Universidad Nacional del Sur

Tesis de Doctorado en Ciencias de la Computación

Un enfoque declarativo para modelar el comportamiento en Sistemas Reactivos

Fernando Asteasuain

Septiembre 2013

Prefacio

Esta Tesis es presentada como parte de los requisitos para optar al grado académico de Doctor en Ciencias de la Computación de la Universidad Nacional del Sur y no ha sido presentada previamente para la obtención de otro título en esta universidad u otras. La misma contiene los resultados obtenidos en investigaciones llevadas a cabo en el Departamento de Ciencias e Ingeniería de la Computación de la UNS, y en el Departamento de Ciencias de la Computación de la FCEyN de la UBA durante el período comprendido entre los años 2006 y 2013 bajo la dirección del Dr. Víctor Braberman, Profesor Adjunto del Departamento de Computación de la FCEyN de la UBA, y del Dr. Pablo R. Fillottrani, Profesor Asociado del Departamento de Ciencias e Ingeniería de la Computación de la UNS.

Fernando Asteasuain

fasteasuain@dc.uba.ar

Departamento de Ciencias e Ingeniería de la Computación

Universidad Nacional del Sur

Bahía Blanca, septiembre 2013.

UNIVERSIDAD NACIONAL DEL SUR

Secretaría General de Posgrado y Educación Continua

La presente tesis ha sido aprobada el .../.../..., mereciendo la calificacion de ......(.........).

Agradecimientos

En primera instancia quisiera agradecer a mis directores Pablo Fillottrani y Víctor Braberman, por su invaluable guía durante todo el desarrollo de la presente tesis. Al excelente acompañamiento y visión académica y profesional, también quisiera destacar su apoyo desde lo emocional y humano.

También quiero mencionar a todos en el grupo de investigación LAFHIS, con quienes he compartido estos años inolvidables mañanas, tardes, noches y madrugadas. A Diego, Hernán, Esteban, Dipi, Chapa, Germán, Guido y Sebastián, por su amistad y todo lo vivido juntos estos años.

A todos los "bahienses" que también me acompañaron este tiempo. En particular a Elsa Estévez, Mercedes Vitturini, Luciano Salotto y Javier Echáiz, que siempre tuvieron una palabra de aliento.

A todo el cineclub, amigos enormísimos. A Maga, Ana, Nico y Jose y León, Guille, Manu, Darío, Marvin y Sato. Al resto del clan porteño, a Rami y Mariana y Antonia, y a Andran y Sonia, por cada encuentro compartido.

A mis viejos, Raúl y Noemí, que me bancan siempre. A mis suegros, Horacio y Mabelita, por su infinito cariño. También a mis hermanos, Andrés y Mariano, y a mi cuñada Paula. Gracias a todos por el apoyo incondicional.

A mi mujer, Carolina, por todo su amor y aguante. Gracias vida, por todo.

Abstract

Behavior needs to be understood from early stages of software development. In this context incremental and declarative modeling seems an attractive approach for closely capturing and analyzing requirements without early operational commitment. In this situation, the most widespread formalism is LTL (Linear Temporal Logic). Unfortunately, in many cases, the formal description and validation of properties results in a daunting task, even for trained people. Moreover, LTL expressive power fall short of if the practitioner want to declaratively build a system from scratch upon the specified properties.

Several attempts arise to make life easier for practitioners, in order to circumvent the complexity involved when specifying properties in LTL. The purpose of these approaches is to provide to the user an easier way to express the desired behavior. Most of them rely on specification patterns. Although patterns offer a friendlier way to express typical requirements, the user still needs to validate or modify the property. However, to perform all these validations tasks based solely on the natural language description of the patterns may be hard to achieve and many times the pattern translation into an formalism must be analyzed instead. Besides, sometimes a property needs to be slightly modified to suit actual system requirements, and this, again, suggests manipulations of the translated versions. Therefore, using patterns is not enough to entirely hide the subtleties of the underlying formalism from the user.

This suggests the need of a formal specification language easy to use, and sufficiently expressive to enable skilled and non-skilled users to use it appropriately, which constitutes the main objective of this work. To achieve this goal we present FVS (Feather Weight Visual Scenarios) and its extension ω-FVS, a declarative language, not founded on temporal logics, but on simple graphical scenarios, which aims to improve and ease the property specification process, and powerful enough to express omega-regular properties. The notation is equipped with declarative semantics based on morphisms and a tableau procedure

is given enabling the possibility of automatic analysis.

We illustrate FVS applicability by modeling all the specification patterns and we thoroughly compare FVS to other known approaches, showing that FVS specifications are better suited for validation tasks. What is more, FVS is presented as a useful aspect-oriented modeling language.

In addition, we present ω-FVS, a simple extension of FVS capable of expressing ω-regular properties. This feature is achieved by introducing a distinctive type of events, called "ghost" events, that allow to predicate behavior in a higher level of abstraction. We illustrate ω-FVS expressive power by modeling several examples, including real world protocols specifications. On the top of these features, we also developed a synthesis procedure which translates FVS specifications into Buchi automata, enabling the possibility for different sort of analysis, namely consistency checking, model checking, and oracle for runtime verification among others.

Resumen El comportamiento debe contemplarse desde etapas tempranas del desarrollo de software. En este contexto el modelado declarativo incremental constituye una opción atractiva para capturar y analizar los requerimientos con precisión sin atarse a prematuras decisiones operacionales. Bajo esta visión, el formalismo más utilizado son lógicas temporales como LTL. Sin embargo, en muchas ocasiones la especificación de propiedades en LTL es un desafío importante, aún para personas con experiencia en el formalismo. Más aún, el poder expresivo de LTL no es suficiente si se desea construir declarativamente un sistema desde cero a partir de las propiedades que describen su comportamiento.

Varias alternativas se han desarrollado para hacer más sencilla la especificación de propiedades. La mayoría de ellas se basa en la utilización de patrones de especificación. Si bien los patrones de especificación ofrecen una manera más amigable de expresar los requerimientos típicos, el usuario necesita validar o modificar la propiedad. La evidencia indica que para realizar todas estas tareas de validación no alcanza con analizar la descripción en lenguaje natural del patrón elegido, sino que debe analizarse su traducción a un lenguaje formal. Y para estos casos, el usuario no tiene otra alternativa que enfrentarse a los lenguajes formales. Luego, la utilización de patrones de especificación no logra esconder por completo los problemas de utilizar lenguajes formales.

Este contexto sugiere la necesidad de contar con un lenguaje formal de especificación

que sea sencillo de usar, y lo suficientemente expresivo para permitir a todo tipo de usuarios utilizarlo de manera intuitiva. Para lograr este objetivo se presenta FVS (Feather Weight Visual Scenarios) y su extensión ω-FVS, un lenguaje declarativo, no basado en lógicas temporales, sino en simples escenarios gráficos, cuya meta es mejorar y facilitar el proceso de especificación de propiedades, y con el suficiente poder expresivo para denotar propiedades ω-regulares. La notación cuenta con una semántica declarativa basada en morfismos y un procedimiento de tableau que permite la posibilidad de realizar análisis automático.

Se exhibe la usabilidad y aplicabilidad de FVS modelando todos los patrones de especificación, comparando las especificaciones resultantes con otras notaciones conocidas. Tal comparación permite observar el impacto de FVS en la especificación de propiedades. Asimismo, FVS también puede ser utilizado como un poderoso lenguaje de modelado orientado a aspectos.

Finalmente, la extensión ω-FVS permite lograr el suficiente poder expresivo para denotar propiedades ω-regulares. Esto se logra a través de un tipo especial de eventos, denominados eventos fantasmas, que permiten introducir niveles de abstracción en etapas de modelado. Se ilustra el poder expresivo de ω-FVS modelando varios ejemplos, incluyendo protocolos de comunicación de software. ω-FVS cuenta con un proceso de síntesis, que traduce sus especificaciones a autómatas de Buchi, permitiendo así la capacidad de llevar a cabo diferentes tipos de razonamiento automático como *consistency checking*, *model checking*, y oráculo para verificación en ejecución entre otros.

Índice general

Agradecimientos iii

Abstract iv

Lista de figuras xii

Lista de tablas xiii

1. Descripción del Problema 1

1.1. Introducción 1

1.2. Objetivos 4

1.3. Metodología 5

1.3.1. Especificación formal de Propiedades 6

1.3.2. Especificación Declarativa del comportamiento 7

1.3.3. El origen de FVS y ω-FVS 9

1.4. Resumen de Resultados 9

1.5. Estructura de la tesis 10

2. Conceptos Preliminares 11

2.1. Patrones de Especificación 11

2.2. Autómatas de Buchi 13

3. Especificación temprana de comportamiento 16

3.1. Introducción 16

3.1.1. Declaración declarativa de propiedades: dificultades actuales 18

3.2. FVS: Feather weight Visual Scenarios 20

3.2.1. Reglas FVS 22

3.2.2. Anti-Scenarios 23
3.3. Sintaxis y Semántica de FVS 24
3.3.1. Sintaxis FVS 24
3.3.2. Semántica FVS 27

4. Modelando Patrones de Especificación 30
4.1. Patrones de Especificación en FVS 30
4.1.1. Modelando alcances 31
4.1.2. Patrones de Ocurrencia 32
4.1.3. Patrones de Orden 36
4.1.4. Extensión a la especificación de patrones 39
4.1.5. Combinando las nuevas propiedades 42

5. Comparando Especificaciones 45
5.1. Comparación de los atributos de calidad 45
5.1.1. Sucinto 46
5.1.2. Comparabilidad 50
5.1.3. Complemento 52
5.1.4. Modificabilidad 55
5.2. Propuesta en acción 57

6. FVS como Lenguaje de Modelado Orientado a Aspectos 60
6.1. Motivación 60
6.2. FVS como un lenguaje de modelado orientado a aspectos 65
6.3. Modelado Incremental y Flexibilidad del modelo de pointcuts 66
6.3.1. Análisis y Resultados 68
6.4. Interferencia de Aspectos 69
6.4.1. Especificaciones Declarativas y la interferencia entre aspectos 69
6.4.2. Anti-escenarios y la interferencia entre aspectos 72
6.4.3. Modelado Localizado y la interferencia entre aspectos 74
6.4.4. Análisis y Resultados 74

7. Aumentado el poder expresivo: lenguajes ω regulares 76
7.1. ω-FVS 76

7.2. El poder expresivo de ω-FVS 80
7.2.1. Yendo más allá del poder expresivo de LTL 82
7.2.2. Ejemplo 1: Protocolo de Comunicación SMB2 83
7.2.3. Ejemplo2: Protocolo de audio Philips 86

8. Proceso de Síntesis **91**
8.1. Síntesis 91
8.1.1. Definiciones 92
8.1.2. Algoritmo de Tableau 93
8.1.3. Correctitud y Completitud del Tableau 98
8.2. Esquema punta a punta del proceso de síntesis 101

9. Validación Construcción de Autómatas **103**
9.1. Caso de Estudio 103
9.1.1. Resultados 104
9.1.2. Análisis de los resultados 106

10. Trabajo Relacionado **107**
10.1. Notaciones basadas en Escenarios 107
10.2. Aproximaciones basadas en Patrones de Especificación 109
10.3. Más allá del poder expresivo de LTL 110
10.4. Orientación a aspectos 110

11. Trabajo Futuro **112**
11.1. Sistemas Abiertos 112
11.2. Especificaciones parciales en estructuras "branching time" 113
11.3. Combinación de los enfoques 114
11.3.1. Metodología 115

12. Conclusiones **116**

A. Demostración Lema Caracterización Estados del Tableau **119**

Bibliografía **123**

Lista de figuras

3.1. Elementos Básicos en FVS 21

3.2. Escenarios FVS modelando el comportamiento de un protocolo 22

3.3. Reglas FVS 23

3.4. Un anti-escenario en FVS 23

3.5. Un ejemplo de un morfismo entre escenarios 26

3.6. Reglas FVS modelando un protocolo de comunicación 27

3.7. Ejemplo de un escenario satisfaciendo una regla 28

4.1. El patrón Respuesta modelado en FVS 31

4.2. El patrón Respuesta bajo distintos alcances 32

4.3. El patrón Ausencia en FVS 33

4.4. El patrón Existencia modelado con reglas FVS 34

4.5. El patrón 2-Existencia limitada modelado en FVS 35

4.6. El patrón Precedencia en FVS 36

4.7. Precedencia Encadenada con dos estímulos y una respuesta 37

4.8. Precedencia Encadenada con un estímulo y dos respuestas 38

4.9. El patrón Respuesta Encadenada con un estímulo y dos respuesta 38

4.10. El patrón Respuesta Encadenada con dos estímulos y una respuesta 39

4.11. Una instancia posible del patrón Encadenamiento Restingido en FVS . . . 40

4.12. Reglas FVS para cubrir la especificación extendida de patrones 42

4.13. Combinando pre y post aridad para el patrón Respuesta 43

4.14. Se agregan nuevas restricciones utilizando los conceptos de inmeadiatez y nulidad 43

5.1. El patrón respuesta modelado bajo 3 notaciones diferentes 46

5.2. Anti-escenarios para el patrón Respuesta con alcance Entre P y Q 52

5.3. Anti-escenario para el patrón Ausencia 53
5.4. Anti-escenario para el patrón Existencia 53
5.5. Anti-escenario para el patrón 2-Existencia limitada 53
5.6. Comportamiento complementario para el patrón Precedencia 53
5.7. Comportamiento complementario para el patrón Precedencia Encadenada . 54
5.8. Anti-escenarios para el patrón Respuesta Encadenada 54
5.9. Anti-escenarios para el patrón Encadenamiento Restringido 55
5.10. Dos variantes para el patrón Respuesta Encadenada 56
5.11. Una variante del patrón Encadenamiento Restringido 57
5.12. Completando el proceso de especificación de propiedades en FVS 59

6.1. Una regla FVS comportándose como un aspecto 66
6.2. Regla FVS controlando el estado de la puerta 67
6.3. Aspectos controlando cómo y cuándo se deben apagar las luces 67
6.4. Aspectos en FVS para el sistema de luces 68
6.5. Reglas FVS modelando requerimientos del ejemplo Telecom 70
6.6. El aspecto tiempo en FVS 70
6.7. El aspecto Facturador en FVS 71
6.8. Aspectos en conflicto en el sistema JukeBox 73
6.9. Anti-escenarios para el sistema JukeBox 73
6.10. Conflicto de aspectos revisitado tras análisis de anti-escenarios 74
6.11. Conflicto logging-encriptación 75

7.1. Reglas que denotan el modo caro del sistema de luces 79
7.2. Un morfismo proyección en ω-FVS 80
7.3. Un autómata de Buchi 81
7.4. Reglas ω-FVS para modelar un autómata de Buchi 82
7.5. Contando en ω-FVS 83
7.6. p debe vale en cada paso par 83
7.7. Todo pedido recibe una respuesta (1) y toda respuesta es precedida por un pedido(2). 84
7.8. Reglas para el protocolo de Ventana Deslizante 85
7.9. Evitando la inanición 86

7.10. Caracterizando el modo no caótico de acuerdo a los eventos in 88
7.11. Caracterizando los modo caótico y no caótico para los eventos out 88
7.12. Un autómata de Buchi modelando una codificación Manchester 89
7.13. Reglas que modelan el comportamiento del autómata propuesto 90
7.14. Reglas para modelar la propiedad deseada 90

8.1. Un ejemplo de una situación 93
8.2. La situación inicial evoluciona contemplando todos los casos 98
8.3. El autómata construido por el tableau para el patrón Respuesta 98
8.4. Esquema punta a punta del proceso de síntesis 102

9.1. Comparación de tamaño entre autómatas 105

Lista de tablas

5.1. Comparación del tamaño de especificaciones para el patrón Respuesta . . . 47
5.2. Crecimiento fórmulas LTL para patrones de ocurrencia 48
5.3. Crecimiento reglas FVS para patrones de ocurrencia 48
5.4. Crecimiento fórmulas LTL para los patrones de orden 49
5.5. Crecimiento FVS para patrones de orden 50

6.1. Mapeo entre conceptos de la orientación a aspectos a conceptos FVS . . . 66

9.1. Comparación de tamaño de autómatas 106

Capítulo 1

Descripción del Problema

1.1. Introducción

Existe consenso en la comunidad de la Ingeniería de Software que es fundamental entender, modelar y describir el comportamiento del software desde etapas tempranas del desarrollo. El conjunto de estas actividades se conoce como descripción temprana de comportamiento, cuyo principal objetivo es explorar y razonar sobre el comportamiento del sistema a ser desarrollado lo antes posible. En general, una descripción temprana de comportamiento incluye un conjunto de propiedades que el sistema debe satisfacer, el cual puede ser visto como una instancia previa de lo que luego serán los requerimientos del sistema [107]. Cada propiedad se enfoca en un aspecto o interacción particular del comportamiento del sistema, por lo que no es necesario especificar el comportamiento del sistema de manera completa o global.

