="yes" partnerLink="OliveOilPlnk"
  portType="tns:OliveOilService"
  operation="order" variable="request">
  <verbus:altMessages>
    <verbus:message>
      <request>
        <quantity>1</quantity>
        <maxPrice>3</maxPrice>
        <deadline>5</deadline>
        <customerId>c102000</customerId>
      </request>
    </verbus:message>
  </verbus:altMessages>
</receive>
(...)
<invoke name="askPrice1"
  partnerLink="OilProviderPlnk" portType="tns:OilProviderService"
  operation="getPrice" inputVariable="oOrder" outputVariable="oPrice">
  <verbus:altMessages>
    <verbus:message><price>1</price></verbus:message>
    <verbus:message><price>2</price></verbus:message>
    <verbus:message><price>3</price></verbus:message>
    <verbus:message><price>4</price></verbus:message>
    <verbus:message><price>5</price></verbus:message>
    <verbus:message><price>6</price></verbus:message>
  </verbus:altMessages>
</invoke>
(...)
<onMessage partnerLink="OliveOilControlPlnk"
  portType="OliveOilServiceControl"
  operation="command"
  variable="controlCommand">
  <assign>...</assign>
  <verbus:altMessages>
    <verbus:message><request>cancel</request></verbus:message>
  </verbus:altMessages>
</onMessage>
(...)
<invoke partnerLink="OilProviderPlnk" portType="tns:OilProviderService"
  operation="order" inputVariable="oOrder" outputVariable="oOrderConfirm">
  <verbus:altMessages>
    <verbus:message><accepted>true</accepted></verbus:message>
    <verbus:message><accepted>false</accepted></verbus:message>
  </verbus:altMessages>
</invoke>
```

Figura 8.4: Mensajes de entrada alternativos en el proceso de pedido de aceite de oliva.

```
goal rg1 {
    (process_oliveProcess_lc__.state=completed
     | process_oliveProcess_lc__.state=cancelled
     | process_oliveProcess_lc__.state=compensated)
}
prereq rg2_sequence_act_1_begin (sequence_act_1, begin) {
    (process_oliveProcess_lc__.state!=cancelled
     & process_oliveProcess_lc__.state!=compensated
     & process_oliveProcess_lc__.state!=completed)
}
prereq rg2_main_act_4_begin (main_act_4, begin) {
    (process_oliveProcess_lc__.state!=cancelled
     & process_oliveProcess_lc__.state!=compensated
     & process_oliveProcess_lc__.state!=completed)
}
...
goal rg3 {
    ((sequence_act_1_lc__.state=completed
      | sequence_act_1_lc__.state=not_started
      | sequence_act_1_lc__.state=cancelled)
     & (prepare_neg_response_act_2_lc__.state=completed
        | prepare_neg_response_act_2_lc__.state=not_started
        | prepare_neg_response_act_2_lc__.state=cancelled)
     & (response_neg_act_3_lc__.state=completed ...)
     ...)
}
exeall rg5;
```

Figura 8.5: Requisitos generales aplicados al proceso ejemplo.

```
<verbus:property type="goal" name="response_sent"
  expression="verbus:activityState('response') = 'completed'
              or verbus:activityState('response_neg') = 'completed'" />
<verbus:property type="invariant" name="deadline"
  expression="bpws:getVariableData('response', 'waitingDays') &lt;=
             bpws:getVariableData('request', 'request', 'deadline')" />
<verbus:property type="invariant" name="acceptable_price"
  expression="bpws:getVariableData('response', 'price') &lt;=
             bpws:getVariableData('request', 'request', 'maxPrice')
             or
             bpws:getVariableData('response', 'waitingDays') =
             bpws:getVariableData('request', 'request', 'deadline')" />
```

```
goal response_sent {
    (response_act_19_lc__.state=completed
     | (response_neg_act_3_lc__.state=completed))
}
invariant deadline {
    (response.waitingDays <= request.request__deadline)
}
invariant acceptable_price {
    ((response.price <= request.request__maxPrice)
     | (response.waitingDays=request.request__deadline))
}
```

Figura 8.6: Requisitos específicos para el proceso de pedidos de aceite de oliva.

```
<verbus:property type="ltl" name="ltl1"
  expression="verbus:tlg(
                 not(bpws:getVariableData('response', 'price') &lt;=
                   bpws:getVariableData('request', 'request', 'maxPrice')
                   and verbus:activityState('askPrice1') = 'completed'
                   and verbus:activityState('order') = 'not_started')
               or
                 (verbus:tlf(verbus:activityState('order') = 'running'))
              )"
/>
```

```
tl (ltl) {
  G( !((oPrice.price<=request.request__maxPrice)
       & (askPrice1_act_7_lc__.state=completed)
       & (order_act_16_lc__.state=not_started))
    || F ((order_act_16_lc__.state=running)))
}
```

```
#define p ((oPrice.price <= request.request__maxPrice)
           && (order_act_16_lc__.state==BPEL_lc_state_not_started)
           && (askPrice1_act_7_lc__.state==BPEL_lc_state_completed))
#define q ((order_act_16_lc__.state==BPEL_lc_state_running))