Este conjunto de propiedades deseables constituye un trascendental parámetro de entrada para numerosas técnicas y herramientas de verificación formal de software como model checking [31] o model based testing [105, 34]. A pesar del notable crecimiento del área de verificación formal de software desde los años 70, la aplicación de sus técnicas y herramientas no constituyen una práctica común dentro del mundo industrial del desarrollo de software.

En este sentido, varios autores identifican al proceso de especificación de propiedades como uno de principales obstáculos para la transferencia de tecnología de verificación de software [47, 53, 43, 98, 32, 90, 18]. Esto se debe a que existe una enorme diferencia entre una propiedad expresada en lenguaje natural, y la propiedad expresada en el lenguaje

de especificación utilizado en la verificación formal. Muchas veces una propiedad que empleando lenguaje natural resulta extremadamente simple y fácil de comprender, se vuelve un artefacto incomprendible e inmanejable cuando es codificada en algún lenguaje de especificación formal. En otras palabras, existen serias dificultades para que el ingeniero de software pueda expresar la propiedad de interés en un lenguaje formal de especificación, como así también validar que dicha propiedad denote el comportamiento que tienen en mente. Como resultado de esto, muchos ingenieros de software tienden a evitar el uso de la verificación formal [53, 98, 18].

El poder expresivo de la notación utilizada representa otro importante obstáculo. Un lenguaje formal debe ser lo suficientemente expresivo como para poder describir y expresar todas las propiedades de interés del sistema. Poder expresivo y usabilidad son dos principios usualmente en conflicto. En ocasiones, en pos de obtener mayor expresivdad la usabilidad del lenguaje se ve disminuida, obteniéndose un lenguaje de especificación poco atractivo [92, 115, 27].

El presente trabajo está enfocado en proveer un lenguaje de especificación para describir de manera declarativa el comportamiento esperado de sistemas reactivos como un conjunto de trazas válidas, considerando al sistema desde una perspectiva de un sistema cerrado lineal. En este contexto, las lógicas temporales representan el mecanismo más empleado para especificar el comportamiento esperado de un sistema de una manera declarativa. En particular, Linear Temporal Logics (LTL) [84] es el formalismo más conocido. LTL permite describir cómo los distintos eventos del sistema se relacionan con el paso del tiempo. Sin embargo, escribir propiedades en LTL no es una tarea sencilla. Algunas propiedades son extremadamente difíciles de expresar en LTL, aún para personas con experiencia en lenguajes formales [47, 53, 43, 98, 32, 90, 18]. A pesar que las propiedades están enfocadas en reflejar sólo una posible interacción en particular del comportamiento del sistema, lograr formulas LTL que las reflejen de manera precisa puede convertirse en una actividad compleja y propensa a errores. Durante la codificación de una fórmula pueden pasarse por alto con facicilidad muchos detalles, con lo que no se logra capturar con exactitud el comportamiento deseado.

Varias aproximaciones surgieron con la intención de facilitarle al ingeniero llevar adelante el proceso de especificación de propiedades en LTL. Básicamente, estas aproximaciones ofrecen formas más simples de especificar propiedades, buscando esconder y evitar

la complejidad inherente a LTL. La mayoría busca cumplir este objetivo basándose en los patrones de especificación propuestos en [43, 42]. Sin embargo, estas aproximaciones no proveen una solución completamente satisfactoria [13, 32]. Si bien los patrones de especificación ofrecen una manera más amigable de expresar los requerimientos típicos, el usuario necesita validar la propiedad. La evidencia indica que para realizar todas estas tareas de validación no alcanza con analizar la descripción en lenguaje natural del patrón elegido, sino que debe analizarse su traducción a un lenguaje formal [53, 98, 13]. Llevar a cabo estas tareas implica la manipulación de formulas LTL complejas, que requiere de usuarios expertos para minimizar la posibilidad de introducir errores. Aún para casos sencillos en muchas ocasiones se requiere la asistencia de herramientas expuestas a operaciones costosas y sofisticadas.

El poder expresivo de LTL también ha sido cuestionado por varios autores [92, 27, 115, 85]. Estos trabajos afirman que en determinadas situaciones el poder expresivo de LTL no es suficiente para describir todas las propiedades de interés. En particular, el trabajo en [113] demostró que LTL no puede expresar todo el conjunto de las propiedades ω-regulares. Por ejemplo, LTL no puede expresar una propiedad como "*p ocurre solamente en los instantes pares*". Desde otro punto de vista, trabajos como [37, 83] expresan que el poder expresivo de LTL no es suficiente en aquellas situaciones donde se desea construir desde cero un sistema a partir de las propiedades que describen su comportamiento (síntesis). Para lograr esto último se requieren formalismos capaces de expresar propiedades ω-regulares, como los autómatas de Buchi [31], especialmente si se desea capturar declarativamente el comportamiento esperado como un conjunto de trazas válidas.

Han surgido distintos trabajos enfocados en superar esta falta de poder expresivo de las lógicas temporales, incluso presentando formalismos con el mismo poder expresivo que los autómatas de Buchi [23, 114, 108, 68, 115, 27]. Por mencionar ejemplos, algunas lógicas como [114, 108, 27] incorporar en su notación características operaciones como autómatas para aumentar así el poder expresivo. Sin embargo, en muchas de estas aproximaciones el aumento del poder expresivo implica un enorme retroceso en la usabilidad del formalismo, limitando así su impacto a aspectos téoricos [92, 115].

Dado este contexto, en la presente tesis de doctorado se presenta un nuevo entorno declarativo para especificar el comportamiento esperado de un sistema desde etapas tempranas, con el suficiente poder expresivo para expresar propiedades ω-regulares. En parti-

cular, se introduce el lenguaje declarativo FVS (Feather Weight Scenarios) y su extensión ω-FVS para manejar propiedades ω-propiedades. FVS no está basado en lógicas temporales, sino que es un lenguaje gráfico, donde el comportamiento se expresa a través de simples escenarios. FVS cuenta con una sintaxis clara y una semántica formal clara y precisa, lo que permite la posibilidad de realizar todo tipo de razonamiento automático, modelado incremental y validación intuitiva de propiedades. FVS permite una natural descripción del comportamiento desde etapas tempranas, donde el ingeniero de software puede especificar de manera simple e intuitiva las propiedades que ilustran el comportamiento esperado. Además, el poder expresivo de su extensión ω-FVS, capaz de modelar propiedades ω-regulares, es suficiente para lograr una completa especificación declarativa del comportamiento. Finalmente, toda especificación en FVS y ω-FVS puede traducirse a autómatas de Buchi, a través de un procedimiento tipo tableau encargado de la traducción. Esta última característica permite combinar especificaciones FVS y ω-FVS con otras notaciones operacionales, y lograr aumentar así la gamma de análisis automático que puede aplicarse sobre las mismas.

El resto de este primer capítulo se estructura como sigue. La siguiente sección (sección 1.2) presenta los objetivos propuestos para la presente tesis, mientras que la sección 1.3 describe la metodología empleada para conseguirlos. La sección 1.4 enuncia los resultados obtenidos como frutos de la investigación y finalmente la sección 1.5 establece la estructura general de la presente tesis.

1.2. Objetivos

El **objetivo global** de esta tesis fue elaborar un enfoque de modelado declarativo, capaz de manejar distintos niveles de abstracción, con semántica precisa y clara, para modelar el comportamiento de sistemas reactivos. Esto incluye la capacidad de especificar propiedades deseables, como también la posibilidad de expresar de manera declarativa el comportamiento esperado de un sistema. El **objetivo específico** de esta tesis fue desarrollar un lenguaje de modelado declarativo, basado en notaciones gráficas (escenarios), capaz de modelar y describir el comportamiento de sistemas reactivos. El lenguaje cuenta una semántica y sintaxis clara y precisa, con la posibilidad de realizar razonamiento automático, modelado incremental, y validación intuitiva de propiedades. En pocas palabras,

los siguientes puntos ilustran las principales características del lenguaje desarrollado:

- Lenguaje Minimal
- con semántica declarativa,
- que permita el modelado incremental,
- que cuente con operadores de refinamiento y especialización,
- que provea especificaciones flexibles,
- que maneje distintos niveles de abstracción,
- que cuente con expresividad suficiente para describir lenguajes ω-regular.

1.3. Metodología

La metodología empleada para lograr los objetivos propuestos consistió de dos grandes etapas. La primera se enfocó en el desarrollo de un lenguaje declarativo denominado FVS (Feather Weight Visual Scenarios) para llevar a cabo la **especificación formal de propiedades** en sistemas reactivos. Luego, en la segunda etapa se concibieron las extensiones necesarias para poder **especificar de manera declarativa el comportamiento de un sistema**. La especificación de propiedades puede influir en el comportamiento de un sistema, pero para poder describir por completo el comportamiento de un sistema o incluso, construirlo a partir de un conjunto de propiedades a satisfacer, es necesario aumentar el poder expresivo. Esta segunda etapa se desarrolló en dos partes, permitiendo afianzar los conceptos de manera gradual. En la primera parte se analizó la aplicabilidad de FVS como un lenguaje de modelado orientado a aspectos. La orientación a aspectos propone técnicas avanzadas de modularización, por lo que constituye un caso de estudio más que atractivo. Una de las conclusiones de esta etapa es que se necesita contar con lenguajes de especificación con un alto poder expresivo para poder describir por completo todo el comportamiento esperado del sistema. Esto motivó en parte la transición hacia el segundo paso: **aumentar el poder expresivo** de FVS. Para lograr esto se introdujo ω-FVS, una extensión a FVS para denotar lenguajes ω-regulares. ω-FVS incluye como nueva característica la posibilidad de introducir eventos de alto nivel, que no están presentes en el

vocabulario del sistema. Estos nuevos eventos, denominados "eventos fantasmas", permiten introducir distintos niveles de abstracción en el modelado del comportamiento. Con su presencia, ω-FVS permite describir aún propiedades ω-regulares, obteniendo el mismo poder expresivo que los autómatas de Buchi.

En las siguientes subsecciones describen con mayor profundidad cada una de estas etapas, y finalmente, en la sección 1.3.3 se describe el origen de los lenguajes FVS y ω-FVS.

1.3.1. Especificación formal de Propiedades

Como primera instancia se definieron atributos de calidad deseables en un lenguaje formal para una correcta especificación y validación de propiedades. Dichos atributos son: *Sucinto*, *Comparabilidad*, *Complemento* y *Modificabilidad*. *Sucinto* se refiere a qué tan conciso puede ser expresar una propiedad. Este atributo es esencial para poder hacer más sencilla la validación de propiedades. *Comparabilidad* establece que las propiedades deben ser fáciles de comparar, de distinguir, y de entender la relación entre ellas. *Complemento* se refiere a que debe ser sencillo entender las distintas situaciones que llevan a la violación de la propiedad. Esta información es de gran utilidad a la hora de validar una propiedad. Finalmente, *modificabilidad* se refiere a la habilidad para poder manipular una propiedad para poder adaptarla a nuevas contextos de aplicación. Se analizaron formalismos para la especificación de comportamiento como lógicas temporales y notaciones basadas en autómatas y se concluyó que no manejaban adecuadamente estos atributos de calidad planteados. Luego, se presentaron los lineamientos generales del lenguaje declarativo diseñado para cubrir todos los objetivos mencionados, denominado FVS (FeatherWeight Visual Scenarios). FVS, a pesar de su simpleza, es lo suficientemente poderoso como para describir todos los patrones de especificación. Por un lado, su naturaleza gráfica y visual ayuda al usuario en concentrarse únicamente en las propiedades y no tratar con las complicaciones de su formalización. Los escenarios son construidos usando una cantidad minimal de operadores simples, obteniendo así especificaciones concisas y compactas. Relaciones lógicas y semánticas pueden fácilmente deducirse directamente de los escenarios, aumentando la posibilidad de razonar sobre las propiedades. El lenguaje cuenta con la posibilidad de construir automáticamente anti-escenarios, los cuales ayudan al usuario en la especificación de propiedades examinando el comportamiento que lleva a una violación de

una propiedad. Por último, la especificación de propiedades es flexible, y puede adaptarse fácilmente a diferentes contextos de aplicación.

Para validar la aplicabilidad y usabilidad de FVS respecto de los atributos definidos, se tomó como caso de estudio los patrones de especificación propuestos en [43] y una extensión a los mismos propuesta en [98]. Los patrones de especificación se describen detallando por un lado el comportamiento capturado por el patrón, como también el alcance o la porción de la ejecución del sistema donde la propiedad debe valer. Además, se proveen traducciones de cada patrón a diversos formalismos [43]. Este caso de estudio es altamente representativo. Un estudio presentado en [43] analiza al menos 555 especificaciones de al menos 35 fuentes diferentes, demostrando que el 92 % de las propiedades podía ser descrita a través de los patrones. El caso de estudio consistió en comparar las especificaciones FVS de los patrones contra versiones en otros formalismos. En la presente investigación se focalizó en el mapeo de los patrones al formalismo LTL, siendo el más representativo de los lenguajes declarativos usados para especificación dentro de una visión lineal de ocurrencia de eventos. Adicionalmente también se realizaron comparaciones con traducciones a notaciones operacionales basadas en autómatas y notaciones basadas en el uso de lenguaje natural disciplinado o restringido mediante gramáticas o templates. Estos últimos ejemplos fueron tomados de la herramienta PROPEL [98]. Como conclusión de dicha comparación se puede establecer que las especificaciones en FVS son más concisas, más simples de comparar, analizar y modificar. Esencialmente, FVS logra manejar apropiadamente todas las tareas que involucra la validación de propiedades. Adicionalmente, como dato extra, se agregó a la especificación de cada patrón la descripción de su comportamiento complementario, el cual representa información valiosa para el usuario.

Es importante notar que todos los patrones de especificación mencionados en la presente tesis son aquellos presentados en [43]. Los mismos están disponibles de manera online en [42]. También se analizó una extensión a la especificación de patrones [98].

1.3.2. Especificación Declarativa del comportamiento

El primer paso de esta etapa fue analizar a FVS como un lenguaje de modelado orientados a aspectos, para luego incorporar nuevos elementos que permitieran aumentar el poder expresivo de FVS.

FVS como un lenguaje de modelado orientado a aspectos

Para incorporar nociones avanzadas de modularización en los trabajos [10, 12, 17] se exploró la posibilidad de definir a FVS como un lenguaje de modelado orientado a aspectos, buscando introducir modelos de *joinpoints* más flexibles. En los últimos años, la orientación a aspectos ha surgido como un enfoque interesante para tratar con la complejidad en la descripción de entidades de software. Sin embargo, algunos autores han señalado dificultades para aplicar la filosofía orientada a aspectos en notaciones operacionales [57, 58]. Muchas aproximaciones orientadas a aspectos terminan recayendo en mecanismos de composición (weaving) sintácticos, sin una semántica clara [57]. FVS ataca estos problemas brindando una mayor flexibilidad para desacoplar la interacción entre los aspectos y el sistema bajo análisis.

Luego, en [14] se exhibió cómo FVS, a través de la capacidad de generar sobre el comportamiento complementario (anti-escenarios) y otros mecanismos de validación de propiedades, puede ser extremadamente útil para analizar y explorar el problema de interacción e interferencia de aspectos, identificado como uno de los principales problemas de la orientación a aspectos en etapas de modelado.

ω-FVS: aumentando el poder expresivo

Habiendo establecido la necesidad de contar con lenguajes formales declarativos con suficiente poder expresivo como para modelar propiedades ω-regulares, se buscó aumentar el poder expresivo de FVS. Para tal fin se incorporaron "eventos fantasmas", los cuales permiten introducir niveles de abstracción. Para ilustrar el nuevo poder expresivo del lenguaje, se modelaron relevantes casos de estudio, enfocados mayormente en protocolos de comunicación. Los ejemplos tratados incluyen el protocolo de comunicación Phillips [26] y el protocolo de Microsoft SMB2 [109].