/*
 * Formula As Typed:  ! []  ( ! p  ||  <> q)
 */
never {    /* ( ! []  ( ! p  ||  <> q)) */
T0_init:
        if
        :: (! ((q)) && (p)) -> goto accept_S4
        :: (1) -> goto T0_init
        fi;
accept_S4:
        if
        :: (! ((q))) -> goto accept_S4
        fi;
}
```

Figura 8.7: Requisito de lógica temporal expresado mediante XPath, CFM y Promela.

Capítulo 9

Conclusiones y trabajos futuros

Este capítulo concluye esta tesis doctoral. En primer lugar, se resumen, a grandes rasgos, las ideas planteadas en la misma. Posteriormente, se exponen las principales contribuciones realizadas. Finalmente, se proponen posibles líneas futuras de investigación planteables a partir de los resultados obtenidos en esta tesis.

9.1. Conclusiones

En esta tesis se ha propuesto una arquitectura abierta, modular y extensible para la verificación de procesos de negocio y composiciones de servicios Web. Para ello, se ha desarrollado un sistema formal para la capa intermedia de dicha arquitectura, que permite desacoplar los lenguajes de definición de procesos de las herramientas de verificación.

El conjunto de la arquitectura y el sistema formal constituyen una solución al problema de la verificación de procesos que resuelve los principales problemas identificados en el estudio realizado de otros trabajos realizados previamente en este campo.

Con el objetivo principal de validar la solución propuesta, se han llevado a cabo varias tareas: análisis basado en patrones, integración de BPEL4WS, integración de Spin y NuSMV, desarrollo de un prototipo y análisis de un caso de estudio.

Por una parte, se ha analizado la adecuación y la potencia expresiva del *Modelo Formal Común* para la representación de procesos de negocio, aplicando los *patrones de workflow*. Los resultados obtenidos, y su comparación con los de otros formalismos y lenguajes de definición, resultan claramente positivos.

Por otra parte, se ha mostrado cómo integrar un lenguaje de definición de procesos complejo, BPEL4WS, en la arquitectura propuesta. Pese a la existencia de alguna limitación que impide la transformación de un pequeño subconjunto de procesos BPEL4WS, los resultados son, globalmente, satisfactorios. La integración resulta, en general, elegante, aunque compleja en la solución a algunas funcionalidades presentes en BPEL4WS, como los mecanismos de manejo de fallos y de cancelación. En la mayoría de los casos, la transformación puede ser llevada a cabo de forma automática o casi automática. Los resultados de este trabajo contribuyen a corroborar la adecuación y expresividad del modelo formal común para representar procesos de negocio.

Adicionalmente, se han integrado con éxito dos herramientas de *model checking* de uso extendido actualmente: Spin y NuSMV. La transformación de definiciones de procesos CFM

a los lenguajes de entrada de estas herramientas resulta realmente sencilla, lo cual corrobora la validez de este formalismo desde el punto de vista de la verificación.

El desarrollo de un primer prototipo, que implementa prácticamente toda la funcionalidad necesaria para realizar las transformaciones BPEL4WS–CFM, CFM–Spin y CFM–NuSMV, ha sido una gran ayuda para desarrollar, probar, detectar errores y mejorar las distintas propuestas realizadas en la tesis. Por otra parte, demuestra la viabilidad de la arquitectura propuesta.

Por último, se ha propuesto un caso de estudio que muestra la aplicación de esta arquitectura para la verificación de requisitos funcionales de un proceso BPEL4WS sencillo, utilizando Spin y el prototipo desarrollado. Las conclusiones extraídas de este caso de estudio son también satisfactorias, y muestran la utilidad de la solución propuesta.

9.2. Principales contribuciones

En esta tesis se han realizado varias contribuciones en el ámbito de la definición de requisitos funcionales de procesos de negocio y composiciones de servicios Web. A continuación se enumeran las principales:

1. Estudio de los trabajos más relevantes en el campo de la verificación de requisitos funcionales de procesos de negocio y composiciones de servicios Web.

2. Propuesta de una arquitectura de verificación abierta, modular y extensible para procesos de negocio y composiciones de servicios Web.

3. Propuesta de un sistema formal para la representación de procesos de negocio y composiciones de servicios Web, desde el punto de vista de las perspectivas de control del flujo y datos.

4. Identificación de tipos de requisitos útiles en la definición de especificaciones de procesos de negocio y composiciones de servicios Web.

5. Aplicación de dos herramientas de *model checking* a la verificación de procesos de negocio y composiciones de servicios Web.

6. Propuesta de un mecanismo de transformación de procesos BPEL4WS a modelos verificables.

7. Propuesta de un mecanismo para la definición de especificaciones de procesos BPEL4WS.

En los próximos apartados se resume brevemente cada una de las contribuciones mencionadas.

9.2.1. Estudio de los trabajos más relevantes en el campo de la verificación de requisitos funcionales de procesos de negocio y composiciones de servicios Web

En el capítulo 2 se identifican los trabajos más importantes que se han desarrollado hasta ahora en este ámbito, y se resumen sus principales puntos fuertes y débiles. Como resultado de este estudio, se identifica que estos trabajos tienen, en general, tres carencias importantes.

Por una parte, *no modelan adecuadamente la perspectiva de datos* de los procesos de negocio. Esto limita su aplicabilidad a lenguajes que sí modelan esta perspectiva, como es el caso de BPEL4WS.

Por otra parte, en general, proporcionan una *flexibilidad reducida para definir requisitos*. La libertad del diseñador para expresar los requisitos se ve limitada por un conjunto reducido de tipos de requisitos que cada uno de estos trabajos permite definir. Esto afecta, por tanto, a la potencia de la verificación en sí misma.

Finalmente, estas soluciones están fuertemente ligadas, en general, a lenguajes de definición de procesos y herramientas de verificación específicos. La posibilidad de integrar lenguajes diversos permitiría ampliar el ámbito de uso y la potencia de las soluciones propuestas. La posibilidad de integrar distintas herramientas de verificación permitiría que se pudiese seleccionar la herramienta más adecuada para verificar cada tipo de requisito o tipo de proceso, ampliando así el conjunto de tipos de requisitos verificables, y permitiendo seleccionar la herramienta más eficiente para cada uno de ellos.

9.2.2. Propuesta de una arquitectura de verificación abierta, modular y extensible para procesos de negocio y composiciones de servicios Web

En el capítulo 3 de esta tesis se ha propuesto una arquitectura de verificación compuesta de tres capas: capa de lenguajes de definición, capa del modelo formal común y capa de herramientas de verificación. La pieza clave de la arquitectura es la capa del modelo formal común, que desacopla los lenguajes de definición de las herramientas de verificación.

Esta arquitectura es abierta (permite a integración de diversos lenguajes de definición y herramientas de verificación), modular (cada lenguaje de definición o herramienta de verificación constituye un módulo, capaz de interactuar con el resto de módulos) y extensible (se pueden añadir nuevos módulos a la arquitectura de tal forma que, una vez introducidos, puedan interactuar con los módulos previamente existentes sin necesidad de modificar ninguno de estos).

Una de las ventajas de la arquitectura está en que la utilización de un formalismo intermedio reduce considerablemente, en un entorno como el actual de múltiples lenguajes de definición de procesos y herramientas de verificación, el número de herramientas de transformación necesarias para que a cada lenguaje de definición se le pueda aplicar cualquiera de las herramientas de verificación.

9.2.3. Propuesta de un sistema formal para la representación de procesos de negocio y composiciones de servicios Web, desde el punto de vista de las perspectivas de control del flujo y datos

En el capítulo 4 se ha definido un sistema formal, llamado CFM, para la representación de procesos de negocio y composiciones de servicios Web, con el objetivo de facilitar la verificación de requisitos funcionales de los mismos.

Este sistema formal se construye sobre el concepto de *diagramas estado–transición etiquetados*, pero se le dota de un conjunto de abstracciones y una notación que lo acerquen más a los conceptos manejados habitualmente en el campo del modelado de procesos de negocio basado en actividad.

Con ello se consigue alcanzar un nivel de compromiso aceptable entre el nivel de formalismo del sistema (y su adecuación a la realización de verificaciones), y su cercanía a los conceptos manejados en el modelado de procesos de negocio.

Una de las principales aportaciones de este formalismo con respecto a otros utilizados previamente para el modelado y verificación de procesos de negocio está en su mecanismo de modelado explícito de la perspectiva de datos de los procesos.

Adicionalmente, con el objetivo de facilitar y homogeneizar el diseño e implementación de compiladores de distintos lenguajes de definición de procesos, se proponen unas *líneas generales de representación*. En ellas se propone una solución a las estructuras más habituales de los lenguajes de definición basados en actividad.

En el capítulo 5 se han analizado la adecuación y potencia expresiva de CFM para la representación de procesos de negocio. Los resultados son satisfactorios y superiores a los de casi cualquier otro formalismo y herramienta, aunque se observa una limitación en la capacidad para representar múltiples instancias de actividades y entidades, en especial cuando no se conoce durante la fase de diseño este número de instancias. Pese a esta limitación, se concluye que resulta adecuado para formar parte de la capa intermedia de la arquitectura propuesta.

9.2.4. Identificación de tipos de requisitos útiles en la definición de especificaciones de procesos de negocio y composiciones de servicios Web

En la primera parte del capítulo 6 se presentan los resultados de un estudio de distintos tipos de requisitos que se han aplicado en trabajos relacionados a procesos de negocio. A partir de ellos, y teniendo en cuenta las características de las herramientas de *model checking*, se ha definido un conjunto de tipos de requisitos útiles para definir las especificaciones de procesos: invariantes, objetivos, pre y post–requisitos de transiciones funcionales, ejecutabilidad de transiciones funcionales y requisitos expresados mediante lógicas temporales. Cada uno de estos tipos de requisitos ha sido definido formalmente en términos de CFM.

Por otra parte, se ha identificado un conjunto de cinco requisitos que, en general, cualquier modelo de un proceso de negocio expresado mediante CFM debería cumplir. Estos requisitos se basan en el concepto de *solidez* introducido por Aalst, y están fundamentalmente orientados a comprobar la corrección, desde un punto de vista estructural, de las definiciones. Estos requisitos permiten comprobar, principalmente, la corrección de la herramienta de compilación empleada para obtener el modelo CFM.

9.2.5. Aplicación de dos herramientas de *model checking* a la verificación de procesos de negocio y composiciones de servicios Web

En la segunda parte del capítulo 6 se presentan los resultados de los trabajos llevados a cabo para integrar dos herramientas de *model checking* en la arquitectura propuesta: Spin y NuSMV.

Para cada una de ellas, se ha propuesto un mecanismo de transformación de definiciones de procesos CFM, así como sus especificaciones, a sus respectivos lenguajes de entrada. Por otra parte, para cada uno de los tipos de requisitos planteados, se indica si son verificables por cada una de las herramientas. En caso afirmativo, se indica cómo.

Dadas las características de la arquitectura, ambas herramientas están automáticamente disponibles para procesos definidos en cualquier lenguaje de definición que se integre en la misma.

9.2.6. Propuesta de un mecanismo de transformación de procesos BPEL4WS a modelos verificables

El lenguaje BPEL4WS es, actualmente, el lenguaje de definición de composiciones de servicios Web más importante. En el capítulo 7 se ha propuesto una semántica para este lenguaje en términos del formalismo CFM.

Esta semántica permite transformar de forma automática definiciones de procesos BPEL4WS a definiciones CFM, equivalentes desde el punto de vista de la verificación de requisitos. De esta forma, es posibles realizar verificaciones de estos procesos con cualquiera de las herramientas de verificación integradas en la arquitectura propuesta.

Tal y como se explica en el capítulo correspondiente, las limitaciones en el formalismo CFM hacen que un pequeño subconjunto de los procesos BPEL4WS (principalmente aquellos que aniden *scopes* compensables dentro de bucles *while*) no sea verificable en su totalidad. En dicho capítulo se explica por qué.

En el capítulo 8 se analiza, a modo de caso de estudio, su aplicación a un ejemplo. Los resultados del análisis resultan satisfactorios y prueban la utilidad del sistema desarrollado.

9.2.7. Propuesta de un mecanismo para la definición de especificaciones de procesos BPEL4WS

En el apartado 7 se ha propuesto también un mecanismo para la definición de especificaciones de procesos BPEL4WS, basado en los tipos de requisitos definidos para el formalismo CFM.

Se ha seleccionado como lenguaje de expresiones XPath, por ser este el lenguaje de expresiones por defecto en BPEL4WS. Adicionalmente, utilizando el mecanismo de extensibilidad de funciones de XPath, se han definido varias funciones adicionales que permiten aumentar considerablemente la potencia expresiva de los requisitos.

Por otra parte, se define un mecanismo de transformación automática de estos requisitos a requisitos expresados en términos de la representación CFM de los procesos.

9.3. Trabajos futuros

En esta tesis se han desarrollado los componentes principales de la arquitectura de verificación de procesos de negocio. A partir de los resultados obtenidos, se pueden abrir varias líneas de investigación. Estas posibles líneas incluyen temas como la integración de otros lenguajes de definición y herramientas de verificación, soporte a procesos concurrentes que se comunican, análisis de rendimiento, técnicas de reducción de complejidad de las definiciones CFM e implementación de un entorno integrado de verificación.

9.3.1. Integración de otros lenguajes de definición y herramientas de verificación

En esta tesis se ha desarrollado la integración de BPEL4WS, Spin y NuSMV en la arquitectura.

Para que el entorno sea realmente útil, y ampliar su ámbito de utilización, es necesario comenzar a integrar en él otros lenguajes de definición de procesos de uso extendido actualmente.

Por otra parte, aunque se ha comprobado que la herramienta Spin es muy útil y flexible para la verificación de procesos, otras herramientas de verificación pueden permitir la verificación de otros tipos de requisitos distintos, o incluso mejorar las prestaciones de Spin en la verificación de alguno de los tipos planteados. Puede resultar especialmente valiosa la integración del entorno IF [20], dado que en dicho entorno se encuentran integradas, también en base a un formalismo intermedio, varias herramientas de verificación.

9.3.2. Verificación de procesos de negocio concurrentes que se comunican

Los procesos de negocio se caracterizan por su interacción con entidades externas. La solución que se ha propuesto en esta tesis sólo permite realizar un modelado abstracto y simplificado del comportamiento de las entidades externas con las que se comunica el proceso.

En ocasiones, varios procesos de negocio deben ser capaces de interactuar entre sí con, posiblemente, múltiples puntos de sincronización basados en el intercambio de mensajes. Esto es habitual, sobre todo, en un entorno de colaboración entre varias organizaciones.

Los procesos que describen el comportamiento de estas entidades externas no tienen por qué ser, necesariamente, los procesos reales ejecutados por dichas entidades. Normalmente, estas entidades publican únicamente los procesos que describen su comportamiento público, o visible desde el exterior, sin revelar su implementación interna concreta.

Una línea de trabajo futuro consiste en ampliar el *Modelo Formal Común* para dar soporte a la verificación conjunta de varios procesos que se ejecuten en paralelo y se comuniquen mediante el intercambio de mensajes.

En esta línea, hemos realizado un trabajo de investigación en el marco del programa de Doctorado, titulado "Modelo formal para la verificación de procesos de workflow entre organizaciones". En él se propone la integración en el modelo CFM de un sistema de colas de mensajes asíncronos tipados, punto a punto, con capacidad para un mensaje. Indicamos cómo implementar el canal mediante atributos CFM, y cómo realizar las operaciones de escritura y lectura de mensajes.

A pesar de ello, se incluye como trabajo futuro porque todavía no ha sido completamente formalizado, analizado e integrado en el modelo presentado en esta tesis. Sería especialmente interesante integrar este mecanismo de tal forma que aprovechase el soporte específico de algunas herramientas de verificación, como Spin, para el modelado de procesos concurrentes y colas, pero permitiendo una verificación convencional en aquellas herramientas que no dispongan de dicho soporte.

9.3.3. Mejora de CFM para que admita instanciación múltiple de atributos

En varios capítulos de esta tesis se ha comprobado que la principal limitación de CFM está en la imposibilidad de crear múltiples instancias de un mismo atributo dinámicamente. Es la causa de que no resulte adecuado para varios patrones de *workflow* y de las limitaciones en los mecanismos de manejo de compensación y de eventos de BPEL4WS.

Una posible línea de investigación consiste en estudiar alternativas que permitan enriquecer el formalismo CFM de tal forma que permita esta instanciación múltiple. Uno de los retos que plantea está en conseguir mantener la sencillez del formalismo, que permite que sea verificado con una amplia variedad de herramientas de verificación.

9.3.4. Análisis de rendimiento de las verificaciones

Aunque se han realizado verificaciones de prueba de distintos procesos, no se ha planteado un estudio formal y sistemático acerca del rendimiento y consumo de recursos de las verificaciones. Este análisis puede resultar útil por varios motivos:

- Contribuirá a identificar qué herramientas de verificación resultan más adecuadas para distintos tipos de procesos y de requisitos.
- Permitirá conocer límites a la aplicabilidad práctica del entorno a definiciones de procesos, en términos de su complejidad.
- Contribuirá a escoger las técnicas de reducción de la complejidad de las verificaciones más adecuadas.
- Permitirá comparar el rendimiento del entorno propuesto en esta tesis con el de otros trabajos relacionados.

Por ejemplo, se ha comprobado informalmente que NuSMV, con la metodología de transformación planteada en esta tesis, no resulta adecuado para verificar procesos en que se definan varias variables de tipo entero, dado el tiempo y tamaño de memoria necesarios para construir los OBDDs. Aplicando técnicas de *model checking acotado* permite realizar estas verificaciones, pero hasta una profundidad que, para muchas definiciones, es excesivamente reducida. Sin embargo, Spin se comporta perfectamente con los mismos procesos. Un estudio sistemático permitirá corroborar esta conclusión, averiguar posibles causas e incluso orientar la búsqueda de soluciones para mejorar el mecanismo de representación de procesos y requisitos CFM mediante NuSMV.

9.3.5. Desarrollo de técnicas de reducción de complejidad de definiciones CFM

Las definiciones de procesos CFM, principalmente por el hecho de ser generadas de forma automática, podrían resultar, en ocasiones, demasiado complejas para ser verificables, o necesitar demasiados recursos de verificación. Por ello, conviene disponer de algoritmos que reduzcan la complejidad de las verificaciones.

Para cumplir este objetivo, proponemos distintas líneas de trabajo complementarias:

- *Optimización de las metodologías de transformación*: un diseño correcto de una metodología de transformación de definiciones de capa 3 a capa 2 es clave para reducir el espacio de estados alcanzable por el modelo CFM. Una metodología no eficiente puede generar caminos alternativos inútiles desde el punto de vista de la verificación, pero que añaden una considerable complejidad a la misma.

- *Simplificación de las definiciones generadas por transformación automática*: las definiciones de procesos CFM se obtienen de la aplicación de herramientas de transformación automáticas. Se pueden aplicar técnicas de simplificación, también automáticas, a dichas definiciones. Se podría construir una definición equivalente, pero más simple, por ejemplo, eliminando variables, posibles valores de variables, transiciones, etc. que, tras un análisis estático, se determine que no afectan al proceso ni a los requisitos. Por otra parte, la definición de las transiciones funcionales puede tener, en ocasiones, condiciones redundantes, que pueden ser eliminadas. Este tipo de simplificaciones puede reducir considerablemente el tamaño de los OBDDs si se aplica *model checking simbólico*.

- *Aplicación de técnicas de abstracción a nivel de CFM*: la *abstracción* es una de las técnicas más potentes para la simplificación de modelos. En el apartado 2.1.6 se introduce brevemente alguno de los mecanismos de abstracción empleados actualmente. Concretamente, resultaría interesante intentar adaptar alguno de los resultados presentados en [57] y [92] al modelo CFM.

- *Aplicación de técnicas de simetría a nivel de CFM*: tal y como se ha presentado en el apartado 2.1.6, las técnicas de simetría explotan el hecho de que, en algunos casos, el orden relativo de ocurrencia de hechos de distintos hilos de ejecución paralelos no afecta a los requisitos a verificar. Dada la estructura de las definiciones de procesos CFM, aparece una tendencia importante a la ocurrencia de simetrías con respecto al orden de ejecución de transiciones funcionales de actividades concurrentes. Resultaría interesante explotar estas técnicas a nivel de CFM.

En este apartado se han presentado sólo algunas de las técnicas que, *a priori*, resultan más prometedoras para reducir la complejidad de las verificaciones. Esta lista no es exhaustiva, y cabe la experimentación con otros tipos de técnicas.

9.3.6. Representación de *scopes* serializables de BPEL4WS mediante CFM

En el capítulo 7 se ha propuesto una representación de BPEL4WS mediante CFM que no da soporte a *scopes* serializables. Una línea de trabajo futuro consiste en implementar, mediante cerrojos, este sistema.

9.3.7. Desarrollo de un entorno de verificación completo y funcional

En esta tesis se han desarrollado los elementos básicos de la arquitectura de verificación. Para que realmente sea usable por los usuarios, requiere, sin embargo, el desarrollo de una interfaz potente, y su integración en herramientas de diseño de procesos.

Un aspecto fundamental para mejorar la usabilidad del entorno es que el usuario no necesite conocer el sistema formal común ni las herramientas de verificación empleadas. Debe ver únicamente el proceso y su verificación desde el punto de vista del lenguaje de definición de procesos que esté usando.

Con este fin, se plantea el desarrollo de los siguientes elementos:

- *Sistema de conversión inversa de los resultados de la verificación*: el usuario expresa sus requisitos mediante un lenguaje de alto nivel, lo más cercano posible al propio lenguaje de definición de procesos que utiliza. En esta tesis se plantea la transformación automática de estos requisitos, junto con la definición del proceso, a requisitos CFM, posteriormente verificables en las herramientas de verificación utilizadas. Los resultados de la verificación deben ser transformados en el sentido inverso, también en dos etapas, de tal forma que al usuario se le presenten en términos de sus requisitos, haciendo referencia a la definición original del proceso. Principalmente, esto requiere que las trazas de la verificación generadas por las herramientas, relativas a líneas y elementos del modelo verificado, sean convertidas para que hagan referencia únicamente a elementos de la definición original del proceso.

- *Sistema de simulación interactiva y animación de trazas*: cuando el usuario obtiene resultados de la verificación, debe disponer de herramientas adecuadas para analizarlos y detectar sus causas originales. Una herramienta básica es un *simulador interactivo*, que le permita explorar posibles evoluciones del proceso, valores de variables, etc. Este simulador debe permitir, por otra parte, la animación de las trazas de contraejemplos devueltos por las herramientas de verificación. El núcleo del simulador debe residir en la capa CFM, con interfaz orientada a eventos. La interfaz de usuario, específica para cada lenguaje de definición, se encargaría de adaptar los eventos en su comunicación con el núcleo.

- *Desarrollo de una interfaz para la programación de aplicaciones (API) para las herramientas de verificación*: las herramientas de verificación suelen estar diseñadas para interactuar directamente con el usuario final, o con *programas de script*. Para integrar estas herramientas en un entorno de verificación, es necesario que proporcionen una API que permita al entorno interactuar con ellas. Actualmente se está desarrollando, en un proyecto de fin de carrera, una API Java para Spin.

- *Integración en herramientas de diseño de procesos*: el entorno de verificación debe estar, idealmente, integrado con las herramientas de diseño de procesos. Ante la dificultad actual para encontrar entornos de diseño potentes de código libre para BPEL4WS, se está desarrollando uno mediante dos proyectos de fin de carrera, con la intención de que sea integrable en el entorno *Eclipse*[1] como *plugin*. Dentro de este entorno de

[1] Eclipse es un entorno integrado de desarrollo con licencia de código libre. Está desarrollado principalmente en Java. Destaca por la potencia y popularidad que le confiere su arquitectura modular basada en *plugins*, que hace de él un sistema abierto y extensible. Está disponible en *http://www.eclipse.org/*.

diseño, se integrará el entorno de verificación.

Bibliografía

[1] W. Aalst. Verification of workflow nets. In *Proceedings of Application and Theory of Petri Nets*, volume 1248 of *Lecture Notes on Computer Science*, pages 407–426, Toulouse, France, 1997. Springer.

[2] W. Aalst. The Application of Petri Nets to Workflow Management. *The Journal of Circuits, Systems and Computers*, 8(1):21–66, 1998.

[3] W. Aalst. Patterns and XPDL: A Critical Evaluation of the XML Process Definition Language. Technical Report FIT-TR-2003-06, Queensland University of Technology, 2003.

[4] W. Aalst. Pi calculus versus petri nets: Let us eat “humble pie” rather than further inflate the “pi hype”. Unpublished discussion paper. Available at: http://tmitwww.tm.tue.nl/research/patterns/download/pi-hype.pdf, 2004.

[5] W. Aalst, P. Barthelmess, C. Ellis, and J. Wainer. Proclets: A framework for lightweight interacting workflow processes. *International Journal of Cooperative Information Systems*, 10(4):443–482, 2001.

[6] W. Aalst, M. Dumas, A. Hofstede, and P. Wohed. Pattern-Based Analysis of BPML (and WSCI). Technical Report FIT-TR-2002-05, Queensland University of Technology, 2002.

[7] W. Aalst and A. Hofstede. Workflow Patterns: On the Expressive Power of (Petri-net-based) Workflow Languages. In K. Jensen, editor, *Proceedings of the Fourth Workshop on the Practical Use of Coloured Petri Nets and CPN Tools (CPN 2002)*, pages 1–20, 2002.

[8] W. Aalst and A. Hofstede. YAWL: Yet Another Workflow Language. Technical Report FIT-TR-2003-04, Queensland University of Technology, 2003.

[9] W. Aalst, A. Hofstede, B. Kiepuszewski, and A. Barros. Workflow patterns. *Distributed and Parallel Databases*, 14(3):5–51, July 2003.

[10] W. Aalst, A. Hofstede, and M. Weske. Business process management: A survey. In *Proceedings of the International Conference on Business Process Management (BPM 2003)*, volume 2678 of *Lecture Notes in Computer Science*, pages 1–12. Springer-Verlag, 2003.

[11] J. R. Abrial. *The B-Book: Assigning Programs to Meanings*. Cambridge University Press, 1996.

[12] A. Agostini and G. de Michelis. A light workflow management system using simple process models. *Computer Supported Cooperative Work*, 9(3/4):335–363, 2000.

[13] G. Alonso, F. Casati, H. Kuno, and V. Machiraju. *Web Services. Concepts, Architectures and Applications*. Springer, 2004.

[14] T. Andrews, F. Curbera, H. Dholakia, and et al. *Business Process Execution Language for Web Services. Version 1.1 Specification*, 2003. Available at http://www-106.ibm.com/developerworks/webservices/library/ws-bpel.

[15] A. Arkin. *Business Process Modelling Language*. Business Process Management Initiative (BPMI), 2002.

[16] A. Arkin, S. Askary, S. Fordin, W. Jekeli, K. Kawaguchi, D. Orchard, S. Pogliani, K. Riemer, S. Struble, P. Takacsi-Nagy, I. Trickovic, and S. Zimeck. *Web Services Choreography Interface (WSCI) 1.0. W3C Note 8 August 2002*. World Wide Web Consortium, 2002. Available at http://www.w3.org/TR/2002/NOTE-wsci-20020808/.

[17] E. Astesiano and M. Wirsing. An introduction to ASL. In *Proceedings of IFIP WG2.1 Conf. on Program Specifications and Transformations*. North-Holland, 1986.

[18] A. Banerji, C. Bartolini, D. Beringer, V. Chopella, K. Govindarajan, A. Karp, H. Kuno, M. Lemon, G. Pogossiants, S. Sharma, and S. Williams. *Web Services Conversation Language (WSCL) 1.0. W3C Note 14 March 2002*. World Wide Web Consortium, 2002. Available at http://www.w3.org/TR/2002/NOTE-wscl10-20020314/.

[19] A. Biere, A. Cimatti, E. Clarke, and Y. Zhu. Symbolic model checking without BDDs. *Lecture Notes in Computer Science*, 1579:193–207, 1999.

[20] M. Bozga, S. Graf, and L. Mounier. IF-2.0: A validation environment for Component-Based Real-Time Systems. In *Proceedings of the 14th International Conference on Computer Aided Verification (CAV 2002)*, volume 2404 of *Lecture Notes in Computer Science*, pages 343–348, Copenhagen, Denmark, July 2002.

[21] P. Bratley, B. L. Fox, and L. E. Schrage. *A Guide to Simulation*. Springer-Verlag, 1987.

[22] A. Brogi, C. Canal, E. Pimentel, and A. Vallecillo. Formalizing web service choreographies. In *Proceedings of the First International Workshop on Web Services and Formal Methods*, Pisa, Italy, February 2004.

[23] R. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, 1986.

[24] R. E. Bryant and J. H. Kukula. Formal methods for functional verification. In A. Kuehlmann, editor, *The Best of ICCAD: 20 Years of Excellence in Computer-Aided Design*. Kluwer Academic Publishers, 2003.

[25] J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and L. J. Hwang. Symbolic model checking: 10^{20} states and beyond. *Information and Computation*, 2(98):142–170, 1992.

[26] R. Cavada, A. Cimatti, E. Olivetti, G. Keighren, M. Pistore, and M. Roveri. *NuSMV 2.2 User Manual*. Istituto per la Ricerca Scientifica e Tecnologica (IRST), Trento, Italy, 2005. Available at http://nusmv.irst.itc.it/NuSMV/userman/v22/nusmv.pdf.

[27] S. Chaki, A. Groce, and O. Strichman. Explaining abstract counterexamples. *SIGSOFT Software Engineering Notes*, 29(6):73–82, 2004.

[28] E. Christensen, F. Curbera, G. Meredith, and S. Weerawarana. *Web Services Description Language Version 1.1. W3C Note 15 March 2001*. World Wide Web Consortium, 2001. Available at http://www.w3.org/TR/2001/NOTE-wsdl-20010315.

[29] A. Cimatti, E. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani, and A. Tacchella. NuSMV Version 2: An OpenSource Tool for Symbolic Model Checking. In *Proc. International Conference on Computer-Aided Verification (CAV 2002)*, volume 2404 of *LNCS*, Copenhagen, Denmark, July 2002. Springer.

[30] E. M. Clarke, A. Biere, R. Raimi, and Y. Zhu. Bounded Model Checking Using Satisfiability Solving. *Formal Methods in System Design*, 19(1):7–34, 2001.

[31] E. M. Clarke and E. A. Emerson. Design and synthesis of synchronization skeletons using branching time temporal logic. In *Proc. of Workshop in Logic of Programs*, volume 131 of *Lecture Notes in Computer Science*. Springer, 1981.

[32] E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. MIT Press, 1999.

[33] M. Davis and H. Putnam. A computing procedure for quantification theory. *Journal of the ACM*, 7:201–215, 1960.

[34] W. Deiters and V. Gruhn. Process management in practice applying the FUNSOFT net approach to large-scale processes. *Automated Software Engineering*, 5(1):7–25, 1998.

[35] G. Dong, R. Hull, B. Kumar, J. Su, and G. Zhou. A framework for optimizing distributed workflow executions. In *Research Issues in Structures and Semistructured Database Programming; 7th International Workshop on Database Programming Languages (DBPL'99)*, volume 1949 of *Lecture Notes on Computer Science (revised version)*, 2000.

[36] M. Dumas and A. Hofstede. UML Activity Diagrams as a Workflow Specification Language. In *Proc. of the International Conference on the Unified Modeling Language (UML)*, Toronto, Canada, October 2001. Springer Verlag.

[37] H. Eertink, W. Janssen, P. O. Luttighuis, W. Teeuw, and C. Vissers. A business process design language. In J.M.Wing, J.Woodcock, and J. Davies, editors, *Proceedings of Formal Methods (FM 99), Volume I*, volume 1708 of *Lecture Notes in Computer Science*, pages 76–95. Springer, 1999.