Una importante contribución de esta etapa es la traducción de las especificaciones de FVS en autómatas de Buchi, permitiendo la interacción con otras notaciones, y aumentando la posibilidad de realizar razonamiento automático. La traducción se realiza a través de un proceso de tableau. Para validar los resultados obtenidos se comparó la complejidad de los autómatas de Buchi generados por el tableau contra autómatas conocidos encontrados en la literatura expresando el mismo lenguaje. Los resultados encontrados fueron satisfactorios, teniendo en cuenta que el proceso de tableau puede ser optimizado para

conseguir mejor desempeño. Todas estas contribuciones fueron plasmadas en el trabajo [15].

1.3.3. El origen de FVS y ω-FVS

FVS y ω-FVS están inspirados en el lenguaje VTS (Visual Timed Event Scenarios) [3, 29], cambiándole el enfoque, para pasar de describir propiedades a describir comportamiento, y removiendo de VTS todo lo referido al manejo del tiempo. Para tal fin fue preciso introducir cambios, como ser el multi-etiquetado en los puntos, las ocurrencias de eventos que pueden ocurrir simultáneamente en un punto se describen con formulas proposicionales, y la distintiva presencia de los eventos fantasmas. La formalización de FVS y ω-FVS está inspirada en la formalización presentada en [28] utilizando morfismos, con los correspondientes modificaciones que provocó el cambio de enfoque. Entre ellas, se presenta en este trabajo de investigación el concepto de escenarios traza.

1.4. Resumen de Resultados

Todas las contribuciones y resultados obtenidos durante la investigación de la presente tesis fueron plasmados en trabajos presentados a coloquios doctorales, conferencias, workshops y revistas, tanto nacionales como internacionales.

Los resultados de la primera etapa, enfocada en la especificación formal de propiedades, fueron presentados en la conferencia internacional SEKE (Software Engineering and Knowledge Engineering), especializada en la especificación formal de requerimientos [13]. Dicho trabajo presenta el impacto de la utilización de FVS para modelar y describir propiedades, enfocándose en algunos patrones de especificación. Como continuación de dicho trabajo, se presentó en la revista INTERNATIONAL JOURNAL OF SOFTWARE ENGINEERING and KNOWLEDGE ENGINEERING, una comparación más exhaustiva con otras notaciones, y describiendo por completo todos los patrones de especificación [11]. Este trabajo actualmente está en su segunda revisión, la cual fue enviada en febrero de 2013.

Para la exploración de FVS como un lenguaje de modelado orientado a aspectos, se publicaron papers en conferencias [12, 14] y revistas nacionales [10], y uno en un workhsop internacional enfocado en el modelado temprano de aspectos [17].

Las contribuciones con la extensión del poder expresivo de FVS fueron exhibidas en el trabajo [15], enviado a la revista *Requirements Engineering*. Allí se exponen la extensión del poder expresivo, el modelado de protocolos de especificación, y todo lo que implica la traducción a autómatas de Buchi.

Adicionalmente, resultados parciales de la presente tesis fueron presentando en el coloquio de doctorandos del CACIC 2011 [9].

1.5. Estructura de la tesis

La tesis está estructurada de la siguiente forma. El capítulo 2 presenta conceptos preliminares que proveen el marco y contexto necesario para presentar en los capítulos siguientes los contenidos de la tesis. En especial, se introducen conceptos sobre patrones de especificación y autómatas de Buchi. El capítulo 3 se concentra en la especificación formal de propiedades. Se hace una presentación general del problema y se presenta FVS, mostrando sus características y definiendo su sintaxis y semántica. En el capítulo 4 se muestran todos los patrones de especificación modelados en FVS mientras que en el capítulo 5 se comparan dichas especificaciones contra otras notaciones. El capítulo 6 muestra a FVS como un lenguaje de modelado orientado a aspectos, y cómo su naturaleza declarativa lo hace particularmente útil para resolver desafíos que surgen al incluir nociones de aspectos en etapas de modelado. En el capítulo 7 se presenta ω-FVS, la extensión al poder expresivo de FVS para contemplar propiedades ω-regulares. Este capítulo incluye el modelado de protocolos de comunicación como casos de estudio y el análisis del poder expresivo de ω-FVS. El capítulo 8 se enfoca en el proceso de síntesis de comportamiento, donde se exhibe el algoritmo de tableau que traduces reglas de ω-FVS a autómatas de Buchi, se demuestra la correctitud del algoritmo y se analiza la interacción entre reglas y autómatas. El capítulo 9 analiza la complejidad de los autómatas generados por el tableau, mientras que los capítulos finales 10, 11 y 12 muestran respectivamente el trabajo relacionado, trabajo futuro, y las conclusiones de la presente investigación. Por último, el apéndice A muestra la prueba formal de un lema clave para el algoritmo de tableau.

Capítulo 2

Conceptos Preliminares

En el presente capítulo se introducen conceptos claves que permiten formar un cimiento sobre el cual luego se construirán los resultados de la tesis. En particular, se introducen los patrones de especificación, que constituyen el corazón de la experimentación y validación de los resultados de la tesis, y definiciones y nociones preliminares sobre autómatas de Buchi, formalismo utilizado para la traducción y proceso de síntesis.

2.1. Patrones de Especificación

El objetivo de la presente sección es brindar un panorama general sobre los patrones de especificación propuestos por [43] en 1999. La idea sigue la propuesta de los patrones de diseño [110]. Los patrones de diseño son esquemas de soluciones generales a problemas recurrentes. En algún sentido, representan la experiencia adquirida para resolver un problema y que puede ponerse en práctica al enfrentar un desafío similar. Las ventajas son obvias: el conocimiento que transmiten los patrones ha sido validado y se ha demostrado su utilidad. Continuando este razonamiento se introdujeron patrones de especificación.

El objetivo principal de estos patrones es facilitarle la tarea al ingeniero de software al momento de especificar una propiedad. Cada patrón representa una propiedad característica que ha sido empleada en varios sistemas. Son especificados a través de una descripción en lenguaje natural que describe el comportamiento que encierra el patrón. Además de esta descripción, se proveen traducciones de la propiedad a distintos formalismos como LTL y CTL. Bajo esta modalidad, el esquema de funcionamiento esperado es el siguiente. El ingeniero de software elige el patrón de especificación que mejor se ajusta

a la propiedad que quiere escribir. Luego, dado que el patrón es una solución general, el mismo es instanciado con los eventos concretos del sistema bajo análisis, para así luego obtener la traducción de dicho patrón en el formalismo que desee.

Un ejemplo podría ser el siguiente. Un ingeniero de software desea modelar en LTL una propiedad de un servicio web, la cual establece que todos los pedidos recibidos serán procesados. El patrón de especificación “Respuesta” es seleccionado ya que se ajusta al comportamiento que se desea modelar. La descripción en lenguaje natural para dicho patrón es la siguiente: “Una o más ocurrencias del evento **acción** resultarán en una o más ocurrencias del evento **respuesta**”, y la correspondiente codificación en LTL propuesta por [43] es: $\Box(accion \rightarrow \Diamond respuesta)$. Una vez seleccionado el patrón, el ingeniero de software debe instanciar los eventos generales en los eventos concretos del sistema bajo análisis. Asumiendo eventos como *pedidoCliente* y *pedidoProcesado* para el sistema del servicio web, el ingeniero instancia el evento **acción** con *pedidoCliente* y el evento **respuesta** con *pedidoProcesado*, para así obtener la siguiente formula LTL: $\Box(pedidoCliente \rightarrow \Diamond pedidoProcesado)$. De esta manera, el ingeniero de software elude el problema de especificar directamente en LTL la propiedad deseada, tomando un camino que le resulta más natural.

Los patrones de especificación se agrupan en dos categorías de acuerdo a su semántica y comportamiento. En la categoría denominada *Occurrencia* se encuentran los patrones que describen si determinados eventos ocurren o no, mientras que la categoría *Orden* se enfoca en condicionar el orden en qué los eventos ocurren. Dentro de la primer categoría se encuentran los siguientes patrones: **Ausencia**, **Universal**, **Existencia** y **Existencia limitada**. El patrón **Ausencia** sirve para denotar que un determinado evento no debe ocurrir durante una porción de la ejecución del sistema, el **Universal** sirve para explicitar que un determinado evento debe estar siempre presente, el **Existencia** para especificar que un determinado evento debe ocurrir al menos una vez en la ejecución del sistema, y el patrón **Existencia limitada** se utiliza para poner un número máximo de veces que un determinado evento debe ocurrir. La categoría **Orden** incluye otros cinco patrones: **Precedencia**, **Respuesta**, **Precedencia Encadenada**, **Respuesta Encadenada**, y por último, el patrón **Encadenamiento Restringido**. El patrón **Precedencia** establece que la ocurrencia de un evento a debe siempre siempre precedida por la ocurrencia de un evento b. El patrón **Respuesta** establece que un evento a debe ser siempre seguido por

la ocurrencia de un evento b. Los patrones de **Precedencia Encadenada** y **Respuesta Encadenada** representan una extensión de los patrones anteriores, ya que contemplan no solo un evento, sino una cadena de eventos. Por ejemplo, se podría especificar utilizando el patrón **Respuesta Encadenada** que un evento a debe ser seguido por los eventos b, c y d. Finalmente, el patrón **Encadenamiento Restringido** permite poner restricciones en la cadena de eventos. Continuando el ejemplo anterior, se podría especificar que un evento a debe ser seguido por los eventos b,c y d, tal que no ocurra un evento e entre b y d.

Otro aspecto importante en la utilización de patrones es la noción de **alcance**, la cual establece el período sobre el cual se espera que se cumpla la propiedad. El alcance **global** se utiliza para establecer que el patrón debe valer durante toda la ejecución del sistema, por lo que es el mas abarcativo de todos. Existen otros cuatro alcances que restringen el campo de aplicación de un patrón. El alcance **Antes de P** especifica que el patrón debe cumplirse antes de la primera ocurrencia del evento P. Similarmente, el alcance **Después de Q** especifica que el patrón debe cumplirse luego de la primera ocurrencia del evento Q. El alcance **Después de Q hasta P** exige que el patrón se cumpla luego de la ocurrencia del evento Q, pero antes de la ocurrencia del evento P, si es que el evento P ocurre. Si se quiere establecer que ocurran ambos delimitadores se debe utilizar el alcance **Entre P y Q**.

Los patrones de especificación constituyen un importante y representativo caso de estudio dentro del mundo de la especificación de propiedades. En [43] se llevó a cabo un experimento donde recolectaron 555 especificaciones de al menos 35 fuentes diferentes, y se develó que el 92 % de las propiedades pudieron ser especificadas utilizand los patrones de especificación propuestos.

2.2. Autómatas de Buchi

La presente tesis se basa en el tradicional modelo matemático de autómatas de Buchi sobre palabras infinitas, donde las transiciones son etiquetadas con fórmulas proposicionales [31]. Más formalmente, un autómata de Buchi es una 5-upla $\mathcal{B} = \langle \Sigma, S, S^0, \Delta, F \rangle$ tal que:

- Σ es un alfabeto finito.

- S es un conjunto finito de estados.
- $S^0 \subseteq S$ es un conjunto de estados iniciales.
- $F \subseteq S$ es un conjunto de estados finales.
- $\Delta \subseteq S \times \mathcal{PL}(\Sigma) \times S$ es la función de transición, donde $\mathcal{PL}$ es el conjunto de formulas proposicionales que pueden ser obtenidos de Σ.

Una **valuación** es una función total v: $\Sigma \rightarrow Bool$, que asigna variables a valores booleanos. Dado esto, una **corrida** p de un autómata de Buchi B se define como una secuencia $p = s_0 v_0 s_1 v_1 ...$ alternando estados y valuaciones de manera tal que $s_0 \in S^0 \wedge \forall i\ (s_i \in S) \wedge (\exists (s_i, \phi, s_{i+1}) \in \Delta \wedge \phi\ (v_i) = true)$. La secuencia de valuaciones $v_0 v_1 ... v_n$ en una corrida p constituye una **traza** del autómata. La semántica de un autómata de Buchi es la descrita en [31]. Sea *inf(p)* el conjunto de estado que aparecen infinitamente frecuente en una corrida p. Luego, una corrida p sobre una palabra infinita es **aceptadora** si y solo si *inf(p)* $\cap\ F \neq \emptyset$. Esto es, cuando algún estado aceptador aparece en p infinatamente frecuente. Se define el conjunto de trazas aceptado por un autómata como sigue.

Definition 2.2.1 (Trazas de un autómata)**.** *El conjunto de trazas aceptado por un autómata* B*, denotado* Tr(B)*, es definido como el conjunto de valuaciones* $= v_0 v_1 ... v_n$ *de cada corrida aceptadora* p*. Como es usual,* Tr(B) *denota el lenguaje reconocido por* B*.*

Es común al manipular alfabetos de autómatas que se necesiten proyectar, esconder, o manejar algunas variables de una manera particular. Dado un alfabeto $\Sigma^{'} \subseteq \Sigma$, se define el operador $\downarrow_{\Sigma'}$ para denotar la proyección del alfabeto Σ sobre $\Sigma^{'}$. En este contexto, la función de valuación sobre $\Sigma^{'}$, $v\downarrow_{\Sigma'}$ es una función total $\Sigma^{'} \rightarrow Boolean$ tal que $v\downarrow_{\Sigma'}$(e)= v(e) $\forall e \in \Sigma^{'}$. De manera similar se pueden establecer el conjunto de trazas sobre $\Sigma^{'}$ para un autómata B, como lo establece la siguiente definición.

Definition 2.2.2 (Trazas Proyectadas de un autómata)**.** *Dado un autómata B con alfabeto Σ y $\Sigma^{'} \subseteq \Sigma$, el conjunto de trazas para* B *over $\Sigma^{'}$, denotado* Tr(B)$\downarrow_{\Sigma'}$*, se define como* $\{\ t' |\ t \in Tr(B) \wedge t^{'} = t \downarrow_{\Sigma'} \}$.

A continuación se define una operación de proyección entre autómatas.

Definition 2.2.3 (Proyección de autómatas)**.** *Dado un autómata $\mathcal{B} = \langle \Sigma, S, S^0, \Delta, F \rangle$ se define el autómata proyectado sobre $\Sigma^{'} \subseteq \Sigma$ como $B\downarrow_{\Sigma'} = \langle \Sigma^{'}, S^{'}, S^{'0}, \Delta^{'}, F^{'} \rangle$ tal que*

T1: $S^{'} = S$

T2: $S^{'0} = S^0$

T3: $F^{'} = F$

T4: *Para cada* $\langle S_i, \mathcal{A}^{'}, S_j\rangle$ *en* $\Delta^{'}$ $\exists$ $\langle S_i, \mathcal{A}, S_j\rangle$ *en* Δ *tal que* $\mathcal{A}^{'} \Rightarrow \exists\ \upsilon_1,\upsilon_2 \ldots \upsilon_n\ \mathcal{A}$ *es una tautología, donde* $\Sigma - \Sigma^{'} = \{\upsilon_1,\upsilon_2 \ldots \upsilon_n\}$ *.*

Como puede observarse, el autómata proyectado B$\downarrow_{\Sigma'}$ es casi idéntico al autómata B, salvo por el alfabeto ($\Sigma^{'} \subseteq \Sigma$), y el proceso de eliminación existencial dado en el item **T4:** manipulando la función de transición Δ. Notar que $\mathcal{A}^{'} \Rightarrow \exists\ \upsilon_1,\upsilon_2 \ldots \upsilon_n\ \mathcal{A}$ es una fórmula booleana cuantificada [46].

El siguiente lema relaciona el lenguaje aceptado por un autómata y su proyección B$\downarrow_{\Sigma'}$.

Lemma 2.2.4. *Dado un autómata* $\mathcal{B} = \langle\Sigma, S, S^0, \Delta, F\rangle$ *y su proyección* $B\downarrow_{\Sigma'}$ *entonces* Tr(B$\downarrow_{\Sigma'}$*)* = Tr(B)$\downarrow_{\Sigma'}$*. En otras palabras, la proyección de las trazas aceptadas por* B *es exactamente el lenguaje denotado por* $B\downarrow_{\Sigma'}$*.*

Este lema se verifica trivialmente siguiendo la clásica proyección entre trazas [31].

Capítulo 3

Especificación temprana de comportamiento

En este capítulo se presenta el desafío de la especificación temprana de comportamiento. Tras una introducción en la sección 3.1, se presenta el lenguaje declarativo FVS. Más precisamente, la sección 3.2 presenta sus características de manera intuitiva, mientras que la sección 3.3 formaliza su sintaxis y semántica.