[38] C. Ellis and G. Nutt. Modelling and enactment of workflow systems. In M. A. Marsan, editor, *Application and Theory of Petri Nets 1993*, volume 691 of *Lecture Notes in Computer Science*, pages 1–16, 1993.

[39] E. A. Emerson. *Branching Time Temporal Logic and the Design of Correct Concurrent Programs*. PhD thesis, Harvard University, 1981.

[40] E. A. Emerson and A. P. Sistla. Symmetry and model checking. *Formal Methods in System Design*, 9(1-2):105–131, 1996.

[41] R. Eshuis. *Semantics and Verification of UML Activity Diagrams for Workflow Modelling*. PhD thesis, University of Twente, 2002.

[42] R. Eshuis and R. Wieringa. A formal semantics for uml activity diagrams. Technical Report TR-CTIT-01-04, University of Twente, 2001.

[43] R. Eshuis and R. Wieringa. A real-time execution semantics for uml activity diagrams. In *Proceedings of Fundamental Approaches to Software Engineering (FASE 2001)*, number 2029 in Lecture Notes in Computer Science, pages 76–90, 2001.

[44] R. Eshuis and R. Wieringa. Verification support for workflow design with uml activity graphs. In *Proceedings of the International Conference on Software Engineering (ICSE 2002)*, pages 166–176. ACM Press, 2002.

[45] J. A. Fisteus, L. S. Fernandez, and C. D. Kloos. Formal Verification of BPEL4WS Business Collaborations. In *Proceedings of the 5th International Conference on Electronic Commerce and Web Technologies*, volume 3182 of *Lecture Notes in Computer Science*, Zaragoza, Spain, August 2004.

[46] J. A. Fisteus, L. S. Fernández, and C. D. Kloos. Applying Model Checking to BPEL4WS Business Collaborations. In *Proceedings of the 20th Annual ACM Symposium on Applied Computing (SAC '2005)*, pages 826–830, Santa Fe, New Mexico, USA, March 2005.

[47] J. A. Fisteus, C. D. Kloos, and L. S. Fernández. Verificación de Composiciones de Servicios Web: Aplicación de Model Checking a BPEL4WS. In *Actas de las IX Jornadas de Ingeniería del Software y Bases de Datos (JISBD '04)*, Málaga, Noviembre 2004.

[48] J. A. Fisteus, C. D. Kloos, and L. S. Fernández. Verificación de Composiciones de Servicios Web: Aplicación de Model Checking a BPEL4WS. *IEEE America Latina*, 3(1), Marzo 2005.

[49] J. A. Fisteus, C. D. Kloos, and A. M. López. Modelo formal para la verificación de procesos de negocio. In *Actas del III Congreso Iberoamericano de Telemática (CITA 2003)*, Montevideo, Uruguay, Octubre 2003.

[50] J. A. Fisteus, A. Marin, and C. Delgado. VERBUS: A Formal Model for Business Process Verification. In *Proceedings of the 2004 IRMA International Conference*, New Orleans, USA, May 2004.

[51] J. A. Fisteus, J. V. Román, C. D. Kloos, and L. S. Fernández. Gestión de procesos de negocio aplicada a la ingeniería web. In *III Taller sobre Ingeniería del Software Orientada al Web (Web Engineering) WebE'2003*, Alicante, España, Noviembre 2003. Publicado en línea en: http://sipl68.si.ehu.es/webe03/WebE2003-Programa.htm.

[52] D. Florescu, A. Grunhagen, and D. Kossman. XL: an XML Programming Language for Web Service Specification and Composition. In *Proceedings of the 11th International World Wide Web Conference (WWW02)*, Hawaii, USA, May 2002.

[53] R. Floyd. Assigning meanings to programs. In *Mathematical Aspects of Computer Science. From Proc. of Symp. Appl Maths., vol. 19*, pages 19–32, 1967.

[54] H. Foster, S. Uchitel, J. Magee, and J. Kramer. Model–based Verification of Web Service Compositions. In *Proceedings of the 18th IEEE International Conference on Automated Software Engineering*, October 2003.

[55] H. Foster, S. Uchitel, J. Magee, and J. Kramer. Compatibility verification for web service choreography. In *Proceedings of the International Conference on Web Services*, San Diego, USA, July 2004.

[56] K. Futatsugi, J. Goguen, J.-P. Jounnaud, and J. Mesguer. Principles of OBJ. In *Proceedings of ACM Sysmposium on Principles of Programming Languages*, pages 52–66, 1985.

[57] M. M. Gallardo, P. Merino, and E. Pimentel. A generalized semantics of promela for abstract model checking. *Formal Aspects of Computing*, 16(3):166–193, 2004.

[58] H. Garcia-Molina and K. Salem. Sagas. In *Proceedings of the Association for Computing Machinery Special Interest Group on Management of Data (SIGMOD)*, San Francisco, USA, December 1987.

[59] M.-C. Gaudel. Structuring and Modularizing Algebraic Specifications: the PLUSS specification language, evolutions and perspectives. In *Proceedings of STAS'92*, volume 557 of *Lecture Notes in Computer Science*, pages 3–18. Springer, 1992.

[60] D. Georgakopoulos, M. F. Hornick, and A. P. Sheth. An overview of workflow management: From process modeling to workflow automation infrastructure. *Distributed and Parallel Databases*, 3(2):119–153, 1995.

[61] M. J. Gordon. Hol: A proof generating system for higher-order logic. In G. Birtwistle and P. A. Subrahmanyam, editors, *VLSI Specification, Verification and Synthesis*, pages 73–128. Kluwer Academic Publishers, 1988.

[62] D. Gross. *Fundamentals of Queueing Theory*. John Wiley & Sons, 3rd edition, 1998.

[63] O. M. Group. *OMG Unified Modeling Language Specification*, 2003. Version 1.5 (formal/03-03-01).

[64] J. V. Guttag and J. J. Horning. *LARCH: Languages and Tools for Formal Specification*. Springer-Verlag, 1993.

[65] J. J. Halliday, S. K. Shrivastava, and S. M. Wheater. Flexible Workflow Management in the OPENflow System. In *Proceedings of the 4th International Enterprise Distributed Object Computing Conference (EDOC 2001)*, pages 82–92. IEEE Computer Society, september 2001.

[66] D. Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8(3):231–274, 1987.

[67] D. Harel, H. Lachover, A. Naamad, A. Pnueli, M. Politi, R. Sherman, A. Shtull-Trauring, and M. Trakhtenbrot. STATEMATE: A Working Environment for the Development of Complex Reactive Systems. *IEEE Transactions on Software Engineering*, 16(4):403–414, 1990.

[68] M. G. Hinchey and J. P. Bowen, editors. *Applications of Formal Methods*. Prentice Hall, 1995.

[69] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–583, Oct. 1969.

[70] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice Hall, 1985.

[71] D. Hollingsworth. *Workflow Management Coallition - The Workflow Reference Model*, 1995. WfMC TC00-1003 (Issue 1.1).

[72] G. Holzmann. *Design and Validation of Computer Protocols*. Prentice Hall, 1990.

[73] G. J. Holzmann. *The Spin model checker*. Addison-Wesley, 2003.

[74] M. N. Huhns and M. P. Singh. Service-oriented computing: Key concepts and principles. *IEEE Internet Computing*, 9(1):75–81, January/February 2005.

[75] S. Jablonski and C. Bussler. *Workflow Management: Modeling Concepts, Architecture and Implementation*. International Thomson Computer Press, London, UK, 1996.

[76] W. Janssen, R. Mateescu, S. Mauw, and J. Springintveld. Verifying business processes using spin. In E. Najm, A. Serhrouchni, and G. Holzmann, editors, *Procedings of the International Spin Workshop*, 1998.

[77] C. B. Jones. *Systematic Software using VDM*. Prentice Hall, 2nd edition, 1990.

[78] S. Katz and O. Grumberg. A framework for translating models and specifications. In *Proceedings of the Integrated Formal Methods Conference (IFM2002)*, Turku, Finland, May 2002.

[79] N. Kavantzas, D. Burdett, G. Ritzinger, T. Fletcher, and Y. Lafon. *Web Services Choreography Description Language Version 1.0. W3C Working Draft 17 December 2004*. World Wide Web Consortium, 2004. Available at http://www.w3.org/TR/2004/WD-ws-cdl-10-20041217/.

[80] B. Kiepuszewski. *Expressiveness and Suitability of Languages for Control Flow Modelling in Workflows*. PhD thesis, Queensland University of Technology, 2002.

[81] C. D. Kloos. *Semantics of Digital Circuits*, volume 285 of *Lecture Notes in Computer Science*. Springer–Verlag, 1987.

[82] M. Koshkina and F. Breugel. Verification of business processes for web services. Technical Report CS-2003-11, University of York (Canada), 2003.

[83] D. Kozen. Results on the propositional μ-calculus. *Theoretical Computer Science*, 27(3):333–354, 1983.

[84] L. Cabrera et al. *Web Services Atomic Transaction (WS-AtomicTransaction)*. BEA, IBM, Microsoft, November 2004. Available at ftp://www6.software.ibm.com/software/developer/library/WS-AtomicTransaction.pdf.

[85] L. Cabrera et al. *Web Services Business Activity Framework (WS-BusinessActivity)*. BEA, IBM, Microsoft, November 2004. Available at ftp://www6.software.ibm.com/software/developer/library/WS-BusinessActivity.pdf.

[86] L. Cabrera et al. *Web Services Coordination (WS-Coordination)*. BEA, IBM, Microsoft, November 2004. Available at ftp://www6.software.ibm.com/software/developer/library/WS-Coordination.pdf.

[87] A. v. Lamsweerde. Formal specification: a roadmap. In *Proceedings of the conference on The future of Software engineering*, pages 147–159. ACM Press, 2000.

[88] N. G. Leveson, M. P. Heimdahl, and H. Hildtreth. Requirements specification for process-control systems. *IEEE Transactions on Software Engineering*, 20(9):684–706, Sept. 1994.

[89] F. Leymann. Web services flow language. version 1.0. Technical report, International Business Machines Corporation (IBM), May 2001.

[90] J. Magee and J. Kramer. *Concurrency - State Models and Java Programs*. John Wiley, 1999.

[91] Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems*. Springer-Verlag, 1992.

[92] J. Martinez. *Un enfoque basado en estándares para la integración de técnicas y herramientas de ingeniería de protocolos*. PhD thesis, Universidad de Málaga, 2005.

[93] K. L. McMillan. *Symbolic Model Checking: An Approach to the State Explosion Problem*. Kluwer Academic, 1993.

[94] B. Medjahed, B. Benatallah, A. Bouguettaya, A. H. H. Ngu, and A. K. Elmagarmid. Business-to-business interactions: issues and enabling technologies. *The VLDB Journal*, 12:59–85, May 2003.

[95] Microsoft Corporation. Microsoft BizTalk Server 2002 Enterprise Edition, 2002.

[96] N. Milanovic and M. Malek. Current solutions for web service composition. *IEEE Internet Computing*, 8(6):51–59, 2004.

[97] J. Miller, D. Palaniswami, A. Sheth, K. Kochut, and H. Singh. WebWork: METEOR's Web-based Workflow Management System. *Journal of Intelligence Information Management Systems*, pages 185–215, 1997.

[98] R. Milner. *Communication and Concurrency*. Prentice Hall, 1989.

[99] R. Milner. *Communicating and Mobile Systems: the π-Calculus*. Cambridge University Press, 1999.

[100] C. Mohan, G. Alonso, R. Güntör, and M. Kamath. Exotica: A research perspective on workflow management systems. *IEEE Data Enineering Bulletin*, 18(1):19–26, 1995.

[101] T. Murata. Petri nets: Properties, analysis, and applications. *Proc. of the IEEE*, 77(4):541–580, 1989.

[102] P. Muth, J. Weissenfels, M. Gillmann, and G. Weikum. Integrating light–weight workflow management systems within existing business environments. In *Proceedings of the 15th International Conference on Data Engineering*, pages 286–293. IEEE Computer Society, march 1999.

[103] P. Muth, D. Wodtke, J. Weissenfels, G. Weikum, and A. Kotz-Dittrich. Enterprise-wide work ow management based on state and activity charts. In A. Dogac, L. Kalinichenko, T. Özsu, and A. Sheth, editors, *Workflow Management Systems and Interoperability*, NATO/ASI. Springer, 1998.

[104] S. Nakajima. Verification of web-services flows with model-checking techniques. In *Proceedings of the First International Symposium on Cyber Worlds*, pages 378–386, Tokio, Japan, November 2002. IEEE Computer Society Press.

[105] P. Naur. Proofs of algorithms by general snapshots. *BIT*, 6:310–316, 1969.

[106] A. Oberweis, R. Schätzle, W. Stucky, W. Weitz, and G. Zimmermann. INCOME/WF: A Petri net based approach to workflow management. In H. Krallmann, editor, *Wirtschaftsinformatik 97*, pages 557–580. Springer, 1997.

[107] M. A. Ould. *Business processes: modelling and analysis for re–engineering and improvement*. John Wiley & Sons, 1995.

[108] S. Owre, J. Rushby, and N. Shankar. Formal verification for fault-tolerant architectures: Prolegomena to the design of pvs. *IEEE Transactions on Software Engineering*, 21(2):107–125, Feb. 1995.

[109] D. L. Parnas and J. Madey. Functional documents for computer systems. *Science of Computer Programming*, 25:41–61, 1995.

[110] J. Pasley. How BPEL and SOA are changing Web services development. *IEEE Internet Computing*, 9(3):60–67, May-June 2005.

[111] J. Peterson. *Petri Net Theory and the Modeling of Systems*. Prentice Hall, 1981.

[112] A. Pnueli. The temporal logics of programs. In *Proceedings of the 18th IEEE Symp. on Foundations of Computer Science*, pages 46–57, 1977.

[113] J. P. Quielle and J. Sifakis. Specification and verification of concurrent systems in CESAR. In *Proceedings of the 5th International Symposium on Programming*, pages 337–350, 1982.

[114] M. Reichert and P. Dadam. ADEPTflex: Supporting Dynnamic Changes of Workflow without Loosing Control. *Journal of Intelligent Information Systems*, 10(2):93–129, 1998.

[115] W. Reisig and G. Rozenberg, editors. *Lectures on Petri nets I: Advances in Petri nets*, volume 1491 of *Lecture Notes in Computer Science*. Springer, 1998.

[116] J. Rushby. Formal methods and their role in the certification of critical systems. Technical Report CSL-95-1, Computer Science Laboratory, SRI International, 1995.

[117] K. Salimifard and M.Wright. Petri net-based modelling of work ow systems: An overview. *European Journal of Operational Research*, 134(3):218–230, 2001.

[118] A. W. Scheer. *ARIS - Business Process Modeling*. Springer, 2nd edition edition, 1999.

[119] P. Senkul, M. Kifer, and I. H. Toroslu. A logical framework for scheduling workflows under resource allocation constraints. In *Proceedings of the 28th International Conference on Very Large Data Bases*, pages 694–705, 2002.

[120] J. M. Spivey. *The Z Notation. A Reference Manual*. International Series in Computer Science. Prentice Hall, 2nd edition, 1992.

[121] H. Verbeek, T. Basten, and W. Aalst. Diagnosing workflow processes using woflan. *The Computer Journal*, 44(4):246–279, 2001.

[122] M. Weske. Formal foundation and conceptual design of dynamic adaptations in a workflow management system. In *Proceedings of the 34th Annual Hawaii International Conference on System Science (HICSS-34)*. IEEE Computer Society, 2001.

[123] S. A. White. *Business Process Modeling Notation (Version 1.0)*. Business Process Management Initiative (BPMI), May 2004.

[124] T. Winograd and R. Flores. *Understanding Computers and Cognition*. Addison–Wesley, 1987.

[125] P. Wohed, W. Aalst, M. Dumas, and A. Hofstede. Pattern-based analysis of bpel4ws. Technical Report FIT-TR-2002-04, Queensland University of Technology, 2002.

[126] Workflow Management Coallition. *Workflow Management Coallition Standard - Terminology and Glossary*, 1999. WFMC-TC-1011 (Issue 3.0).

[127] Workflow Management Coallition. *Workflow Management Coallition Workflow Standard - Workflow Process Definition Interface - XML Process Definition Language*, 2002. WFMC-TC-1025 (Version 1.0).

[128] World Wide Web Consortium. *XML Path Language (XPath) W3C Recommendation 16 November*, 1999. Available at http://www.w3.org/TR/1999/REC-xpath-19991116.

[129] World Wide Web Consortium. *Extensible Markup Language (XML) 1.0 (Second Edition)*, October 2000. W3C Recommendation.

[130] World Wide Web Consortium. *Web Services Glossary*, February 2004. W3C Working Group Note.

[131] World Wide Web Consortium. *XML Schema Part 0: Primer Second Edition. W3C Recommendation 28 October*, 2004. Available at http://www.w3.org/TR/2004/REC-xmlschema-0-20041028/.

[132] World Wide Web Consortium. *XML Schema Part 1: Structure Second Edition. W3C Recommendation 28 October*, 2004. Available at http://www.w3.org/TR/2004/REC-xmlschema-1-20041028/.

[133] World Wide Web Consortium. *XML Schema Part 2: Datatypes Second Edition. W3C Recommendation 28 October*, 2004. Available at http://www.w3.org/TR/2004/REC-xmlschema-2-20041028/.

Apéndice A

Caso de estudio: código completo

En este apéndice se muestra el código completo de los ejemplos mostrados en el caso de estudio del capítulo 8:

- En el apartado A.1 se muestra el código BPEL4WS incial del proceso, antes de introducir el modelado de mensajes alternativos mostrado en la figura 8.4.
- En el apartado A.2 se muestra el código CFM correspondiente a la conversión del proceso BPEL4WS. Las actividades *terminate_end* y *fault_cancel_end* de la actividad que representa el proceso se muestran parcialmente, dado el excesivo tamaño de sus definiciones.
- En el apartado A.3 se muestra el código Promela correspondiente a la conversión de la definición CFM del proceso. Dada su excesiva longitud, se muestra sólo la conversión de las primeras transiciones funcionales del proceso.

A.1. Proceso de pedido de aceite de oliva (diseño inicial)