3.1. Introducción

La especificación de propiedades es todavía hoy uno de los desafíos más importantes para lograr la transferencia de técnicas de verificación de software como model checking [31] y testing basado en modelos [105, 34]. Los usuarios de estas técnicas deben enfrentar el desafío de expresar propiedades en el lenguaje o formalismo usado en la herramienta de especificación. Utilizar lenguaje natural para expresar requerimientos puede parecer una alternativa a la utilización de notaciones formales, pero en general se obtienen especificaciones ambiguas e imprecisas, debilitando así la capacidad de realizar procesos automáticos de verificación. Otra posibilidad es utilizar lenguajes de especificación cuya notación sea similar a la utilizada en lenguajes de programación conocidos y familiares, como JML [72] o Spec Explorer [109], utilizados generalmente para la especificación bajo la modalidad pre y post condiciones. Sin embargo, cuando el objetivo es razonar sobre el comportamiento esperado sobre trazas en sistemas reactivos de manera declarativa la opción más utilizada son aproximaciones formales como Lógica Lineal Temporal (LTL)

u notaciones operacionales como máquinas de estado. Estas aproximaciones requieren que los usuarios finales sean expertos en el formalismo utilizado para poder expresar con precisión los requerimientos([47, 53, 43, 98, 32, 90, 18]), lo cual constituye un obstáculo serio.

La utilización de patrones ha sido propuesta como una interesante alternativa para solucionar este problema [43, 42]. El principal propósito de un patrón es capturar soluciones recurrentes a un tipo particular de problemas. En [43], los patrones se describen considerando dos aspectos. Por un lado, se describe la *intención* de un patrón, donde se especifica el comportamiento esperado del patrón. Dicha descripción se realiza utilizando una notación de lenguaje natural disciplinado (en inglés DNL [98]). Por ejemplo, la *intención* del patrón *Respuesta* es "Una o más ocurrencias del evento **acción** resultarán en una o más ocurrencias del evento **respuesta**". Por otro lado, cada patrón tiene un alcance, el cual define la porción de la ejecución del programa sobre la cual el patron debe valer. Por ejemplo, se puede especificar que un patrón debe valer **entre** la ocurrencia de dos eventos dados.

A pesar de que los patrones proveen una forma más amigable de expresar requerimientos, el usuario final todavía debe validar la propiedad. Esto es básicamente responder la siguiente pregunta: "*¿Modela esta propiedad correctamente el requerimiento?*". Adicionalmente, el usuario necesita poder comparar con facilidad propiedades. Por ejemplo, para poder seleccionar entre dos propiedades candidatas el usuario debe poder responder preguntas usuales como "*¿Cuál de las dos representa un requerimiento más fuerte?*,'*¿Cuál de las dos impone más restricciones?*, '*¿Están relacionadas ambas propiedades?, ¿Porqué?*. Otra fuente útil de información al momento de validad una propiedad es razonar mediante comportamiento complementario. Esto es, poder observar y analizar los escenarios generales que llevan a la violación de la propiedad, y así poder tener un panorama más completo acerca del comportamiento que la propiedad está excluyendo del sistema. Esta característica es especialmente importante al momento de estar conociendo y explorando el comportamiento esperado del sistema. Realizar todas estas tareas de validación utilizando solamente la descripción DNL del patrón puede ser difícil de realizar, por lo que se debe analizar la traducción del patrón a un formalismo dado [53, 98]. Además, muchas veces una propiedad debe ser modificada en algún aspecto para poder adaptarlar al sistema actual, y esto lleva, nuevamente, a manipular la traducción del patrón.

Todo esto implica que la utilización de patrones no es suficiente, no alcanza para poder esconder completamente al usuario la complejidad y las sutilezas de los lenguajes formales. Luego, surge la necesidad de contar con un lenguaje formal de especificación que sea fácil de usar, y lo suficientemente expresivo para permitir a los usuarios expertos y no expertos expresar en él los requerimientos y las propiedades del sistema [81, 53]. Teniendo esto en cuenta, en la presente investigación se proponen cuatro atributos de calidad deseables para un lenguaje formal de especificación: *sucinto*, *comparabilidad*, *complemento*, y *modificabilidad*. A continuación se describe cómo los formalismos más utilizados manejan estos atributos de calidad, concentrandose principalmente en LTL.

3.1.1. Declaración declarativa de propiedades: dificultades actuales

Como se mencionó anteriormente, se proponen cuatro atributos de calidad como características deseables en un lenguaje formal de especificación que maneje adecuadamente tareas de validación: *sucinto*, *comparabilidad*, *complemento*, y *modificabilidad*.

Sucinto, definido en [112], se refiere a qué tan compacta y breve puede ser expresada una fórmula, y es considerado como una de las medidas cuantitativas más importantes al momento de caracterizar el poder expresivo de un lenguaje y comparar su fortaleza con otras notaciones [48]. En este sentido, *sucinto* busca que la especificación de un patrón expresada en el formalismo sea tan compacta como la descripción DNL del patrón.

Las especificaciones formales de los patrones deben ser fáciles de comparar y distinguir. Este objetivo constituye el atributo *comparabilidad*. Por ejemplo, al comprar dos patrones, es natural tratar de entender cuál de los dos es más restrictivo. Más concretamente, se define *comparabilidad* como la habilidad de considerar dos especificaciones formales de dos patrones relacionados y poder entender las diferencias y similitudes entre ambas, y poder determinar si entre ellas existe algún tipo de implicación lógica.

Complemento se refiera a la habilidad de razonar sobre el comportamiento que lleva a violar una propiedad, que constituye una información valiosa para el usuario. Un lenguaje de especificación formal debe proveer una manera sencilla de razonar sobre el comportamiento complementario. Específicamente, se define *complemento* como la habilidad de un lenguaje de generar escenarios, expresados en el mismo lenguaje, que ilustren

el comportamiento que lleva a la violación de una propiedad.

Finalmente, se define *modificabilidad* como la habilidad de manipular artefactos expresados en el formalismo de manera que una especificación pueda adaptarse a cambios en el contexto de aplicación, siguiendo el espíritu de lo propuesto por Parnas en [80]. Esto es, pequeños cambios en el contexto de aplicación deben tener como respuesta pequeños y bien determinados cambios en la especificación. Esto implica que la diferencia entre la especificación original y la modificada debe ser pequeña, y relativa al cambio realizado.

En algún punto, las traducciones en LTL propuestas en [43, 42] para los patrones de especificación no logran manejar adecuadamente los cuatro atributos de calidad propuestos. Las formulas LTL obtenidas son en algunos casos artefactos complejos, que son difíciles de comparar sin manipulación deductiva y asistencia por herramienta. Esta situación se ve incluso con propiedades relativamente sencillas. El razonamiento por complemento también es complicado ya que quiere manipulaciones sofisticadas. Más aún, para algunas formulas LTL la repuesta que brindan algunas herramientas no aportan mucho al usuario para poder ganar conocimiento sobre la propiedad. Por ejemplo, el usuario puede utilizar alguna herramienta sobre LTL para obtener la negación de una fórmula. Sin embargo, a partir de esta nueva formula que representa el comportamiento complementario no es sencillo visualizar los escenarios generales que llevan a violar la propiedad. De manera similar, para poder comparar y analizar si una dada fórmula implica lógicamente a otra para poder compararlas, la respuesta "Si" or "No" que puede devolver una herramienta al recibir dos fórmulas es de poca utilidad al usuario, ya que no brinda demasiada información sobre el porqué de la respuesta. Modificabilidad implica manipulación lógica de las formulas. Dicha manipulación, en una fórmula con muchos argumentos, operadores y nivel de anidamiento es usualmente una tarea engorrosa y propensa a errores, y no hay herramientas disponibles que ayuden al usuario en el proceso.

El panorama no cambia demasiado si se utilizan notaciones basadas en autómatas. Para poder comparar con precisión dos autómatas se deben llevar a cabo operaciones de inclusión de lenguajes. Similarmente, operaciones para complementar el lenguaje de un autómata no son para nada triviales, y pueden estar expuestas a problemas de explosión de estados. Con respecto a modificabilidad, se puede trazar un razonamiento análogo. Operaciones intrincadas pueden llegar a ser necesarias para adaptar un autómata a diferentes situaciones.

Para concluir, es importante destacar que el poder expresivo del lenguaje de especificación es un aspecto a tener en cuenta. Tanto la complejidad como el poder expresivo deben estar sutilmente balanceado: el formalismo debe ser lo suficientemente expresivo para expresar todas las propiedades de interés, manteniendo las especificaciones simples y aptas para realizar tareas de validación.

A continuación se presenta el lenguaje declarativo FVS, la propuesta de la presente investigación para intentar solucionar los problemas mencionados.

3.2. FVS: Feather weight Visual Scenarios

En esta sección se describirán de manera informal las principales características de FVS. Una caracterización formal del lenguaje se presenta más adelante en la sección 3.3.

FVS es un lenguaje gráfico basado en escenarios. Los escenarios son órdenes parciales de eventos basados en puntos, los cuales son etiquetados con los posibles eventos que pueden ocurrir en ese punto, y en flechas que los conectan. Una flecha entre dos puntos indica precedencia del origen respecto del destino. Por ejemplo, en la figura 3.1-(a) el evento A precede al evento B, indicando que ocurre antes. FVS cuenta con abreviaciones para indicar la próxima ocurrencia de un evento luego de otro, o la ocurrencia inmediata anterior de un evento precediendo a otra. Para el primer caso, la abreviación se denota gráficamente con una segunda flecha (abierta) cerca del punto de destino. Por ejemplo, en la figura 3.1-b el escenario captura la primer ocurrencia de un evento B que sigue a un evento A. Para el segundo caso, la segunda flecha se ubica cerca del punto de origen. Continuando el ejemplo, el escenario en la figura 3.1-c captura la ocurrencia inmediata anterior de un evento A que precede a un evento B. Se introducen en la notación dos puntos destacados para denotar el principio y el fin de una ejecución: un circulo completo grande para el comienzo y dos circulos concéntricos para el final (ilustrado en la figura 3.1-d). Las flechas pueden ser etiquetadas para restringir comportamiento, interpretándose como eventos prohibidos. En la figura 3.1-e el evento A precede al evento B de manera tal que el evento C no ocurre entre ambos. Notar en este punto que los operadores de eventos próximo y anterior introducidos previamente pueden modelarse con restricciones como se ve en la figura 3.1-e. El escenario en la parte superior de la figura 3.1-e es equivalente al escenario en la figura 3.1-b, y similarmente lo son el escenario en la parte inferior de la

figura 3.1-e con el escenario en la figura 3.1-c. Finalmente, FVS cuenta con la posibilidad de definir "aliasing" entre puntos. El escenario en la figura 3.1-g indica que la ocurrencia del evento A también implica la ocurrencia simultánea del evento B. En este punto vale la pena destacar que el evento A se repite en el etiquetado del segundo punto sólo para una cuestión formal de la sintaxis del lenguaje (ver sección 3.3)

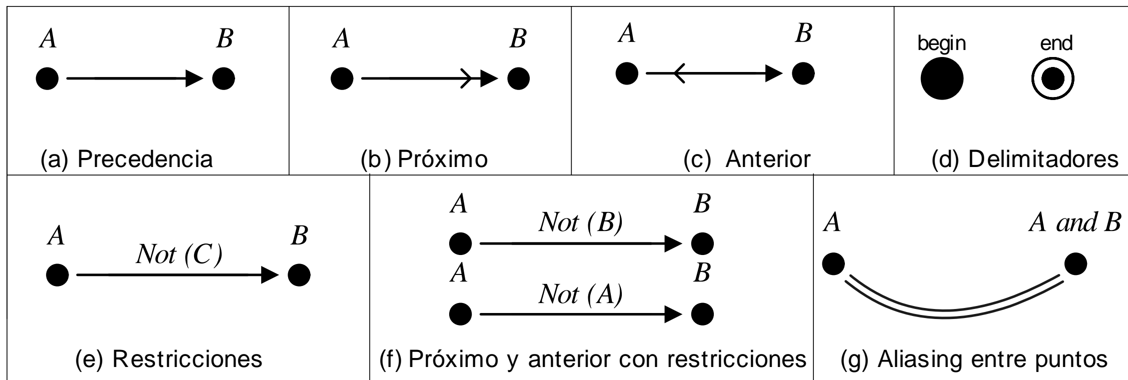

Figura 3.1: Elementos Básicos en FVS

Ejemplo: Protocolo simple de comunicación

Para una mejor compresión de esta sección se brinda un pequeño ejemplo inspirado en un protocolo simple y pequeño, donde un servidor procesa pedidos por parte de los clientes, los cuales intentan acceder a determinados recursos, los cuales pueden estar bloqueados. El protocolo maneja cuestiones de seguridad a través del uso de contraseñas. El escenario en la figura 3.2 tipifica una situación donde ocurren tres eventos distintos antes de que un pedido de acceso a un recurso sea otorgado: el pedido del acceso, el ingreso de una password de manera correcta, y que el recurso esté en estado liberado. Notar que el escenario no estipula ningún orden especial de ocurrencia entre el pedido, el ingreso de la contraseña y el chequeo del estado. Lo único que requiere es que los tres eventos ocurran, sin importar el orden en que los mismos ocurren. Es decir, contempla todas las posibles combinaciones.

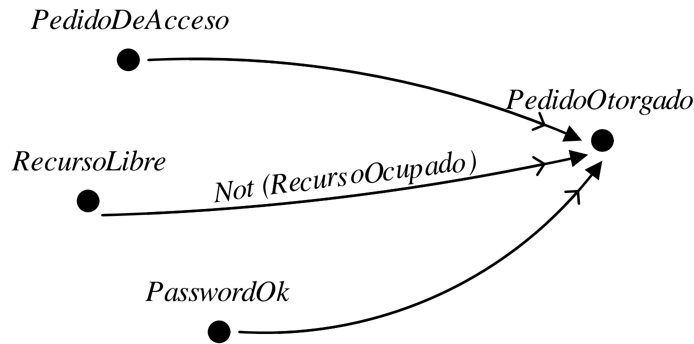

Figura 3.2: Escenarios FVS modelando el comportamiento de un protocolo

3.2.1. Reglas FVS

En esta sección se presenta el concepto de **Escenario de Reglas** o simplemente una regla[1], un concepto clave del lenguaje. Básicamente, una regla está dividida en dos partes: un escenario tomando el rol del antecedente, y al menos un scenario tomando el rol de un consecuente. La intuición es que siempre que una traza "encuentre" un antecedente, entonces también debe encontrar al menos uno de los consecuentes. En otras palabras, una regla toma la forma de una implicación, con un escenario antecedente, y uno o más escenarios consecuentes. El escenario antecedente es una sub-estructura común a todos los consecuentes, lo cual permite establecer relacione complejas entre los puntos del antecedente y consecuentes. Esto otorga gran flexibilidad a FVS, ya que sus reglas no están limitadas, como la mayoría de las aproximaciones que manejan algún tipo de escenarios de implicación, donde el antecedente funciona únicamente como una estructura que debe preceder a los consecuentes. De esta manera las reglas pueden expresar comportamiento que ocurrió en el pasado, o en el medio de otros eventos. Gráficamente, el antecedente se muestra de color negro, mientras que los consecuentes en color gris. Como las reglas pueden tener más de un consecuente, cada elemento que no pertenece al antecedente se identifica con un número para identificar a qué consecuente pertenece. La figura 3.3 muestra un ejemplo de una regla FVS. La interpretación es que, siempre que un evento *PedidoDeAccesso* es seguido por un evento *PedidoOtorgado* (sin la ocurrencia de un evento *logoff* entre ellos), una de dos posibles secuencias de eventos debe también observarse en la traza. La primera (consecuente 1) requiere que luego de que se hace el pedido, se ingrese

[1]Los escenarios de reglas representan la versión FVS de los escenarios condicionales del lenguaje VTS [29]

de manera correcta una contraseña. En cambio, la segunda (consecuente 2) contempla la posibilidad de que la contraseña haya sido ingresada antes de la realización del pedido. Es decir, o se ingresó la contraseña luego de hacerse el pedido, o el usuario ya la había ingresado al momento de realizar el pedido. Es importante notar el poder expresivo de FVS, donde el antecedente no está forzado a preceder al consecuente.

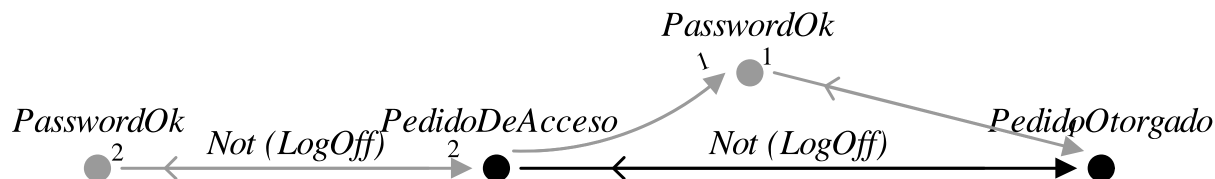

Figura 3.3: Reglas FVS

3.2.2. Anti-Scenarios

Una interesante característica en FVS es que anti-escenarios pueden generarse automáticamente a partir de reglas. El algoritmo que los genera está detallado en [29], pero brevemente lo que el algoritmo hace es computar todas las posibles situaciones donde se encuentra un antecedente en la traza, pero no es posible encontrar alguno de los consecuentes. Originalmente, los anti-escenarios fueron propuestos como una manera posible de realizar model checking en el lenguaje VTS [29]. Un efecto de esto es que los anti-escenarios representan información valiosa para el ingeniero de software ya que representan escenarios generales de cómo eventos pueden encadenarse de tal forma de provocar la violación de una regla.

Un anti-escenario para la regla de la figura 3.3 se muestra en la figura 3.4. En este caso, no se ingresó nunca una contraseña de manera correcta desde el comienzo de la ejecución y aún así el acceso es otorgado.

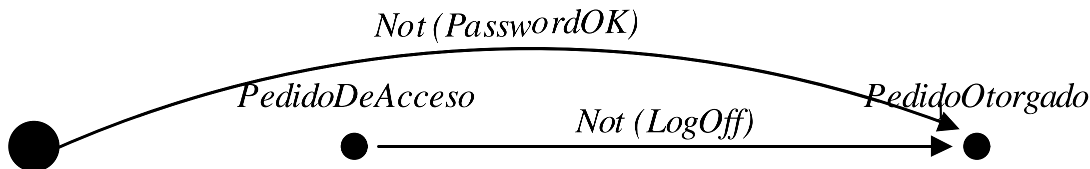

Figura 3.4: Un anti-escenario en FVS

3.3. Sintaxis y Semántica de FVS

En esta sección se describe de manera formal la sintaxis y semántica de FVS. Dicha descripción seguirá los siguientes pasos. En prima instancia se definirá el concepto de *escenarios FVS*. Luego, se define una operación clave entre escenarios: los **morfismos**, la cual permitirá definir la noción de *reglas FVS*. Finalmente, se definirá la semántica formal de FVS, estableciendo el concepto de *trazas* y satisfacción de reglas.

3.3.1. Sintaxis FVS

Un **escenario FVS** queda establecido por lo delineado en la siguiente definición.

Definition 3.3.1 (Escenario FVS)**.** *Un* escenario FVS *es una tupla* $\langle \Sigma, P, \ell, \equiv, \not\equiv, <, \gamma \rangle$ *tal que:*

S1: Σ *es un conjunto de variables proposicionales representando tipos de eventos;*

S2: P *es un conjunto finito de puntos;*

S3: $\ell : P \to \mathcal{PL}(\Sigma)$*, es una función que etiqueta puntos con una fórmula dada. Más precisamente,* $\mathcal{PL}$ *es el conjunto de fórmulas proposicionales que pueden obtenerse a partir de* Σ*;*

S4: $\equiv \ \subseteq P \times P$ *es una relación de equivalencia, utilizada para hacer "aliasing" entre puntos. Puntos relacionados por esta función denotan el mismo comportamiento ;*

S5: $\not\equiv \ \subseteq P \times P$ *es una relación asimétrica entre puntos. Se utiliza para indicar "separación" entre puntos);*

S6: $<\subseteq (P \uplus \{\mathbf{0}\} \times P \uplus \{\infty\}) \setminus \{\langle \mathbf{0}, \infty \rangle\}$ *es una relación de precedencia entre puntos (*$\mathbf{0}$ *y* ∞ *representan el comienzo y el final de la ejecución, reespectivamente.); La precedencia se usa para indicar el orden en que ocurren los eventos.*

S7: $\gamma : (\not\equiv \cup <) \to \mathcal{PL}(\Sigma)$ *asigna a cada par de puntos, relacionados por precedencia o separación, una fórmula que restringe el conjunto de eventos que puede ocurrir entre el par de puntos. La función* γ *satisface la siguiente condición.* $\gamma(\mathsf{p}, \mathsf{q}) \Rightarrow \gamma(\mathsf{p}, \mathsf{w}) \vee \ell(\mathsf{w}) \vee \gamma(\mathsf{w}, \mathsf{q}), \forall \mathsf{p} < \mathsf{w} < \mathsf{q} \in P$*. Esta condición chequea que los eventos asignados a un par de puntos por esta función impliquen lógicamente cualquier otro evento ocurriendo entre ellos. Es decir , si el evento* c *ocurre entre los eventos* a *y* b*, entonces* $\neg$ *no puede estar incluido en* $\gamma(a, b)$

Morfismos

Ahora se define formalmente el concepto de *morfismo* entre escenarios. Intuitivamente, se busca obtener un matching entre escenarios, es decir, un *mapeo* entre sus puntos exhibiendo cómo un escenario "especializa" al otro.

Definition 3.3.2 (Morfismo)**.** *Dados dos escenarios $\mathcal{S}_1$, $\mathcal{S}_2$ (asumiendo una universo común de proposiciones de eventos), y f una función total entre P_1 y P_2 decimos que f es un morfismo de $\mathcal{S}_1$ a $\mathcal{S}_2$ (denotado $f : \mathcal{S}_1 \rightarrow \mathcal{S}_2$) si y solo si*

M1: $\ell_2(a) \Rightarrow \ell_1(\mathsf{p})$ *es una tautología para todo* $\mathsf{p} \in P_1$ *y todo* $a \in P_2$ *tal que* $a \equiv_2 f(p)$*;*

M2: $\gamma_2(f(\mathsf{p}), f(\mathsf{q})) \Rightarrow \gamma_1(\mathsf{p}, \mathsf{q})$ *es una tautología para todo* $\mathsf{p}, \mathsf{q} \in P_1$*;*

M3: *si* $\mathsf{p} \equiv_1 \mathsf{q}$ *entonces* $f(\mathsf{p}) \equiv_2 f(\mathsf{q})$ *para todo* $\mathsf{p}, \mathsf{q} \in P_1$*;*

M4: *si* $\mathsf{p} \not\equiv_1 \mathsf{q}$ *entonces* $f(\mathsf{p}) \not\equiv_2 f(\mathsf{q})$ *para todo* $\mathsf{p}, \mathsf{q} \in P_1$*;*

M5: *si* $\mathsf{p} <_1 \mathsf{q}$ *entonces* $f(\mathsf{p}) <_2 f(\mathsf{q})$ *para todo* $\mathsf{p}, \mathsf{q} \in P_1$*.*

Se puede ver que $\mathcal{S}_2$ tiene más restricciones que $\mathcal{S}_1$ cuando existe una morfismo $m : \mathcal{S}_1 \rightarrow \mathcal{S}_2$. Esta relación entre dos escenarios establece que $\mathcal{S}_1$ está embebido en $\mathcal{S}_2$ (esto es análogo a la subsunción lógica). Análogamente, se puede afirmar que, en este caso, que $\mathcal{S}_2$ especializa $\mathcal{S}_1$.

En la figura 3.5 se puede ver un ejemplo un morfismo en acción (el morfismo se muetra con flechas punteadas y no forma parte de la representación gráfica de FVS), desde el $\mathcal{S}_1$ to scenario $\mathcal{S}_2$. El escenario en la parte superior de la figura (escenario $\mathcal{S}_2$) muestra una sequencia de pedidos y respuestas, pero también teniendo en cuenta eventos que reflejan el estado de los recursos. Por otro lado, el escenario en la parte inferior de la figura (escenario $\mathcal{S}_1$) muestra únicamente el pedido de un cliente, seguido de su respuesta. Puede observarse que el escenario $\mathcal{S}_2$ exhibe más restricciones al enfocarse también en el estado de los recursos. En particular, y repasando la definición 3.3.2 se puede ver que se satisfacen todas las condiciones de un morfismo. Más concretamente, se satisface que $Cliente1\ PedidoDeAcesso \Rightarrow Cliente1\ PedidoDeAcesso$ y $RecursoLibre \wedge PedidoOtorgado \Rightarrow PedidoOtorgado$ son tautologías.

Reglas FVS

Las reglas modelan el comportamiento esperado del sistemas, proveyendo un mecanismo expresivo, flexible y poderoso para predicar y razonar sobre el comportamiento del

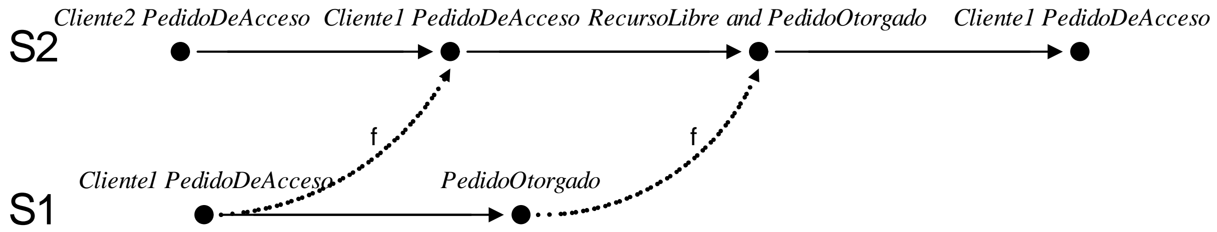

Figura 3.5: Un ejemplo de un morfismo entre escenarios

sistema. Como se mencionó anteriormente, la estructura de una regla consta de dos partes: un escenario que toma el papel de antecedente de la regla, y, al menos, un escenario tomando el papel del consecuente. Siempre que en una traza se establezca un matching con el antecedente, esto es, que exista un morfismo f, entonces f debe ser extensible para mostrar un matching con un escenario consecuente. Esto es, al menos uno de los consecuentes también es matcheado. A continuación se da la definición formal.

Definition 3.3.3 (Regla FVS)**.** *Dado un escenario* $\mathcal{S}_0$ *(*antecedente*) y un conjunto indexado de escenario y morfismos desde el antecedente* $f_1 : \mathcal{S}_0 \to \mathcal{S}_1$*,* $f_2 : \mathcal{S}_0 \to \mathcal{S}_2$*, ...,* $f_k : \mathcal{S}_0 \to \mathcal{S}_k$ *(*consecuentes*), decimos que* $\mathrm{R} = \langle \mathcal{S}_0, \{f_i\}_{i=1\ldots k} \rangle$ *es una regla FVS.*

Como ejemplo, considerar las siguientes reglas en la figura 3.6, las cuales modelan un fragmento del comportamiento esperado del protocolo mencionado anteriormente. La regla en la figura 3.6-a dice que todo pedido será eventualemente otorgado. Similarmente, la regla en la figura 3.6-b dice básicamente que siempre que ocurre un evento *PedidoOtorgado* entonces tiene que haber ocurrido en el pasado un evento *Cliente PedidoDeAcceso*. Esto es, cada evento *PedidoOtorgado* debe ser precedido por un pedido del cliente. Finalmente, la regla en la figura 3.6-c establece que el cliente 1 no puede hacer dos pedidos consecutivos sin que ocurra antes un pedido por parte del cliente 2. Esta regla fuerza una alternancia entre los pedidos de dos clientes. Vale la pena mencionar que esta reglas representan posibles instanciaciones de algunos patrones de especificación [43]: el patrón *Respuesta* en la figura 3.6-a, el patrón *Precedencia* en la figura 3.6-b y el patrón *Existencia*, en la figura figure 3.6-c.

A continuación se presenta la semántica del lenguaje declarativo FVS.

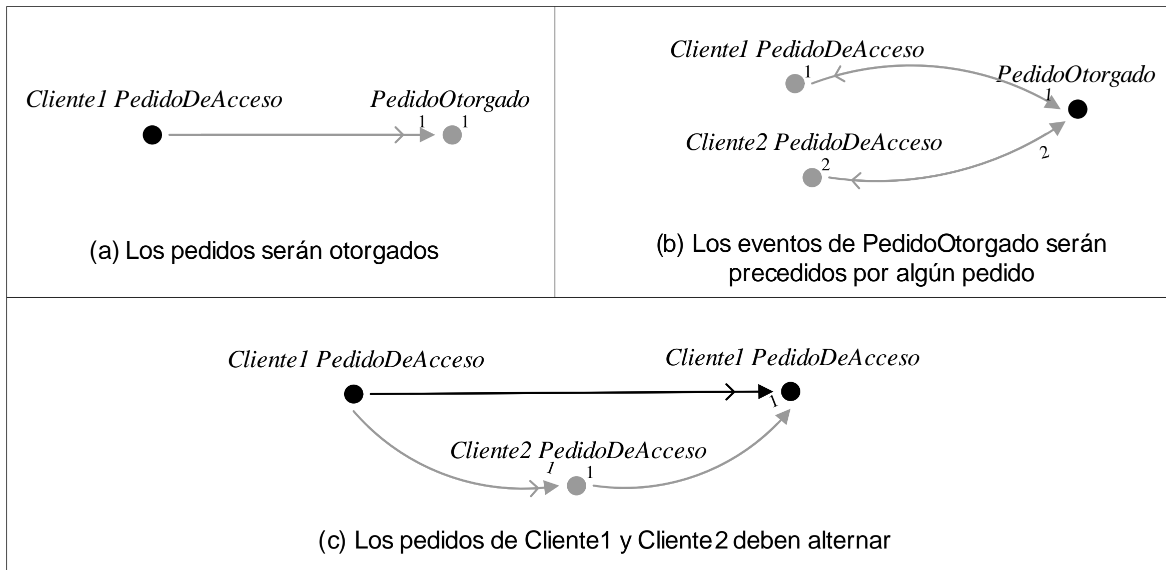

Figura 3.6: Reglas FVS modelando un protocolo de comunicación

3.3.2. Semántica FVS

Establecer la semántica de FVS implica definir cuando una traza satisface un conjunto dado de reglas. Para lograr este objetivo será necesaria transitar los siguiente pasos:

1. Establecer cuándo un escenario satisface una regla
2. Definir la noción de trazas como escenarios
3. Establecer cuándo una traza satisface un conjunto de reglas

La semántica de FVS estará basada en la noción de morfismos (definición 3.3.2). Para cumplir con el primer paso se presenta la siguiente definición, la cual se enfoca en precisar cuándo una escenario $\mathcal{S}$ satisface una regla R:

Definition 3.3.4 (Semántica de una regla FVS)**.** *Un escenario $\mathcal{S}$ satisface una regla R ($\mathcal{S} \models R$) sy y solo si para cada morfismo $m : \mathcal{S}_0 \rightarrow \mathcal{S}$ existe $m_i : \mathcal{S}_i \rightarrow \mathcal{S}$,* ***para algún*** *$i \in \{1..k\}$, tal que $m = m_i \circ f_i$.*

Un ejemplo concreto de la semántica del lenguaje a través de morfismos se ve en la figura 3.7. El escenario (en la parte superior de la figura) satisface la regla (en la parte inferior de la figura) ya que siempre que un evento *PedidoDeAcceso* es encontrado en el escenario, esto es, siempre que existe un morfismo desde el antecedente de la regla al escenario (en el caso de la figura 3.7 hay dos, morfismos *m1* y *m2*), un evento de tipo

PedidoOtorgado es también encontrado después. Formalmente, para el morfismo *m1* los morfismos {*m1,m3*} es un posible morfismo del consecuente del escenario al escenario objetivo de manera que extiende *m1* (es decir, conmutan tal como es pedido en la definición 3.3.4). Análogamente, {*m2,m4*} es una posible extensión para *m2*.

PedidoDeAcceso PedidoOtorgado PedidoDeAcceso PedidoOtorgado
m1 m3 m2 m4
PedidoDeAcceso PedidoOtorgado

Figura 3.7: Ejemplo de un escenario satisfaciendo una regla

Los pasos 2 y 3 involucra tratar con trazas. Esto requiere definir la noción de trazas de manera formal, así como su relación con escenarios y satisfacción de reglas. Como siempre, el objetivo es siempre el mismo: definir la semántica de FVS como el conjunto de trazas válidas del sistema, es decir, aquellas que satisfacen las reglas que describen el comportamiento del sistema.