```
<process name="oliveProcess"
  targetNamespace="http://www.it.uc3m.es/jaf/ns/OliveOilService"
  xmlns:tns="http://www.it.uc3m.es/jaf/ns/OliveOilService"
  xmlns:control="http://www.it.uc3m.es/jaf/ns/OliveOilServiceControl"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns:verbus="http://www.it.uc3m.es/jaf/ns/verbus-bpel/"
  xmlns="http://schemas.xmlsoap.org/ws/2003/03/business-process/">

  <partnerLinks>
    <partnerLink name="OliveOilPlnk" partnerLinkType="OliveOilLnk"
      myRole="provider" />
    <partnerLink name="OilProviderPlnk" partnerLinkType="OilProviderLnk"
      partnerRole="provider" />
    <partnerLink name="OliveOilControlPlnk"
      partnerLinkType="OliveOilControlLnk"
      myRole="provider" />
  </partnerLinks>

  <variables>
    <variable name="request" messageType="tns:OliveOilRequest" />
    <variable name="response" messageType="tns:OliveOilResponse" />
    <variable name="oPrice" messageType="tns:OilPriceResponse" />
    <variable name="oOrder" messageType="tns:OilOrder" />
    <variable name="oOrderConfirm" messageType="tns:OilConfirmation" />
    <variable name="controlCommand" messageType="control:CommandMessage" />
    <variable name="cancelled" type="xsd:boolean" />
  </variables>

  <faultHandlers>
    <catchAll>
      <sequence>
        <assign name="prepare_neg_response">
          <copy>
            <from variable="request" part="request" />
            <to variable="response" part="request" />
          </copy>
          <copy>
            <from><ordered>false</ordered></from>
            <to variable="response" part="ordered" />
          </copy>
        </assign>

        <reply name="response_neg" partnerLink="OliveOilPlnk"
          portType="tns:OliveOilService" operation="order"
                    variable="response" />
      </sequence>
    </catchAll>
  </faultHandlers>


  <sequence name="main">

    <!-- wait for a request -->
    <receive createInstance="yes" partnerLink="OliveOilPlnk"
        portType="tns:OliveOilService"
        operation="order" variable="request" />

    <!-- initialise variables -->
    <assign name="initializations">
      <copy>
        <from><waitingDays>0</waitingDays></from>
        <to variable="response" part="waitingDays"/>
```

```
63          </copy>
64          <copy>
65            <from expression="bpws:getVariableData(’request’, ’request’,
66                                                   ’quantity’)" />
67            <to variable="oOrder" part="quantity" />
68          </copy>
69          <copy>
70            <from expression="bpws:getVariableData(’request’, ’request’,
71                                                   ’customerId’)" />
72            <to variable="oOrder" part="customerId" />
73          </copy>
74          <copy>
75            <from><oilType>olive</oilType></from>
76            <to variable="oOrder" part="oilType" />
77          </copy>
78          <copy>
79            <from expression="false()" />
80            <to variable="cancelled" />
81          </copy>
82        </assign>
83
84        <!-- ask the price for the first time -->
85        <invoke name="askPrice1"
86          partnerLink="OilProviderPlnk" portType="tns:OilProviderService"
87          operation="getPrice" inputVariable="oOrder" outputVariable="oPrice" />
88
89        <!-- while the price is too high... -->
90        <while name="wait" condition="bpws:getVariableData(’response’,’waitingDays’)
91          &lt; bpws:getVariableData(’request’, ’request’, ’deadline’)
92          and bpws:getVariableData(’oPrice’, ’price’) >
93          bpws:getVariableData(’request’, ’request’, ’maxPrice’)
94          and not(bpws:getVariableData(’cancelled’))">
95
96          <!-- wait for one day or receive a cancellation message -->
97          <pick name="wait a day">
98            <onMessage partnerLink="OliveOilControlPlnk"
99              portType="OliveOilServiceControl"
100             operation="command"
101             variable="controlCommand">
102             <assign>
103               <copy>
104                 <from expression="true()" />
105                 <to variable="cancelled" />
106               </copy>
107             </assign>
108           </onMessage>
109           <onAlarm for="1D">
110             <sequence name="loop">
111
112               <!-- a new day! count it -->
113               <assign name="incdays">
114                 <copy>
115                   <from expression="bpws:getVariableData(’response’, ’waitingDays’)
116                         + 1" />
117                   <to variable="response" part="waitingDays" />
118                 </copy>
119               </assign>
120
121               <!-- ask the price again-->
122               <invoke name="askPrice2" partnerLink="OilProviderPlnk"
123                 portType="tns:OilProviderService"
124                 operation="getPrice" inputVariable="oOrder"
125                 outputVariable="oPrice" />
126             </sequence>
```

```
127         </onAlarm>
128       </pick>
129     </while>
130
131     <switch>
132       <case condition="not(bpws:getVariableData('cancelled'))">
133         <sequence>
134           <!-- order the oil -->
135           <invoke name="order"
136             partnerLink="OilProviderPlnk"
137             portType="tns:OilProviderService"
138             operation="order" inputVariable="oOrder"
139             outputVariable="oOrderConfirm" />
140
141           <!-- respond the user: tell him if the oil has been ordered or not -->
142           <assign name="prepare_response">
143             <copy>
144               <from variable="request" part="request" />
145               <to variable="response" part="request" />
146             </copy>
147             <copy>
148               <from variable="oOrderConfirm" part="accepted" />
149               <to variable="response" part="ordered" />
150             </copy>
151             <copy>
152               <from variable="oPrice" part="price" />
153               <to variable="response" part="price" />
154             </copy>
155             <copy>
156               <from expression="bpws:getVariableData('request', 'request',
157                                                       'quantity')
158                 * bpws:getVariableData('oPrice', 'price')" />
159               <to variable="response" part="totalPrice" />
160             </copy>
161           </assign>
162         </sequence>
163       </case>
164       <case condition="bpws:getVariableData('cancelled')">
165         <assign name="prepare_negative_response">
166           <copy>
167             <from variable="request" part="request" />
168             <to variable="response" part="request" />
169           </copy>
170           <copy>
171             <from expression="false()" />
172             <to variable="response" part="ordered" />
173           </copy>
174         </assign>
175       </case>
176     </switch>
177
178     <reply name="response" partnerLink="OliveOilPlnk"
179         portType="tns:OliveOilService"
180         operation="order" variable="response" />
181
182   </sequence>
183 </process>
```

A.2. Proceso de pedido de aceite de oliva (traducción a CFM)

```
1   process oliveProcess
2   {
3       enttype BPEL_lc_basic {
4           state: enum (not_started, running, completed, cancelled);
5       }
6       enttype BPEL_lc {
7           state: enum (not_started, running, completed, compensating,
8                        compensated, faulted, fault_cancelling, cancelling,
9                        terminating, cancelled);
10          fault: enum (none);
11          fault_handler: enum (none, sequence_act_1);
12          compensate: boolean;
13      }
14      enttype BPEL_lnk {
15          evaluated: boolean;
16          evaluation: boolean;
17      }
18      /* message {http://www.it.uc3m.es/jaf/ns/OliveOilService}OliveOilRequest */
19      enttype OliveOilRequest {
20          request__quantity: integer;
21          request__maxPrice: integer;
22          request__deadline: integer;
23          request__customerId: enum (_abstract__none, c102000);
24      }
25      /* message {http://www.it.uc3m.es/jaf/ns/OliveOilService}OliveOilResponse */
26      enttype OliveOilResponse {
27          totalPrice: integer;
28          ordered: boolean;
29          waitingDays: integer;
30          price: integer;
31          request__quantity: integer;
32          request__maxPrice: integer;
33          request__deadline: integer;
34          request__customerId: enum (_abstract__none, c102000);
35      }
36      /* message {http://www.it.uc3m.es/jaf/ns/OliveOilService}OilPriceResponse */
37      enttype OilPriceResponse {
38          price: integer;
39      }
40      /* message {http://www.it.uc3m.es/jaf/ns/OliveOilService}OilOrder */
41      enttype OilOrder {
42          oilType: enum (olive, soja, sunflower);
43          quantity: integer;
44          customerId: enum (_abstract__none, c102000);
45      }
46      /* message {http://www.it.uc3m.es/jaf/ns/OliveOilService}OilConfirmation */
47      enttype OilConfirmation {
48          accepted: boolean;
49      }
50      /* message {http://www.it.uc3m.es/jaf/ns/OliveOilServiceControl}
51                 CommandMessage */
52      enttype CommandMessage {
53          request: enum (cancel);
54      }
55      /* type {http://www.w3.org/2001/XMLSchema}boolean */
56      enttype Type_boolean {
57          data: boolean;
58      }
59      enttype wait_a_day_act_9_pick_sel__ {
60          activated: enum (none, command_0, onAlarm_0);
61      }
62      enttype switch_act_14_switch_sel__ {
```