Semántica basada en trazas

Como se mencionó anteriormente, la noción de trazas abstrae la posible ejecución de un sistema basado en eventos. Para mantener la homogeneidad del lenguaje, las trazas serán definidas como un tipo especial de escenarios FVS cumpliendo así el paso 2, como se muestra a continuación. En los *escenarios traza* la función de precedencia es un orden total y la función de etiquetado ℓ explícitamente especifica la presencia o ausencia de cada posible evento en cada punto de la traza, retornando lo que se conoce como un *minterm* (una cláusula conjuntiva donde las proposiciones de evento aparecen una única vez, ya sea complementada o sin complementar) sobre el conjunto de eventos disponibles.

Definition 3.3.5 (Escenarios Traza)**.** *Un escenario traza* $\mathcal{S}_\sigma$ *es un escenario* $\langle \Sigma_\sigma, P_\sigma, \ell_\sigma, \equiv_\sigma, \not\equiv_\sigma, <_\sigma, \gamma_\sigma \rangle$ *donde:*

T1: P_σ *y* $<_\sigma$ *satisfacen que* $\exists$ *un índice biyectivo:* $\mathbb{N} \rightarrow P_\sigma$ *tal que indice(i)* $<_\sigma$ *indice(j) si y solo si* $i < j$

T2: $\ell_\sigma(\mathsf{p})$ *retorna un* minterm *sobre* Σ_σ *para todo* $\mathsf{p} \in P_\sigma$*;*

T3: $\gamma_\sigma(\mathsf{p},\mathsf{q})$ = *falso si no existe* w *tal que* $\mathsf{p} <_\sigma \mathsf{w} <_\sigma \mathsf{q}$*;*

T4: $\mathsf{p} \equiv_\sigma \mathsf{q}$ *si y solo si* $\mathsf{p} = \mathsf{q}$ *, para todo* $\mathsf{p},\mathsf{q} \in P_\sigma$*.*

Finalmente, la siguiente definición provee la semántica de FVS (paso 3). La semántica de un conjunto de reglas está dado por el conjunto de todas las trazas que satisfacen R. Formalmente:

Definition 3.3.6 (Semántica de trazas para un conjunto de reglas FVS)**.** *Un escenario traza* $\mathcal{S}_\sigma$*, satisface un conjunto de reglas* R *sí y solo sí* $\forall\ r \in \mathrm{R}\ \mathcal{S}_\sigma \models r$*.*

Notar que las trazas de autómatas tal como fueron definidas en la definición 2.2.2 pueden ser mapeadas inyectivamente en escenarios trazas como sigue.

Definition 3.3.7 (Relacionando trazas de autómatas con escenarios traza)**.** *Dada una traza de autómata* p *se construye el siguiente mapeo a un escenario traza* $\mathcal{S}_\sigma$*:*

- $\Sigma_\sigma = \Sigma$,
- $P_\sigma = \mathbf{N}$, *el conjunto de los números naturales,*
- $\ell_\sigma(i) = \Delta(i)$, *donde* $\Delta(i)$ *se interpreta como un minterm que contiene a la proposición* p *cuando* $\Delta(i)(p)$ = *verdadero y* $\neg$ p *si* $\Delta(i)(p)$ = *falso. Esto es, cada punto será etiquetado de acuerdo a su valuación en* Δ
- $\equiv_\sigma = id_{\mathbf{N}}$ *, la relación de identidad entre los números naturales*
- $\not\equiv_\sigma = \not\equiv_{\mathbf{N}}$ *, la relación de inequidad entre los números naturales*
- $<_\sigma = <_{\mathbf{N}}$ *, la relación* $<$ *("menor qué") entre números naturales*
- $\forall i\ \gamma_\sigma(i, i+1)$= *falso , indicando que no existen puntos entre dos índices consecutivos en un escenario traza.*

Similarmente, es trivial observar que también existe un mapeo embebiendo escenarios trazas en trazas de autómatas. Dados estos mapeos inyectivos se usarán de manera intercambiable en el resto de la tesis los términos escenarios traza y trazas de autómatas.

Capítulo 4

Modelando Patrones de Especificación

En el presente capítulo se modelan todos los patrones de especificación en FVS, incluyendo una extensión a la especificación original (en [43]) presentada en [98].

4.1. Patrones de Especificación en FVS

En esta sección se ilustrará cómo FVS puede describir todos los patrones de especificación especificados en [43]. Inicialmente se trabajará con uno de los patrones más simples y utilizados en la práctica, como es el patrón Respuesta. A continuación se presentará la noción de alcance, siempre a modo de introducción al modelado en FVS de los patrones de especificación. Luego, se especifican en FVS todos los patrones de especificación, presentando primero los patrones en la categoría *Ocurrencia* y luego aquellos en la categoría *Orden*. Finalmente, esta sección concluye analizando una extensión a los patrones de especificación propuesta por [98].

En el patrón respuesta la ocurrencia de un evento (conocido como el evento *Acción* o estímulo) lleva a la ocurrencia de otro evento, denominado *Respuesta*. Este comportamiento ilustra una relación causa-efecto entre los eventos involucrados. Como posibles ejemplos de su utilización se pueden nombrar requerimientos como “Todo pedido por parte del cliente debe ser atendido por el servidor” o “Cada vez que las puertas se abren se deben encender las luces”. La regla en la figura 4.1 modela en FVS el comportamiento de este patrón.

Figura 4.1: El patrón Respuesta modelado en FVS

4.1.1. Modelando alcances

En la sección anterior se modeló lo que el trabajo en [43] define como la *intención* del patrón, es decir, la estructura de su comportamiento. Para completar la especificación de un patrón se agrega a la intención, *su alcance*. El alcance de un patrón establece la porción de la ejecución sobre la cual el patrón debe valer. En [43] se definen cinco posibles alcances:

- **Global**: el patrón debe valer durante toda la ejecución.
- **Antes de P**: el patrón debe valer antes de la primera ocurrencia de un evento delimitador P.
- **Después de Q**: el patrón debe valer luego de la primera ocurrencia de un evento delimitador Q.
- **Después de Q hasta P**: el patrón debe valer después de la ocurrencia de Q pero antes de la ocurrencia de P. No es necesario que ocurra el delimitador final P para que el patrón valga.
- **Entre P y Q**: Similar al alcance anterior. El patrón debe valer después de la ocurrencia de P pero antes de la ocurrencia de Q. Sin embargo, en este caso, ambos delimitadores deben ocurrir para que el patrón tenga validez.

Los alcances se modelan en FVS como cualquier otra restricción de comportamiento. De esta manera, el alcance Global se modela implícitamente al no introducir restricciones extras (todas las reglas mostradas hasta ahora en este trabajo asumían un alcance global). Para completar la noción de alcances, en la figura 4.2 se especifica el patrón Respuesta considerando el resto de los alcances posibles.

Luego de esta introducción al modelado en FVS especificando el patrón Respuesta, se completa la definición de todos los patrones a través de reglas FVS, considerando todos los alcances posibles.

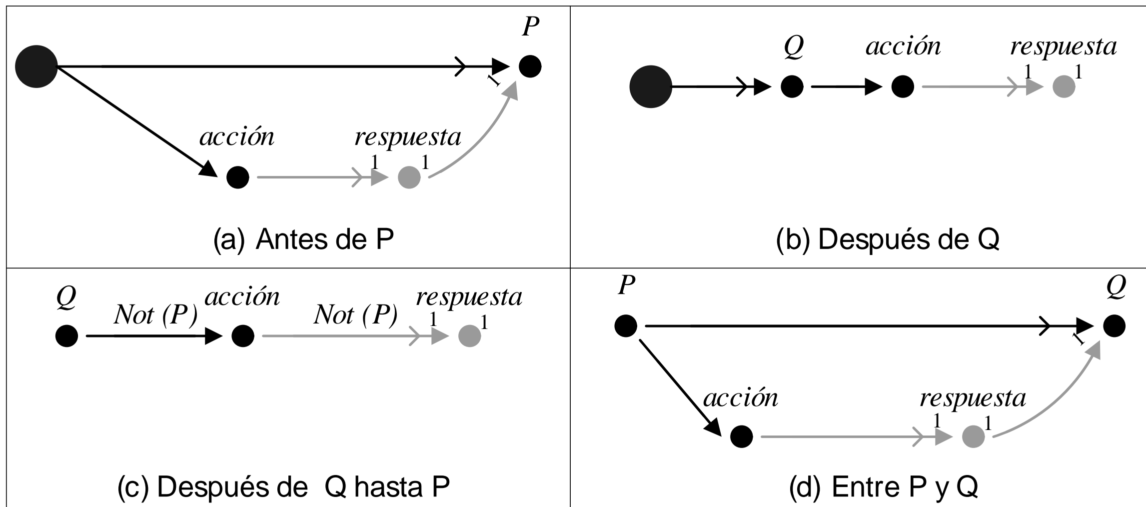

Figura 4.2: El patrón Respuesta bajo distintos alcances

4.1.2. Patrones de Ocurrencia

En las siguientes subsecciones se modelan los cuatro patrones en esta categoría: Ausencia, Universal, Existencia, y Existencia Limitada.

Patrón Ausencia

La intención de este patrón es describir una porción de la ejecución del sistema donde no tiene que ocurrir un dado evento. Uno de los ejemplos más comunes de la aplicación de este evento es la exclusión mutua entre procesos. En este contexto, el alcance del patrón sería el segmento de ejecución en el cual un proceso se encuentra en su sección crítica (es decir, entre que el proceso accede a su sección y sale de la misma), y el evento que no debe ocurrir es la entrada de otro proceso.

Las siguientes reglas en la figura 4.3 describen este patrón considerado cada alcance posible. En estas reglas, se considera al evento F como el evento prohibido, el que no debe ocurrir durante el alcance especificado en el patrón. Esto implica que las trazas válidas del sistema son aquellas que exhiben la ausencia del evento F en el alcance elegido. Por ejemplo, la regla para el alcance Global especifica que el evento prohibido no debe ocurrir entre el comienzo y el final de la traza. Para los demás alcances alcanza simplemente con adaptar esta restricción a los distintos delimitadores. Vale la pena mencionar en este punto que la regla para el alcance *Después de Q hasta P* tiene dos consecuentes, denotando dos posibles comportamientos válidos. Dado que el alcance puede satisfacerse con o sin la

ocurrencia del delimitador P, la especificación debe cubrir ambos casos. En la regla para este alcance en la figura 4.3 se deben cumplir uno de los dos consecuentes especificados: luego de la ocurrencia del delimitador Q deben suceder uno de dos comportamientos posibles: o bien ni el delimitador P ni el evento prohibido F ocurren hasta el final de la traza (comportamiento ilustrado en el consecuente 1), o por el contrario, el delimitador P si ocurre, pero el evento prohibido F no ocurre en el segmento entre Q y P (consecuente 2).

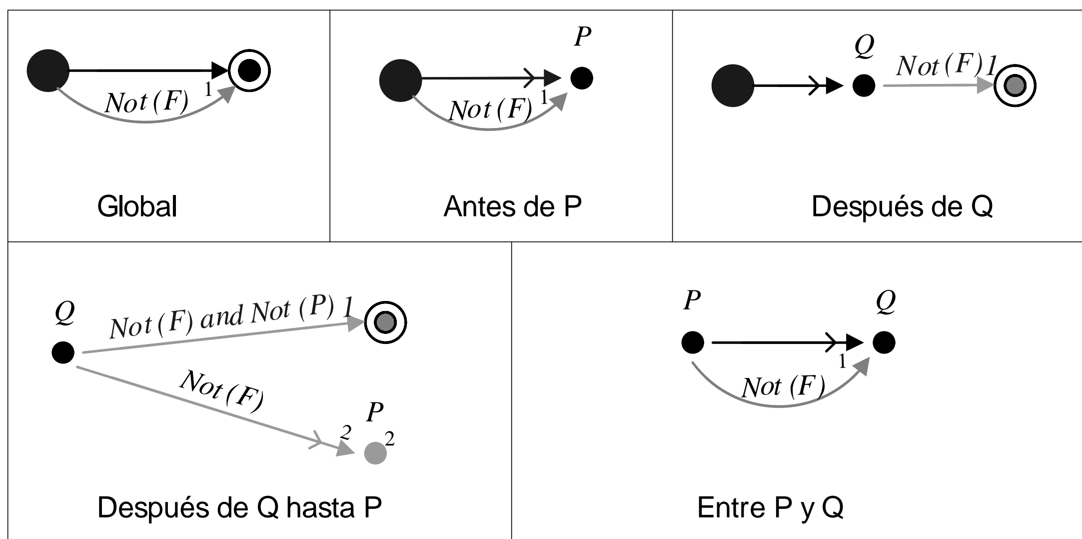

Figura 4.3: El patrón Ausencia en FVS

Patrón Universal

Este patrón es aplicable en aquellos casos donde un evento ocurre a lo largo de un cierto alcance. Según lo definido por [43], este patrón es más apropiado para sistemas basados en estados que para sistemas basados en eventos. En los sistemas basados en estados, este patrón se utiliza para expresar que una cierta propiedad debe ser verdadera en cada estado incluido en el alcance. Para modelar este patrón considerando eventos en vez de estados, se incluye un evento representando el caso donde la propiedad no es válida, es decir, donde la propiedad es falsa. Bajo esta visión, este patrón puede ser formulado para exigir la ausencia de este evento. Luego, este patrón puede modelarse en sistemas basados en eventos utilizando el patrón *Ausencia*. Esta reformulación sigue lo planteado en [43], donde se establece que la "universalidad" de un evento puede entenderse como la ausencia de su negación. Tomando al evento F como el evento que representa el momento

en que la propiedad de interés deja de ser verdadera, las reglas en la figura 4.3 describen por completo este patrón bajo los cinco alcances posibles.

Patrón Existencia

El patrón Existencia, muy cercano al Universal y al Ausencia, intenta describir una porción de la ejecución del sistema que contiene una instancia de un evento dado. Es decir, este patrón se utiliza para aquellos casos donde es requerida la ocurrencia de un cierto evento en un alcance dado. Las reglas FVS en la figura 4.4 refleja el comportamiento de este patrón, donde el evento E representa el evento que debe ocurrir. Las mismas exigen al menos una ocurrencia del evento E dentro de cada alcance.

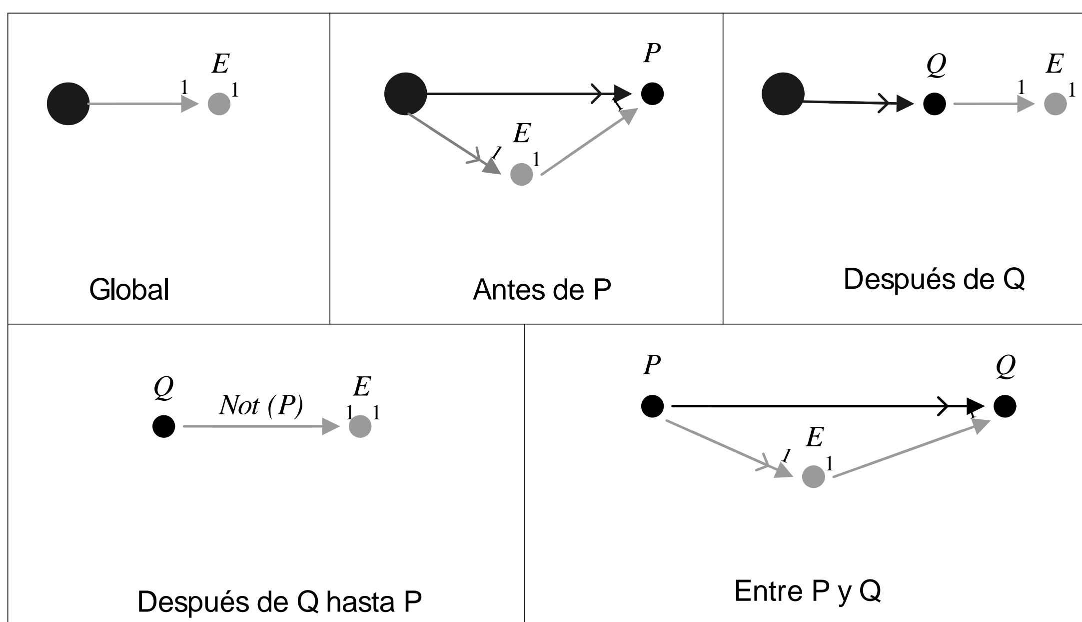

Figura 4.4: El patrón Existencia modelado con reglas FVS

Patrón Existencia Limitada

Este patrón permite describir una porción de la ejecución del sistema que contenga como máximo k ocurrencias de un evento dado. Dado este parámetro k, este patrón también es conocido como el patrón k-Existencia limitada. Dentro de un protocolo de asignación de recursos se puede utilizar este patrón para decir que un cliente puede tener acceso a un recurso como máximo dos veces seguidas mientras esté esperando otro cliente por el mismo recurso. Para modelar este patrón se utilizó un $k = 2$. Otras versiones con

un valor de k mayor son fáciles de intuir a partir del ejemplo dado. Asimismo, el evento E representa el evento cuya existencia se debe acotar.

La figura 4.5 ilustra este patrón considerando cada alcance posible. La regla para el alcance global dice que dada la primer ocurrencia del evento E entonces es el caso que E ocurre una única vez más (consecuente 1), o bien, E no vuelve a ocurrir durante la ejecución (consecuente 2). Para los demás alcances el análisis es similar, teniendo en cuenta que se modifica la porción de ejecución afectada por el patrón. A primera vista la regla que modela el alcance *Después de Q hasta P* parece compleja, al tener cuatro consecuentes posibles. Sin embargo, cada uno cubre el comportamiento básico ya descrito (luego de la primera ocurrencia de E o bien E se repite sólo una vez más o directamente no vuelve a aparecer durante el alcance especificado), pero considerando en este caso dos posibles delimitadores del alcance, ya que P podría no ocurrir. Los consecuentes 1 y 2 cubren los casos donde P ocurre, mientras que los consecuentes 3 y 4 atacan el caso contrario.

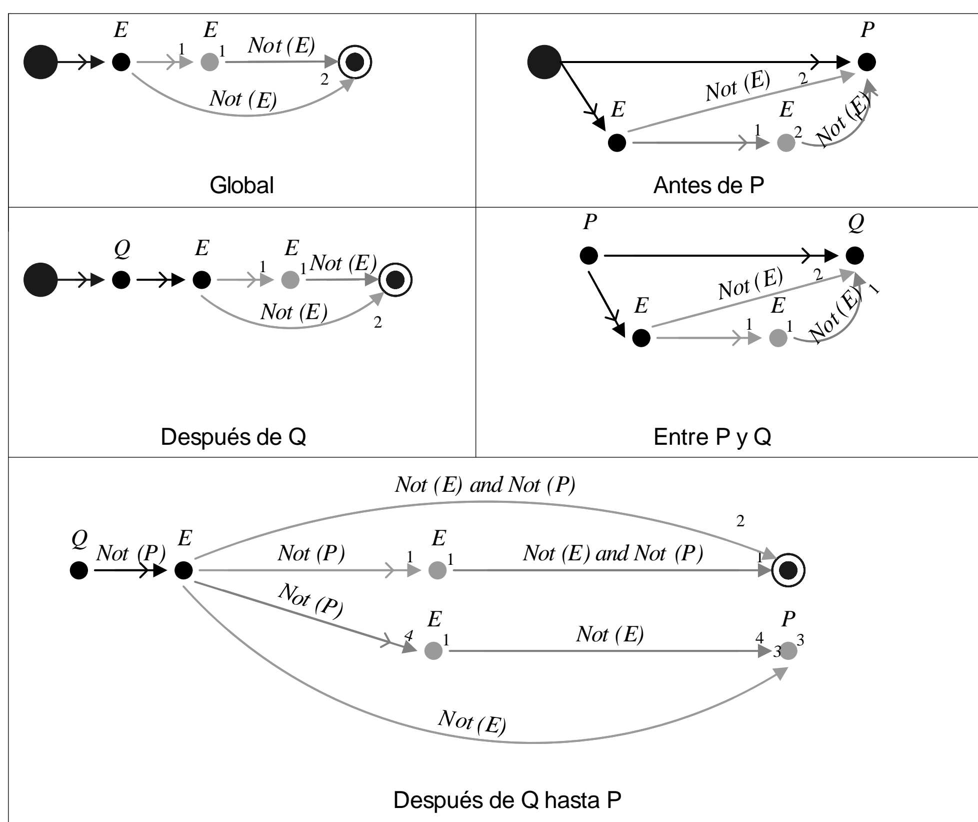

Figura 4.5: El patrón 2-Existencia limitada modelado en FVS

4.1.3. Patrones de Orden

Los patrones en esta categoría se enfocan en el orden relativo en que eventos múltiples ocurren durante la ejecución del sistema. Los cinco patrones en esta categoría son: *Precedencia*, *Respuesta*, *Precedencia Encadenada*, *Respuesta Encadenada*, y el *Encadenamiento Restringido*. A continuación se describen cada uno de ellos a través de reglas FVS.

Patrón Respuesta

Este patrón ya fue descrito por completo en las figuras 4.1 y 4.2, al introducir el modelado en FVS de los patrones de especificación.

Patrón Precedencia

La intención de este patrón es describir la relación entre un par de eventos donde la ocurrencia de uno de ellos toma la forma de una pre-condición necesaria para la ocurrencia del segundo [42]. En este caso, la ocurrencia de un evento *estímulo* (evento S en la figura) debe preceder la ocurrencia de un evento *respuesta* (evento R en la figura). En la figura 4.6 se encuentran las reglas FVS que modelan este patrón bajo todos los alcances posibles.

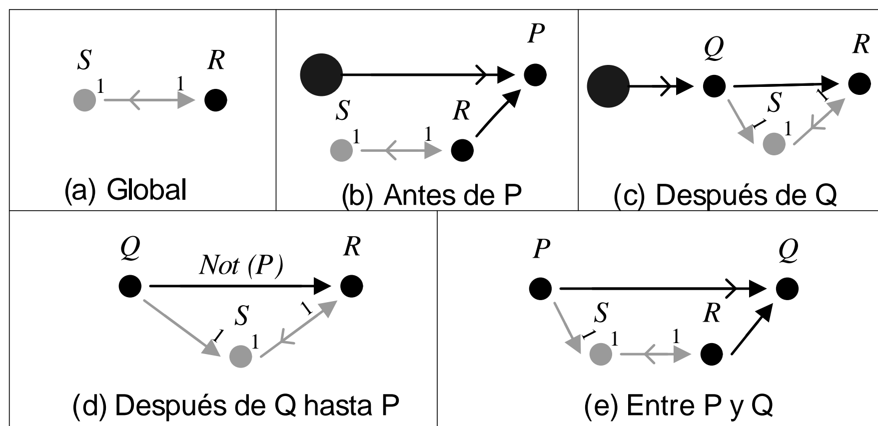

Figura 4.6: El patrón Precedencia en FVS

Precedencia Encadenada

Tanto el patrón Precedencia como el patrón Respuesta pueden generalizarse y así obtener los patrones Precedencia Encadenada y Respuesta Encadenada. En estas versiones

generalizadas contemplan los casos donde en vez de tener un único evento como estímulo y como respuesta, se tienen una secuencia de eventos formando el estímulo y/o la respuesta. En particular, el patrón Precedencia Encadenada se emplea para describir la relación entre dos secuencias de eventos en donde la ocurrencia de la secuencia $R_1, R_2, \ldots, R_n$ en el alcance especificado debe precedida por la secuencia de eventos $S_1, S_2, \ldots, S_n$ dentro del mismo alcance. Siguiendo lo presentado en [42] se modelan a continuación dos instancias posibles de este patrón. En la primera una respuesta R debe ser precedida por dos estímulos *S1* y *S2* (ver figura 4.7) mientras que en la segunda se cubre el caso donde dos respuestas *R1* y *R2* deben siempre ser precedidas por un estímulo S (ver figura 4.8). Otras posibles combinaciones de este patrón se modelan con reglas similares.

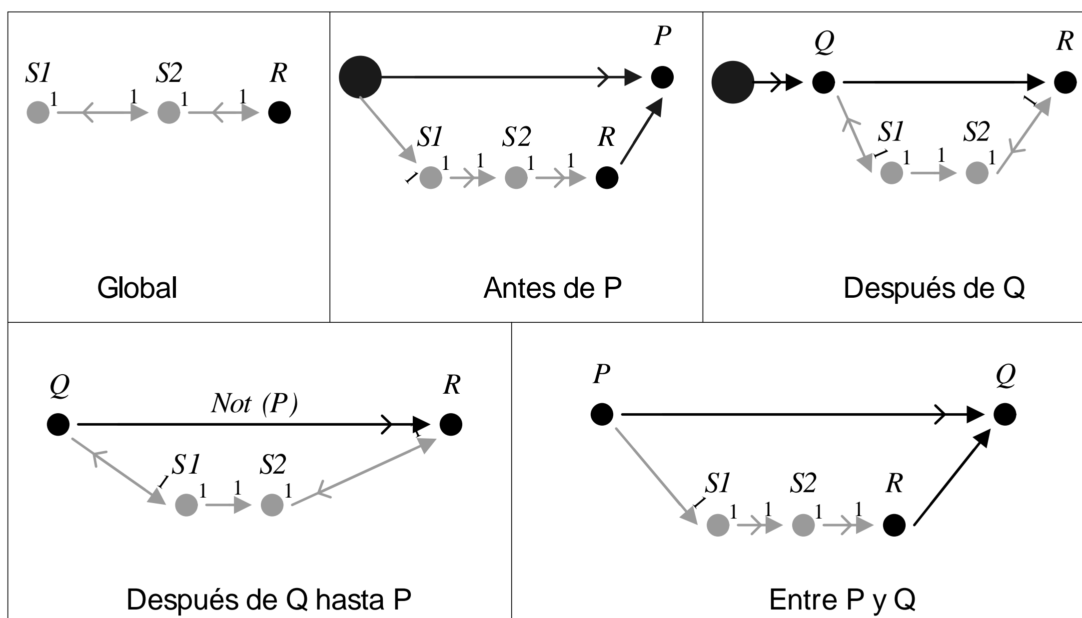

Figura 4.7: Precedencia Encadenada con dos estímulos y una respuesta

Respuesta Encadenada

Ahora se considera la generalización del patrón Respuesta: el patrón *Respuesta Encadenada*. En este patrón la secuencia de eventos $S_1, S_2, \ldots, S_n$ debe ser siempre seguida de otra secuencia de eventos $R_1, R_2, \ldots, R_n$. Nuevamente, se analizarán sólo dos posibles instancias de este patrón. Las reglas en la figura 4.9 consideran el caso donde un estímulo S debe ser seguido por dos respuestas R_1 y R_2, siempre dentro del alcance especificado.

Es importante ver que la única diferencia con las reglas modelando el patrón Respuesta (sería el caso de un estímulo y una respuesta), captado en las figuras 4.1 y 4.2 es la inclusión

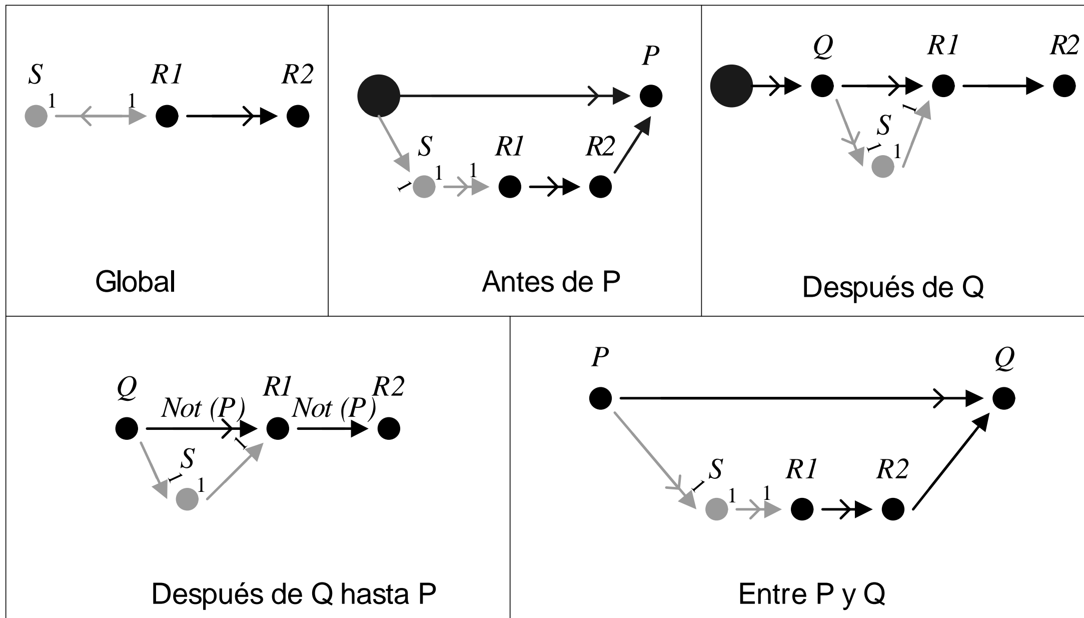

Figura 4.8: Precedencia Encadenada con un estímulo y dos respuestas

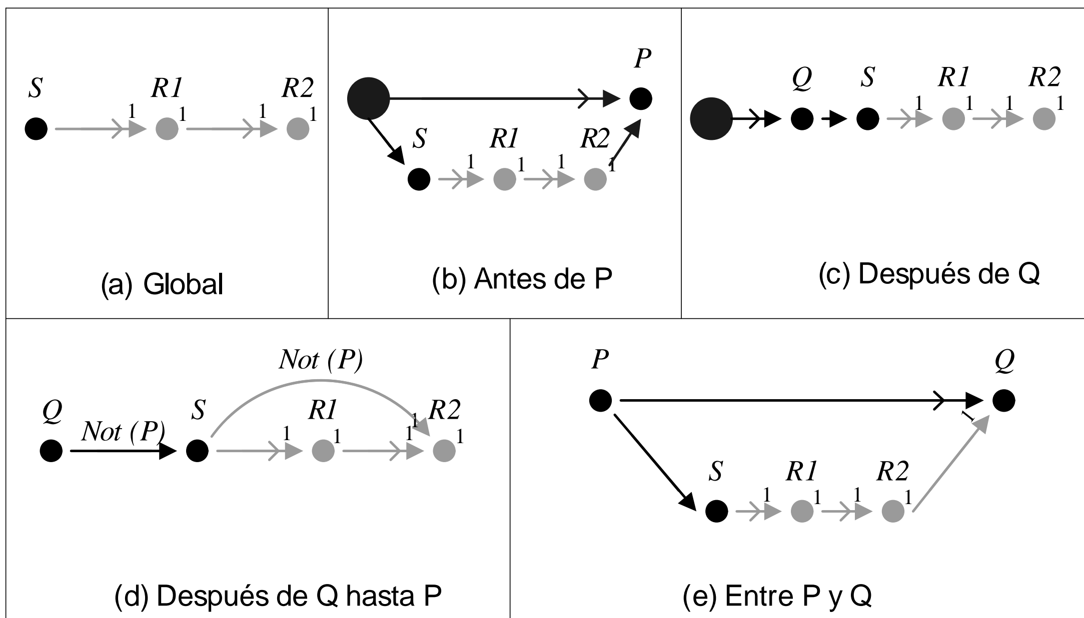

Figura 4.9: El patrón Respuesta Encadenada con un estímulo y dos respuesta

de la segunda respuesta, manteniendo la especificación sucinta y simple, cambiando de la misma manera que cambia la especificación en lenguaje natural.

Una segunda instancia de este patrón se incluye en las reglas de la figura 4.10, donde dos estímulos deben ser seguidos por una única respuesta.