```
63          selected: enum (none, case_0, case_1, otherwise);
64      }
65      entity request: OliveOilRequest;
66      entity response: OliveOilResponse;
67      entity oPrice: OilPriceResponse;
68      entity oOrder: OilOrder;
69      entity oOrderConfirm: OilConfirmation;
70      entity controlCommand: CommandMessage;
71      entity cancelled: Type_boolean;
72      entity process_oliveProcess_lc__: BPEL_lc;
73      entity sequence_act_1_lc__: BPEL_lc_basic;
74      entity prepare_neg_response_act_2_lc__: BPEL_lc_basic;
75      entity response_neg_act_3_lc__: BPEL_lc_basic;
76      entity main_act_4_lc__: BPEL_lc_basic;
77      entity receive_act_5_lc__: BPEL_lc_basic;
78      entity initializations_act_6_lc__: BPEL_lc_basic;
79      entity askPrice1_act_7_lc__: BPEL_lc;
80      entity wait_act_8_lc__: BPEL_lc_basic;
81      entity wait_a_day_act_9_lc__: BPEL_lc_basic;
82      entity wait_a_day_act_9_pick_sel__: wait_a_day_act_9_pick_sel__;
83      entity assign_act_10_lc__: BPEL_lc_basic;
84      entity loop_act_11_lc__: BPEL_lc_basic;
85      entity incdays_act_12_lc__: BPEL_lc_basic;
86      entity askPrice2_act_13_lc__: BPEL_lc;
87      entity switch_act_14_lc__: BPEL_lc_basic;
88      entity switch_act_14_switch_sel__: switch_act_14_switch_sel__;
89      entity sequence_act_15_lc__: BPEL_lc_basic;
90      entity order_act_16_lc__: BPEL_lc;
91      entity prepare_response_act_17_lc__: BPEL_lc_basic;
92      entity prepare_negative_response_act_18_lc__: BPEL_lc_basic;
93      entity response_act_19_lc__: BPEL_lc_basic;
94
95      activity process_oliveProcess_act__ {
96          transition catch_completion {
97              domain: {(process_oliveProcess_lc__.state=faulted
98                        & sequence_act_1_lc__.state=completed)}
99              action: {process_oliveProcess_lc__.state=cancelled}
100         }
101         transition fault_cancel_begin {
102             domain: {(process_oliveProcess_lc__.fault_handler!=none
103                       & process_oliveProcess_lc__.state=running)}
104             action: {process_oliveProcess_lc__.state=fault_cancelling}
105         }
106         transition fault_cancel_end {
107             domain: {(process_oliveProcess_lc__.state=fault_cancelling
108                       & (receive_act_5_lc__.state=completed
109                          | receive_act_5_lc__.state=not_started
110                          | receive_act_5_lc__.state=cancelled)
111                       & (initializations_act_6_lc__.state=completed
112                          | initializations_act_6_lc__.state=not_started)
113                       & (askPrice1_act_7_lc__.state=completed
114                          | askPrice1_act_7_lc__.state=not_started
115                          | askPrice1_act_7_lc__.state=cancelled)
116                       & (assign_act_10_lc__.state=completed
117                          | assign_act_10_lc__.state=not_started)
118                       & ...)}
119             action: {process_oliveProcess_lc__.state=faulted}
120         }
121         transition terminate_end {
122             domain: {(process_oliveProcess_lc__.state=terminating
123                       & (receive_act_5_lc__.state=completed
124                          | receive_act_5_lc__.state=not_started
125                          | receive_act_5_lc__.state=cancelled)
126                       & (initializations_act_6_lc__.state=completed
```

```
127                          | initializations_act_6_lc__.state=not_started)
128                       & (askPrice1_act_7_lc__.state=completed
129                          | askPrice1_act_7_lc__.state=not_started
130                          | askPrice1_act_7_lc__.state=cancelled)
131                       & (assign_act_10_lc__.state=completed
132                          | assign_act_10_lc__.state=not_started)
133                       & (incdays_act_12_lc__.state=completed
134                          | incdays_act_12_lc__.state=not_started)
135                       & ...}
136             action: {process_oliveProcess_lc__.state=cancelled}
137         }
138         transition complete {
139             domain: {(main_act_4_lc__.state=completed
140                       & process_oliveProcess_lc__.state=running)}
141             action: {process_oliveProcess_lc__.state=completed}
142         }
143     }
144     /* catch all activity oliveProcess */
145     activity sequence_act_1 {
146         transition begin {
147             domain: {(process_oliveProcess_lc__.state=faulted
148                       & sequence_act_1_lc__.state=not_started
149                       & process_oliveProcess_lc__.fault_handler=sequence_act_1)}
150             action: {sequence_act_1_lc__.state=running}
151         }
152         transition complete {
153             domain: {(sequence_act_1_lc__.state=running
154                       & response_neg_act_3_lc__.state=completed)}
155             action: {sequence_act_1_lc__.state=completed}
156         }
157     }
158     activity prepare_neg_response_act_2 {
159         transition assign {
160             domain: {(sequence_act_1_lc__.state=running
161                       & prepare_neg_response_act_2_lc__.state=not_started)}
162             action: {(prepare_neg_response_act_2_lc__.state=completed
163                       & (response.request__quantity=request.request__quantity
164                       & response.request__maxPrice=request.request__maxPrice
165                       & response.request__deadline=request.request__deadline
166                       & response.request__customerId=request.request__customerId)
167                       & !response.ordered)}
168         }
169     }
170     activity response_neg_act_3 {
171         transition begin {
172             domain: {(prepare_neg_response_act_2_lc__.state=completed
173                       & response_neg_act_3_lc__.state=not_started)}
174             action: {response_neg_act_3_lc__.state=running}
175         }
176         transition complete {
177             domain: {response_neg_act_3_lc__.state=running}
178             action: {response_neg_act_3_lc__.state=completed}
179         }
180     }
181     activity main_act_4 {
182         transition begin {
183             domain: {main_act_4_lc__.state=not_started}
184             action: {main_act_4_lc__.state=running}
185         }
186         transition complete {
187             domain: {(main_act_4_lc__.state=running
188                       & response_act_19_lc__.state=completed)}
189             action: {main_act_4_lc__.state=completed}
190         }
```

```
191         transition cancel {
192             domain: {(main_act_4_lc__.state=running
193                       & (process_oliveProcess_lc__.state=fault_cancelling
194                          | process_oliveProcess_lc__.state=cancelling
195                          | process_oliveProcess_lc__.state=terminating))}
196             action: {main_act_4_lc__.state=cancelled}
197         }
198     }
199     activity receive_act_5 {
200         transition begin {
201             domain: {(main_act_4_lc__.state=running
202                       & receive_act_5_lc__.state=not_started)}
203             action: {receive_act_5_lc__.state=running}
204         }
205         transition complete_0 {
206             domain: {receive_act_5_lc__.state=running}
207             action: {(receive_act_5_lc__.state=completed
208                       & process_oliveProcess_lc__.state=running
209                       & (request.request__customerId=c102000
210                          & request.request__maxPrice=3
211                          & request.request__deadline=5
212                          & request.request__quantity=1))}
213         }
214         transition cancel {
215             domain: {(receive_act_5_lc__.state=running
216                       & (process_oliveProcess_lc__.state=fault_cancelling
217                          | process_oliveProcess_lc__.state=cancelling
218                          | process_oliveProcess_lc__.state=terminating))}
219             action: {receive_act_5_lc__.state=cancelled}
220         }
221     }
222     activity initializations_act_6 {
223         transition assign {
224             domain: {(receive_act_5_lc__.state=completed
225                       & process_oliveProcess_lc__.state=running
226                       & initializations_act_6_lc__.state=not_started)}
227             action: {(initializations_act_6_lc__.state=completed
228                       & response.waitingDays=0
229                       & oOrder.quantity=request.request__quantity
230                       & oOrder.customerId=request.request__customerId
231                       & oOrder.oilType=olive
232                       & cancelled.data=false)}
233         }
234     }
235     activity askPrice1_act_7 {
236         transition throw1 {
237             domain: {askPrice1_act_7_lc__.state=running}
238             action: {(askPrice1_act_7_lc__.state=cancelled
239                       & process_oliveProcess_lc__.fault_handler=sequence_act_1)}
240         }
241         transition begin {
242             domain: {(initializations_act_6_lc__.state=completed
243                       & process_oliveProcess_lc__.state=running
244                       & askPrice1_act_7_lc__.state=not_started)}
245             action: {askPrice1_act_7_lc__.state=running}
246         }
247         transition complete_0 {
248             domain: {askPrice1_act_7_lc__.state=running}
249             action: {(askPrice1_act_7_lc__.state=completed & oPrice.price=1)}
250         }
251         transition complete_1 {
252             domain: {askPrice1_act_7_lc__.state=running}
253             action: {(askPrice1_act_7_lc__.state=completed & oPrice.price=2)}
254         }
```

```
255         transition complete_2 {
256             domain: {askPrice1_act_7_lc__.state=running}
257             action: {(askPrice1_act_7_lc__.state=completed & oPrice.price=3)}
258         }
259         transition complete_3 {
260             domain: {askPrice1_act_7_lc__.state=running}
261             action: {(askPrice1_act_7_lc__.state=completed & oPrice.price=4)}
262         }
263         transition complete_4 {
264             domain: {askPrice1_act_7_lc__.state=running}
265             action: {(askPrice1_act_7_lc__.state=completed & oPrice.price=5)}
266         }
267         transition complete_5 {
268             domain: {askPrice1_act_7_lc__.state=running}
269             action: {(askPrice1_act_7_lc__.state=completed & oPrice.price=6)}
270         }
271         transition cancel {
272             domain: {(askPrice1_act_7_lc__.state=running
273                       & (process_oliveProcess_lc__.state=fault_cancelling
274                          | process_oliveProcess_lc__.state=cancelling
275                          | process_oliveProcess_lc__.state=terminating))}
276             action: {askPrice1_act_7_lc__.state=cancelled}
277         }
278     }
279     activity wait_act_8 {
280         transition begin {
281             domain: {(askPrice1_act_7_lc__.state=completed
282                       & process_oliveProcess_lc__.state=running
283                       & wait_act_8_lc__.state=not_started)}
284             action: {(wait_act_8_lc__.state=running
285                       & assign_act_10_lc__.state=not_started
286                       & incdays_act_12_lc__.state=not_started
287                       & askPrice2_act_13_lc__.state=not_started
288                       & loop_act_11_lc__.state=not_started
289                       & wait_a_day_act_9_lc__.state=not_started)}
290         }
291         transition continue {
292             domain: {(((response.waitingDays < request.request__deadline)
293                        & ((oPrice.price > request.request__maxPrice)
294                        & !cancelled.data))
295                       & wait_act_8_lc__.state=running
296                       & wait_a_day_act_9_lc__.state=completed)}
297             action: {(assign_act_10_lc__.state=not_started
298                       & incdays_act_12_lc__.state=not_started
299                       & askPrice2_act_13_lc__.state=not_started
300                       & loop_act_11_lc__.state=not_started
301                       & wait_a_day_act_9_lc__.state=not_started)}
302         }
303         transition complete {
304             domain: {((wait_a_day_act_9_lc__.state=completed
305                        | wait_a_day_act_9_lc__.state=not_started)
306                       & !((response.waitingDays < request.request__deadline)
307                       & ((oPrice.price > request.request__maxPrice)
308                          & !cancelled.data))
309                       & wait_act_8_lc__.state=running)}
310             action: {wait_act_8_lc__.state=completed}
311         }
312         transition cancel {
313             domain: {(wait_act_8_lc__.state=running
314                       & (process_oliveProcess_lc__.state=fault_cancelling
315                          | process_oliveProcess_lc__.state=cancelling
316                          | process_oliveProcess_lc__.state=terminating))}
317             action: {wait_act_8_lc__.state=cancelled}
318         }
```

```
319     }
320     activity wait_a_day_act_9 {
321         transition begin {
322             domain: {(wait_act_8_lc__.state=running
323                       & ((response.waitingDays < request.request__deadline)
324                           & ((oPrice.price > request.request__maxPrice)
325                               & !cancelled.data))
326                       & process_oliveProcess_lc__.state=running
327                       & wait_a_day_act_9_lc__.state=not_started)}
328             action: {(wait_a_day_act_9_lc__.state=running
329                       & wait_a_day_act_9_pick_sel__.activated=none)}
330         }
331         transition begin_command_0_0 {
332             domain: {(wait_a_day_act_9_lc__.state=running
333                       & wait_a_day_act_9_pick_sel__.activated=none)}
334             action: {(wait_a_day_act_9_pick_sel__.activated=command_0
335                       & controlCommand.request=cancel)}
336         }
337         transition complete_command_0 {
338             domain: {(wait_a_day_act_9_lc__.state=running
339                       & assign_act_10_lc__.state=completed
340                       & wait_a_day_act_9_pick_sel__.activated=command_0)}
341             action: {(wait_a_day_act_9_lc__.state=completed
342                       & wait_a_day_act_9_pick_sel__.activated=none)}
343         }
344         transition begin_onAlarm_0 {
345             domain: {(wait_a_day_act_9_lc__.state=running
346                       & wait_a_day_act_9_pick_sel__.activated=none)}
347             action: {wait_a_day_act_9_pick_sel__.activated=onAlarm_0}
348         }
349         transition complete_onAlarm_0 {
350             domain: {(wait_a_day_act_9_lc__.state=running
351                       & loop_act_11_lc__.state=completed
352                       & wait_a_day_act_9_pick_sel__.activated=onAlarm_0)}
353             action: {(wait_a_day_act_9_lc__.state=completed
354                       & wait_a_day_act_9_pick_sel__.activated=none)}
355         }
356         transition cancel {
357             domain: {(wait_a_day_act_9_lc__.state=running
358                       & (process_oliveProcess_lc__.state=fault_cancelling
359                          | process_oliveProcess_lc__.state=cancelling
360                          | process_oliveProcess_lc__.state=terminating))}
361             action: {wait_a_day_act_9_lc__.state=cancelled}
362         }
363     }
364     activity assign_act_10 {
365         transition assign {
366             domain: {(wait_a_day_act_9_pick_sel__.activated=command_0
367                       & process_oliveProcess_lc__.state=running
368                       & assign_act_10_lc__.state=not_started)}
369             action: {(assign_act_10_lc__.state=completed
370                       & cancelled.data=true)}
371         }
372     }
373     activity loop_act_11 {
374         transition begin {
375             domain: {(wait_a_day_act_9_pick_sel__.activated=onAlarm_0
376                       & process_oliveProcess_lc__.state=running
377                       & loop_act_11_lc__.state=not_started)}
378             action: {loop_act_11_lc__.state=running}
379         }
380         transition complete {
381             domain: {(loop_act_11_lc__.state=running
382                       & askPrice2_act_13_lc__.state=completed)}
```

```
383             action: {loop_act_11_lc__.state=completed}
384         }
385         transition cancel {
386             domain: {(loop_act_11_lc__.state=running
387                       & (process_oliveProcess_lc__.state=fault_cancelling
388                          | process_oliveProcess_lc__.state=cancelling
389                          | process_oliveProcess_lc__.state=terminating))}
390             action: {loop_act_11_lc__.state=cancelled}
391         }
392     }
393     activity incdays_act_12 {
394         transition assign {
395             domain: {(loop_act_11_lc__.state=running
396                       & process_oliveProcess_lc__.state=running
397                       & incdays_act_12_lc__.state=not_started)}
398             action: {(incdays_act_12_lc__.state=completed
399                       & response.waitingDays=(response.waitingDays+1))}
400         }
401     }
402     activity askPrice2_act_13 {
403         transition throw1 {
404             domain: {askPrice2_act_13_lc__.state=running}
405             action: {(askPrice2_act_13_lc__.state=cancelled
406                      & process_oliveProcess_lc__.fault_handler=sequence_act_1)}
407         }
408         transition begin {
409             domain: {(incdays_act_12_lc__.state=completed
410                      & process_oliveProcess_lc__.state=running
411                      & askPrice2_act_13_lc__.state=not_started)}
412             action: {askPrice2_act_13_lc__.state=running}
413         }
414         transition complete_0 {
415             domain: {askPrice2_act_13_lc__.state=running}
416             action: {(askPrice2_act_13_lc__.state=completed & oPrice.price=1)}
417         }
418         transition complete_1 {
419             domain: {askPrice2_act_13_lc__.state=running}
420             action: {(askPrice2_act_13_lc__.state=completed & oPrice.price=2)}
421         }
422         transition complete_2 {
423             domain: {askPrice2_act_13_lc__.state=running}
424             action: {(askPrice2_act_13_lc__.state=completed & oPrice.price=3)}
425         }
426         transition complete_3 {
427             domain: {askPrice2_act_13_lc__.state=running}
428             action: {(askPrice2_act_13_lc__.state=completed & oPrice.price=4)}
429         }
430         transition complete_4 {
431             domain: {askPrice2_act_13_lc__.state=running}
432             action: {(askPrice2_act_13_lc__.state=completed & oPrice.price=5)}
433         }
434         transition complete_5 {
435             domain: {askPrice2_act_13_lc__.state=running}
436             action: {(askPrice2_act_13_lc__.state=completed & oPrice.price=6)}
437         }
438         transition cancel {
439             domain: {(askPrice2_act_13_lc__.state=running
440                       & (process_oliveProcess_lc__.state=fault_cancelling
441                          | process_oliveProcess_lc__.state=cancelling
442                          | process_oliveProcess_lc__.state=terminating))}
443             action: {askPrice2_act_13_lc__.state=cancelled}
444         }
445     }
446     activity switch_act_14 {
```

```
447         transition begin {
448             domain: {(wait_act_8_lc__.state=completed
449                       & process_oliveProcess_lc__.state=running
450                       & switch_act_14_lc__.state=not_started)}
451             action: {(switch_act_14_lc__.state=running
452                       & switch_act_14_switch_sel__.selected=none)}
453         }
454         transition begin_case_0 {
455             domain: {(switch_act_14_lc__.state=running
456                       & switch_act_14_switch_sel__.selected=none
457                       & !cancelled.data)}
458             action: {switch_act_14_switch_sel__.selected=case_0}
459         }
460         transition complete_case_0 {
461             domain: {(switch_act_14_lc__.state=running
462                       & sequence_act_15_lc__.state=completed
463                       & switch_act_14_switch_sel__.selected=case_0)}
464             action: {(switch_act_14_lc__.state=completed
465                       & switch_act_14_switch_sel__.selected=none)}
466         }
467         transition begin_case_1 {
468             domain: {(switch_act_14_lc__.state=running
469                       & switch_act_14_switch_sel__.selected=none
470                       & cancelled.data & !!cancelled.data)}
471             action: {switch_act_14_switch_sel__.selected=case_1}
472         }
473         transition complete_case_1 {
474             domain: {(switch_act_14_lc__.state=running
475                       & prepare_negative_response_act_18_lc__.state=completed
476                       & switch_act_14_switch_sel__.selected=case_1)}
477             action: {(switch_act_14_lc__.state=completed
478                       & switch_act_14_switch_sel__.selected=none)}
479         }
480         transition complete {
481             domain: {(switch_act_14_switch_sel__.selected=none
482                       & switch_act_14_lc__.state=running
483                       & !!cancelled.data & !cancelled.data)}
484             action: {switch_act_14_lc__.state=completed}
485         }
486         transition cancel {
487             domain: {(switch_act_14_lc__.state=running
488                       & (process_oliveProcess_lc__.state=fault_cancelling
489                          | process_oliveProcess_lc__.state=cancelling
490                          | process_oliveProcess_lc__.state=terminating))}
491             action: {switch_act_14_lc__.state=cancelled}
492         }
493     }
494     activity sequence_act_15 {
495         transition begin {
496             domain: {(switch_act_14_switch_sel__.selected=case_0
497                       & process_oliveProcess_lc__.state=running
498                       & sequence_act_15_lc__.state=not_started)}
499             action: {sequence_act_15_lc__.state=running}
500         }
501         transition complete {
502             domain: {(sequence_act_15_lc__.state=running
503                       & prepare_response_act_17_lc__.state=completed)}
504             action: {sequence_act_15_lc__.state=completed}
505         }
506         transition cancel {
507             domain: {(sequence_act_15_lc__.state=running
508                       & (process_oliveProcess_lc__.state=fault_cancelling
509                          | process_oliveProcess_lc__.state=cancelling
510                          | process_oliveProcess_lc__.state=terminating))}
```

```
511             action: {sequence_act_15_lc__.state=cancelled}
512         }
513     }
514     activity order_act_16 {
515         transition throw1 {
516             domain: {order_act_16_lc__.state=running}
517             action: {(order_act_16_lc__.state=cancelled
518                       & process_oliveProcess_lc__.fault_handler=sequence_act_1)}
519         }
520         transition begin {
521             domain: {(sequence_act_15_lc__.state=running
522                       & process_oliveProcess_lc__.state=running
523                       & order_act_16_lc__.state=not_started)}
524             action: {order_act_16_lc__.state=running}
525         }
526         transition complete_0 {
527             domain: {order_act_16_lc__.state=running}
528             action: {(order_act_16_lc__.state=completed
529                       & oOrderConfirm.accepted)}
530         }
531         transition complete_1 {
532             domain: {order_act_16_lc__.state=running}
533             action: {(order_act_16_lc__.state=completed
534                       & !oOrderConfirm.accepted)}
535         }
536         transition cancel {
537             domain: {(order_act_16_lc__.state=running
538                       & (process_oliveProcess_lc__.state=fault_cancelling
539                          | process_oliveProcess_lc__.state=cancelling
540                          | process_oliveProcess_lc__.state=terminating))}
541             action: {order_act_16_lc__.state=cancelled}
542         }
543     }
544     activity prepare_response_act_17 {
545         transition assign {
546             domain: {(order_act_16_lc__.state=completed
547                       & process_oliveProcess_lc__.state=running
548                       & prepare_response_act_17_lc__.state=not_started)}
549             action: {(prepare_response_act_17_lc__.state=completed
550                       & (response.request__quantity=request.request__quantity
551                          & response.request__maxPrice=request.request__maxPrice
552                          & response.request__deadline=request.request__deadline
553                          & response.request__customerId=request.request__customerId)
554                          & response.ordered=oOrderConfirm.accepted
555                          & response.price=oPrice.price
556                          & response.totalPrice=(request.request__quantity*oPrice.price))}
557         }
558     }
559     activity prepare_negative_response_act_18 {
560         transition assign {
561             domain: {(switch_act_14_switch_sel__.selected=case_1
562                       & process_oliveProcess_lc__.state=running
563                       & prepare_negative_response_act_18_lc__.state=not_started)}
564             action: {(prepare_negative_response_act_18_lc__.state=completed
565                       & (response.request__quantity=request.request__quantity
566                       & response.request__maxPrice=request.request__maxPrice
567                       & response.request__deadline=request.request__deadline
568                       & response.request__customerId=request.request__customerId)
569                       & response.ordered=false)}
570         }
571     }
572     activity response_act_19 {
573         transition begin {
574             domain: {(switch_act_14_lc__.state=completed
```

```
575                      & process_oliveProcess_lc__.state=running
576                      & response_act_19_lc__.state=not_started)}
577            action: {response_act_19_lc__.state=running}
578        }
579        transition complete {
580            domain: {response_act_19_lc__.state=running}
581            action: {response_act_19_lc__.state=completed}
582        }
583        transition cancel {
584            domain: {(response_act_19_lc__.state=running
585                      & (process_oliveProcess_lc__.state=fault_cancelling
586                         | process_oliveProcess_lc__.state=cancelling
587                         | process_oliveProcess_lc__.state=terminating))}
588            action: {response_act_19_lc__.state=cancelled}
589        }
590    }
591    invariant deadline {
592        (response.waitingDays <= request.request__deadline)
593    }
594    invariant acceptable_price {
595        ((response.price <= request.request__maxPrice)
596          | response.waitingDays=request.request__deadline)
597    }
598    goal response_sent {
599        (response_act_19_lc__.state=completed
600         | response_neg_act_3_lc__.state=completed)
601    }
602 }
```

A.3. Proceso de pedido de aceite de oliva (traducción a Promela)

```
/*
 * Automatically generated by verbus-j 0.1a7
 */

/* Constants for enumerated values */
#define BPEL_lc_basic_state_not_started 0
#define BPEL_lc_basic_state_running 1
#define BPEL_lc_basic_state_completed 2
#define BPEL_lc_basic_state_cancelled 3
#define BPEL_lc_state_not_started 0
#define BPEL_lc_state_running 1
#define BPEL_lc_state_completed 2
#define BPEL_lc_state_compensating 3
#define BPEL_lc_state_compensated 4
#define BPEL_lc_state_faulted 5
#define BPEL_lc_state_fault_cancelling 6
#define BPEL_lc_state_cancelling 7
#define BPEL_lc_state_terminating 8
#define BPEL_lc_state_cancelled 9
#define BPEL_lc_fault_none 0
#define BPEL_lc_fault_handler_none 0
#define BPEL_lc_fault_handler_sequence_act_1 1
#define OliveOilRequest_request__customerId__abstract__none 0
#define OliveOilRequest_request__customerId_c102000 1
#define OliveOilResponse_request__customerId__abstract__none 0
#define OliveOilResponse_request__customerId_c102000 1
#define OilOrder_oilType_olive 0
#define OilOrder_oilType_soja 1
#define OilOrder_oilType_sunflower 2
#define OilOrder_customerId__abstract__none 0
#define OilOrder_customerId_c102000 1
#define CommandMessage_request_cancel 0
#define wait_a_day_act_9_pick_sel___activated_none 0
#define wait_a_day_act_9_pick_sel___activated_command_0 1
#define wait_a_day_act_9_pick_sel___activated_onAlarm_0 2
#define switch_act_14_switch_sel___selected_none 0
#define switch_act_14_switch_sel___selected_case_0 1
#define switch_act_14_switch_sel___selected_case_1 2
#define switch_act_14_switch_sel___selected_otherwise 3

/* Declaration of entity types */
typedef T_BPEL_lc_basic {
    byte state = BPEL_lc_basic_state_not_started;
};
typedef T_BPEL_lc {
    byte state = BPEL_lc_state_not_started;
    bit fault = BPEL_lc_fault_none;
    bit fault_handler = BPEL_lc_fault_handler_none;
    bool compensate = 0;
};
typedef T_BPEL_lnk {
    bool evaluated = 0;
    bool evaluation = 0;
};
typedef T_OliveOilRequest {
    int request__quantity = 0;
    int request__maxPrice = 0;
    int request__deadline = 0;
    bit request__customerId = OliveOilRequest_request__customerId__abstract__none;
};
typedef T_OliveOilResponse {
    int totalPrice = 0;
```

```
63      bool ordered = 0;
64      int waitingDays = 0;
65      int price = 0;
66      int request__quantity = 0;
67      int request__maxPrice = 0;
68      int request__deadline = 0;
69      bit request__customerId = OliveOilResponse_request__customerId__abstract__none;
70  };
71  typedef T_OilPriceResponse {
72      int price = 0;
73  };
74  typedef T_OilOrder {
75      byte oilType = OilOrder_oilType_olive;
76      int quantity = 0;
77      bit customerId = OilOrder_customerId__abstract__none;
78  };
79  typedef T_OilConfirmation {
80      bool accepted = 0;
81  };
82  typedef T_CommandMessage {
83      bit request = CommandMessage_request_cancel;
84  };
85  typedef T_Type_boolean {
86      bool data = 0;
87  };
88  typedef T_wait_a_day_act_9_pick_sel__ {
89      byte activated = wait_a_day_act_9_pick_sel___activated_none;
90  };
91  typedef T_switch_act_14_switch_sel__ {
92      byte selected = switch_act_14_switch_sel___selected_none;
93  };
94
95  /* variable declarations */
96  T_OliveOilRequest request;
97  T_OliveOilResponse response;
98  T_OilPriceResponse oPrice;
99  T_OilOrder oOrder;
100 T_OilConfirmation oOrderConfirm;
101 T_CommandMessage controlCommand;
102 T_Type_boolean cancelled;
103 T_BPEL_lc process_oliveProcess_lc__;
104 T_BPEL_lc_basic sequence_act_1_lc__;
105 T_BPEL_lc_basic prepare_neg_response_act_2_lc__;
106 T_BPEL_lc_basic response_neg_act_3_lc__;
107 T_BPEL_lc_basic main_act_4_lc__;
108 T_BPEL_lc_basic receive_act_5_lc__;
109 T_BPEL_lc_basic initializations_act_6_lc__;
110 T_BPEL_lc askPrice1_act_7_lc__;
111 T_BPEL_lc_basic wait_act_8_lc__;
112 T_BPEL_lc_basic wait_a_day_act_9_lc__;
113 T_wait_a_day_act_9_pick_sel__ wait_a_day_act_9_pick_sel__;
114 T_BPEL_lc_basic assign_act_10_lc__;
115 T_BPEL_lc_basic loop_act_11_lc__;
116 T_BPEL_lc_basic incdays_act_12_lc__;
117 T_BPEL_lc askPrice2_act_13_lc__;
118 T_BPEL_lc_basic switch_act_14_lc__;
119 T_switch_act_14_switch_sel__ switch_act_14_switch_sel__;
120 T_BPEL_lc_basic sequence_act_15_lc__;
121 T_BPEL_lc order_act_16_lc__;
122 T_BPEL_lc_basic prepare_response_act_17_lc__;
123 T_BPEL_lc_basic prepare_negative_response_act_18_lc__;
124 T_BPEL_lc_basic response_act_19_lc__;
125
126 /* Process to run the specification */
```

```
127 proctype oliveProcess()
128 {
129
130
131     /* main loop */
132     do
133         :: if
134             /* transition process_oliveProcess_act__::catch_completion */
135             :: ((process_oliveProcess_lc__.state==BPEL_lc_state_faulted)
136                 && ((sequence_act_1_lc__.state==BPEL_lc_basic_state_completed)))
137                 ->
138                 process_oliveProcess_lc__.state=BPEL_lc_state_cancelled
139             /* transition process_oliveProcess_act__::fault_cancel_begin */
140             :: ((process_oliveProcess_lc__.fault_handler!=BPEL_lc_fault_handler_none)
141                 && (process_oliveProcess_lc__.state==BPEL_lc_state_running))
142                 ->
143                 process_oliveProcess_lc__.state=BPEL_lc_state_fault_cancelling
144             /* transition process_oliveProcess_act__::fault_cancel_end */
145             :: ((process_oliveProcess_lc__.state==BPEL_lc_state_fault_cancelling)
146                 && ((receive_act_5_lc__.state==BPEL_lc_basic_state_completed)
147                     || (receive_act_5_lc__.state==BPEL_lc_basic_state_not_started)
148                     || (receive_act_5_lc__.state==BPEL_lc_basic_state_cancelled))
149                 && ((initializations_act_6_lc__.state==BPEL_lc_basic_state_completed)
150                     || (initializations_act_6_lc__.state==BPEL_lc_basic_state_not_started))
151                 && ((askPrice1_act_7_lc__.state==BPEL_lc_state_completed)
152                     || (askPrice1_act_7_lc__.state==BPEL_lc_state_not_started)
153                     || (askPrice1_act_7_lc__.state==BPEL_lc_state_cancelled))
154                 && ((assign_act_10_lc__.state==BPEL_lc_basic_state_completed)
155                     || (assign_act_10_lc__.state==BPEL_lc_basic_state_not_started))
156                 && ...
157                 ->
158                 process_oliveProcess_lc__.state=BPEL_lc_state_faulted
159             /* transition process_oliveProcess_act__::terminate_end */
160             :: ((process_oliveProcess_lc__.state==BPEL_lc_state_terminating)
161                 && ((receive_act_5_lc__.state==BPEL_lc_basic_state_completed)
162                     || (receive_act_5_lc__.state==BPEL_lc_basic_state_not_started)
163                     || (receive_act_5_lc__.state==BPEL_lc_basic_state_cancelled))
164                 && ((initializations_act_6_lc__.state==BPEL_lc_basic_state_completed)
165                     || (initializations_act_6_lc__.state==BPEL_lc_basic_state_not_started))
166                 && ((askPrice1_act_7_lc__.state==BPEL_lc_state_completed)
167                     || (askPrice1_act_7_lc__.state==BPEL_lc_state_not_started)
168                     || (askPrice1_act_7_lc__.state==BPEL_lc_state_cancelled))
169                 && ((assign_act_10_lc__.state==BPEL_lc_basic_state_completed)
170                     || (assign_act_10_lc__.state==BPEL_lc_basic_state_not_started))
171                 && ...
172                 ->
173                 process_oliveProcess_lc__.state=BPEL_lc_state_cancelled
174             /* transition process_oliveProcess_act__::complete */
175             :: ((main_act_4_lc__.state==BPEL_lc_basic_state_completed)
176                 && (process_oliveProcess_lc__.state==BPEL_lc_state_running))
177                 ->
178                 process_oliveProcess_lc__.state=BPEL_lc_state_completed
179             /* transition sequence_act_1::begin */
180             :: ((process_oliveProcess_lc__.state==BPEL_lc_state_faulted)
181                 && (sequence_act_1_lc__.state==BPEL_lc_basic_state_not_started)
182                 && (process_oliveProcess_lc__.fault_handler==
183                       BPEL_lc_fault_handler_sequence_act_1))
184                 ->
185                 sequence_act_1_lc__.state=BPEL_lc_basic_state_running
186             /* transition sequence_act_1::complete */
187             :: ((sequence_act_1_lc__.state==BPEL_lc_basic_state_running)
188                 && (response_neg_act_3_lc__.state==BPEL_lc_basic_state_completed))
189                 ->
190                 sequence_act_1_lc__.state=BPEL_lc_basic_state_completed
```

```
191             /* transition prepare_neg_response_act_2::assign */
192             :: ((sequence_act_1_lc__.state==BPEL_lc_basic_state_running)
193                 && (prepare_neg_response_act_2_lc__.state==BPEL_lc_basic_state_not_started))
194                 ->
195                 atomic{prepare_neg_response_act_2_lc__.state=BPEL_lc_basic_state_completed;
196                        response.request__quantity=request.request__quantity;
197                        response.request__maxPrice=request.request__maxPrice;
198                        response.request__deadline=request.request__deadline;
199                        response.request__customerId=request.request__customerId};
200                        response.ordered=0}
201             (...)
202             (...)
203             (...)
204             /* transition response_act_19::cancel */
205             :: ((response_act_19_lc__.state==BPEL_lc_basic_state_running)
206                 && ((process_oliveProcess_lc__.state==BPEL_lc_state_fault_cancelling)
207                     || (process_oliveProcess_lc__.state==BPEL_lc_state_cancelling)
208                     || (process_oliveProcess_lc__.state==BPEL_lc_state_terminating)))
209                 ->
210                 response_act_19_lc__.state=BPEL_lc_basic_state_cancelled
211             :: else
212                 ->break
213             fi;
214         /* invariant deadline */
215         assert((response.waitingDays <= request.request__deadline));
216         /* invariant acceptable_price */
217         assert(((response.price <= request.request__maxPrice))
218                || (response.waitingDays==request.request__deadline));
219     od;
220 end:
221     assert(((response_act_19_lc__.state==BPEL_lc_basic_state_completed)
222             || (response_neg_act_3_lc__.state==BPEL_lc_basic_state_completed)))
223 }
224 
225 /* Init process */
226 init
227 {
228     run oliveProcess()
229 }
```

Apéndice B

Descripción del prototipo

Con el objetivo de validar la arquitectura propuesta en esta tesis doctoral, se ha desarrollado un prototipo del sistema. Esta implementación ha permitido, por una parte, demostrar la viabilidad, desde el punto de vista de implementación, de las propuestas de la tesis. Por otra, detectar y corregir errores en el diseño de las distintas propuestas desarrolladas. La mayor parte de los ejemplos propuestos en esta tesis han sido generados mediante este prototipo.

El prototipo desarrollado, llamado *verbus-j*[1], implementa la arquitectura propuesta, integrando el lenguaje BPEL4WS y las herramientas de verificación *Spin* y *NuSMV*. A pesar de no ser una implementación completa de la herramienta de transformación de BPEL4WS a CFM, implementa, en la versión existente en el momento de redactar esta tesis, prácticamente toda la metodología expuesta en el capítulo 7.

La aplicación se ha desarrollado en lenguaje Java, funciona en línea de comandos y consta de los siguientes módulos:

- Módulo *bpa*: implementa una estructura de datos de representación en memoria de definiciones y especificaciones CFM, así como su escritura a fichero y transformación a Promela y NuSMV.
- Módulo *bpel*: implementa una estructura de datos de representación en memoria de procesos BPEL4WS, así como su lectura y escritura en fichero.
- Módulo *bpel2bpa*: implementa el conversor de BPEL4WS a CFM, implementando la metodología propuesta en el capítulo 7.
- Módulo *util*: clases con funcionalidad útil para el resto de módulos.

Para estimar de forma aproximada la complejidad asociada a distintas funcionalidades de la arquitectura, se muestra en el cuadro B.1 la estadística de número de líneas de código asociadas a cada una de ellas. Como conclusión principal, se observa que más de la mitad del código se corresponde con la integración de BPEL4WS en la arquitectura, mientras que el código destinado a integrar las herramientas de verificación es considerablemente menor.

[1] http://www.it.uc3m.es/jaf/verbus

Módulo	Sub–módulo	Líneas	%
bpa	núcleo	5954	28
	Spin	790	4
	NuSMV	1356	6
bpel	núcleo	4629	22
bpel2bpa	núcleo	6513	31
util	núcleo	1838	9
TOTAL		21080	

Cuadro B.1: Estadística de número de líneas del prototipo *verbus-j*.