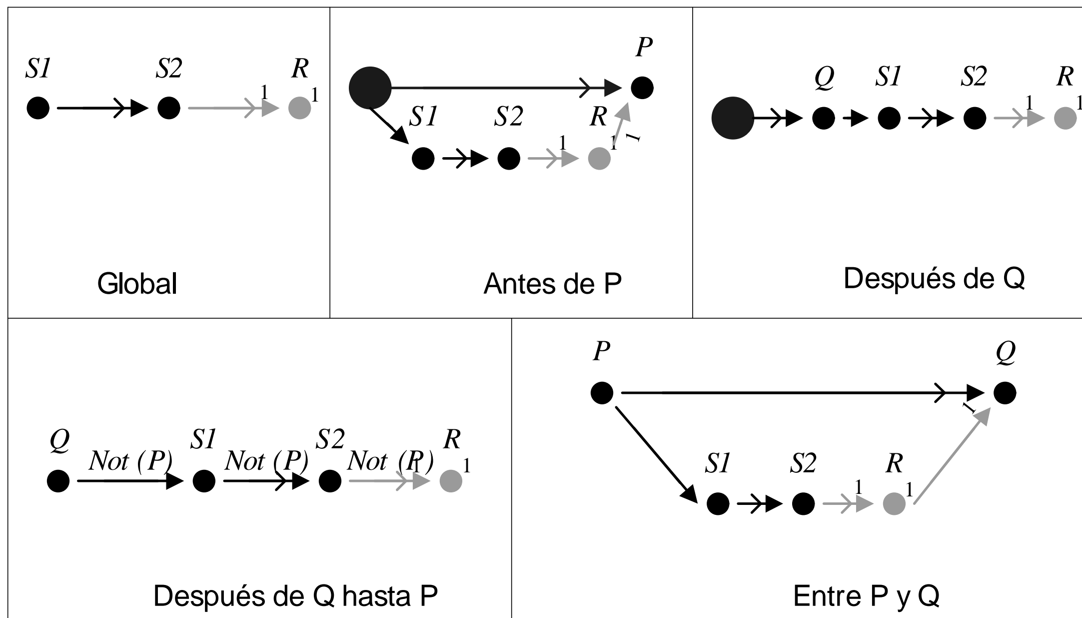

Figura 4.10: El patrón Respuesta Encadenada con dos estímulos y una respuesta

Encadenamiento Restringido

Este patrón constituye una variante de los patrones Respuesta Encadenada y Precedencia Encadenada. En particular, este patrón permite restringir los eventos que pueden ocurrir entre las secuencias de eventos que forman de la cadena. Como se analizó anteriormente, restingir comportamiento es muy simple y directo en FVS, ya que alcanza con unir y etiquetar con flechas los puntos correspondientes. El ejemplo mostrado en la figura 4.11 muestra este patrón en acción, instanciado en un extensión del patrón Respuesta Encadenada con un estímulo y dos respuestas, pero con el adicional considerando ahora el evento W no debe ocurrir en la secuencia de eventos que forma la respuesta para que la propiedad sea satisfecha.

4.1.4. Extensión a la especificación de patrones

El trabajo presentado en [98] plantea algunos problemas en la especificación de patrones presentada en [43]. En particular, mencionan ciertas cuestiones de interés que no están cubiertas en la especificación original. Por ejemplo, tomando el caso del patrón Respuesta, existen aspectos que no están precisados con exactitud. Por ejemplo, ¿puede el evento que constituye el estímulo o acción ocurrir más de una vez antes que ocurra la respuesta? ¿Es aceptable una traza del sistema donde nunca ocurre el evento *acción*? ¿Puede ocurrir

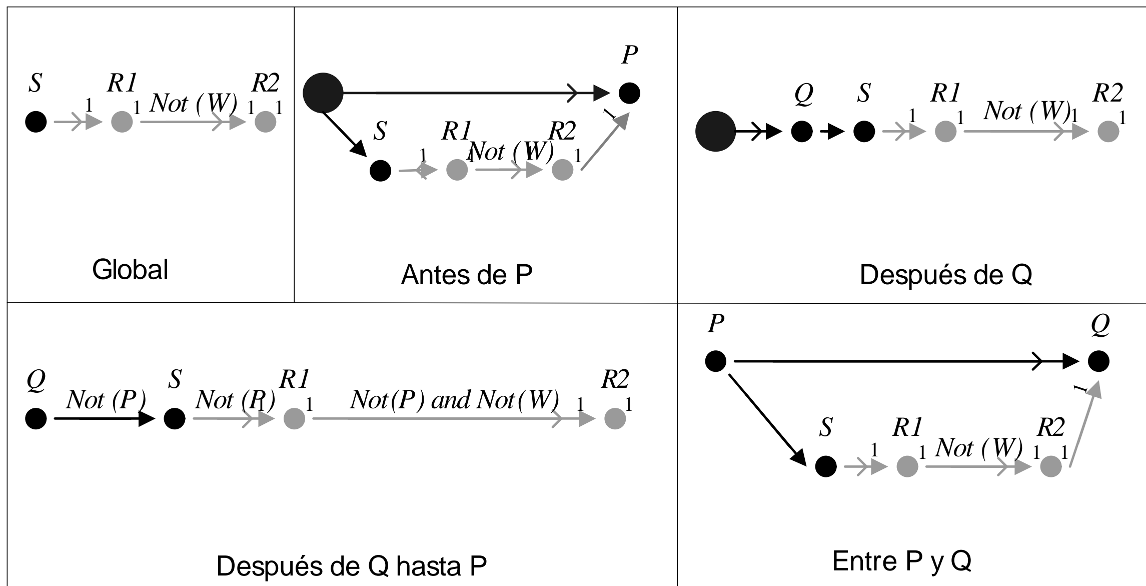

Figura 4.11: Una instancia posible del patrón Encadenamiento Restingido en FVS

más de una vez el evento *respuesta*? ¿Hay eventos que no deben ocurrir entre la acción y su respuesta? Teniendo en cuenta que todas estas interrogantes no están contempladas en la especificación original de patrones, los autores en [98] proponen una extensión a la especificación de patrones, donde se incluyen seis nuevos conceptos:

- **Pre-aridad**: determina si el evento *Acción* ocurre una o más veces antes que ocurra la respuesta,
- **Post-aridad**: determina si el evento respuesta puede ocurrir sólo una vez o muchas luego de la ocurrencia del evento acción.
- **Inmediatez**: determina si pueden o no ocurrir otros eventos entre la acción o estímulo y la respuesta,
- **Precedencia**: determina si la respuesta puede ocurrir antes de la ocurrencia del estímulo.
- **Nulidad**: determina si el evento acción debe ocurrir si o si durante la ejecución del sistema,
- **Repetición**: determina si los eventos acción que ocurren luego de una respuesta deben también ser seguidos por eventos respuesta o no. Es decir, si el comporta-

miento del patrón debe repetirse a lo largo del tiempo o alcanza con ser satisfecho la primera vez.

A continuación se modelan en FVS estos seis puntos, cubriendo la especificación extendida para el patrón Respuesta.

- **Pre-aridad:** La regla en la figura 4.1 modela la situación donde el evento *Acción* puede ocurrir varias veces antes que llegue una respuesta (por defecto en FVS, cualquier evento acción representa un antecedente válido para disparar la regla). Para limitar este comportamiento de manera que la acción no pueda repetirse antes que llegue la respuesta, se agrega una restricción a la regla. Luego, esta restricción prohíbe una nueva ocurrencia del evento acción hasta que ocurra la respuesta. Esto se muestra en la figura 4.12-a.

- **Post-aridad:** La regla en la figura 4.1 también contempla el caso donde el evento respuesta puede repetirse. Si se quiere restringir esta repetición, entonces de debe exigir que entre dos eventos respuesta consecutivos debe ocurrir en el medio un evento acción. De esta manera, las trazas que repitan respuestas sin alternar un evento acción serán descartadas. La figura 4.12-b ilustra este comportamiento.

- **Inmediatez:** Si hubiera eventos a prohibirse entre la acción y la respuesta, los mismos se incluyen simplemente en el etiquetado de la flecha correspondiente. Esto se refleja en la regla de la figura 4.12-c.

- **Precedencia:** este concepto se logra requiriendo que toda ocurrencia de un evento respuesta debe ser siempre precedida por un evento acción, tal como lo muestra la regla en la figura 4.12-d.

- **Nulidad:** el escenario inicial del patrón Respuesta (ver figura 4.1) cubre el caso donde el evento acción no está obligado a ocurrir. Para cambiar este requerimiento y forzar su ocurrencia, se agrega una regla para establecer que ese es el caso: el evento acción debe ocurrir en algún momento desde el comienzo de la traza. La regla en la figura 4.12-e refleja este comportamiento.

- **Repetición:** nuevamente, la regla original del patrón Respuesta en la figura 4.1 permite la repetición del comportamiento marcado en el patrón, ya que requiere

una respuesta para cada evento acción. Si por el contrario se quisiera modelar un comportamiento no repetible, se agrega una nueva regla para contemplar el caso que la propiedad debe cumplirse únicamente para la primera ocurrencia del evento acción, y no para cada una de ellas. Esto está reflejado en la figura 4.12-f. Notar que el antecedente en este caso captura sólo la primer ocurrencia del evento acción, ya que exige que no haya una ocurrencia previa desde el comienzo de la traza. Claramente, sólo la primer ocurrencia satisface esta condición.

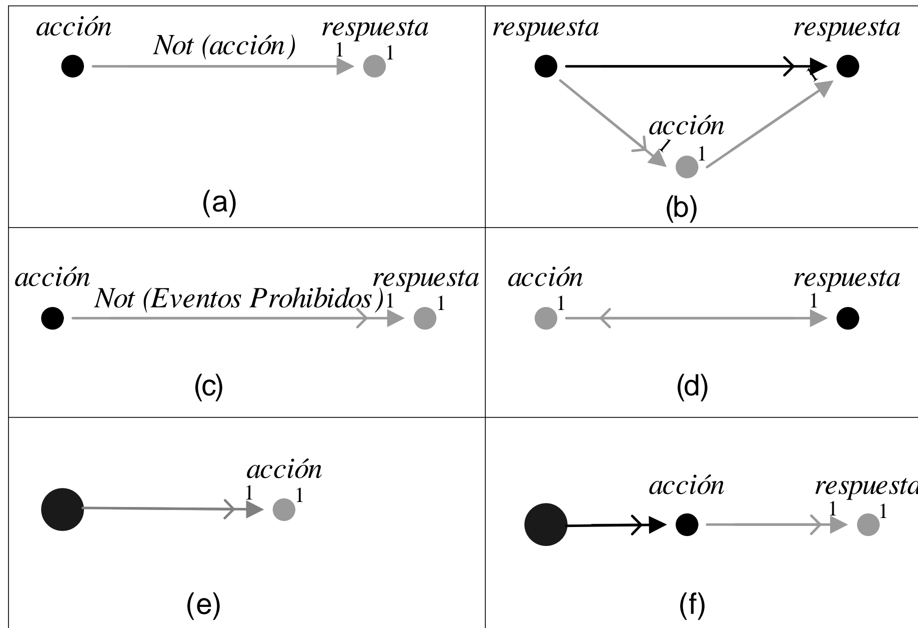

Figura 4.12: Reglas FVS para cubrir la especificación extendida de patrones

4.1.5. Combinando las nuevas propiedades

Las reglas presentadas en la figura 4.12 modelan cada concepto de manera separada. Naturalmente todos pueden combinarse simplemente agregando las reglas necesarias al conjunto de reglas que modelan el comportamiento del sistema. Para ver esto con mayor claridad se presenta a continuación un simple ejemplo. Concretamente, se agregarán de manera incremental a una especificación inicial los conceptos de pre-aridad, post-aridad, inmediatez y nulidad. Se tomará como especificación base el comportamiento modelado con el patrón Respuesta de la figura 3.6(a), donde se dice que todo pedido por parte de un cliente será eventualmente satisfecho. Nuevos requerimientos como asegurar que los pedidos y sus respuestas ocurran de manera intercalada siguiendo el esquema

{pedido,respuesta} pueden agregarse combinando pre y post-aridad, como se ve en las reglas de la figura 4.13. En particular, la regla en 4.13-a establece la pre-aridad, al requerir qué, una vez que ocurre un pedido, ningún otro pedido puede ocurrir hasta que el mismo sea otorgado. De manera similar la regla en la figura 4.13-b establece que entre dos eventos ocurrencias consecutivas de pedido otorgado ocurra en el medio un pedido por parte del cliente, fijando la post-aridad en el comportamiento de la propiedad.

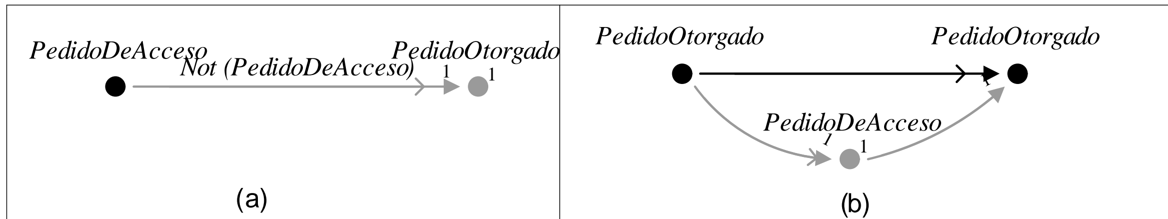

Figura 4.13: Combinando pre y post aridad para el patrón Respuesta

De la misma manera se pueden agregar nuevas restricciones a esta nueva especificación. Usando el concepto de inmediatez se puede modelar un nuevo requerimiento: entre que el pedido es realizado y luego otorgado, no debe ocurrir un evento del tipo *Time-Out*. Esto se ve reflejado en la figura 4.14-a. Notar que esta regla también implica el concepto de pre-aridad descrito anteriormente. Finalmente, el concepto de nulidad lleva a razonar sobre la ocurrencia del evento que representa el pedido del cliente:¿es válida una ejecución del protocolo donde nunca ocurra un pedido por parte del cliente? ¿es este el comportamiento esperado del sistema? Asumiendo que no se desea modelar estos casos sin pedidos por parte del cliente, se agrega una nueva regla para reflejar este nuevo requerimiento descubierto. La regla en la figura 4.14(b) exige que al menos una vez en la ejecución del sistema esté presente un pedido por parte del cliente.

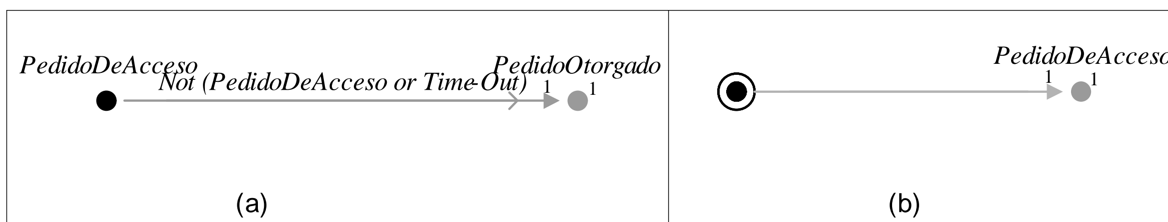

Figura 4.14: Se agregan nuevas restricciones utilizando los conceptos de inmeadiatez y nulidad

El conjunto de reglas en las figuras 4.13-b, 4.14-a y 4.14-b modelan una posible ins-

tanciación de la especificación extendida de patrones propuesta en [98] para el patrón Respuesta. Partiendo de una especificación inicial se agregaron nuevos conceptos cubiertos en la especificación extendida de patrones incluyendo conceptos como pre-aridad, post-aridad, inmediatez y nulidad. Resumiendo, las reglas FVS mostradas para el ejemplo satisfacen las siguiente condiciones que modelan el comportamiento del protocolo:

- Todo pedido por parte del cliente será otorgado.
- El pedido no puede repetirse hasta que haya sido otorgado.
- La respuesa sólo puede ocurrir una vez luego del pedido.
- No puedo ocurrir el evento Time-Out mientras se resuelve el pedido.
- Tiene que ocurrir al menos un pedido

Solo las trazas que satisfagan este conjunto de requerimientos serán consideradas válidas.

Capítulo 5

Comparando Especificaciones

El objetivo de este capítulo es comparar FVS contra otras notaciones tradicionales considerando los cuatro atributos de calidad propuestos: *sucinto*, *comparabilidad*, *complemento* y *modificabilidad*. El caso de estudio consistirá en analizar cómo distintas especificaciones formales de los patrones manejan estos cuatro atributos de calidad. Las notaciones a comparar contra FVS son tres. Por un lado, se analizarán las fórmulas LTL propuestas para cada patrón en [42]. Como segundo formalismo, se tomará los autómatas propuestos por la herramienta Propel en [98]. El tercer formalismo a comparar será la notación en lenguaje natural disciplinado también utilizado en la herramienta Propel. Siempre que sea posible, la comparación será contra los 3 formalismos. La comparación contra las notaciones en Propel son más limitadas ya que la herramienta sólo cubre algunos de los patrones de especificación, como se detalla más adelante. Como extra, al analizar el atributo de *complemento* se describe comportamiento complementario para cada uno de los patrones de especificación.

Finalmente, para proveer más elementos en la comparación, se desarrollará en la sección 5.2 un caso de estudio para ver FVS y otras notaciones en acción completando todo el proceso de especificación de propiedades.

5.1. Comparación de los atributos de calidad

Como se mencionó, las especificaciones FVS serán comparadas con otros 3 formalismos. Para facilitar la comprensión de esta sección se presenta la especificación del patrón Respuesta en todos los formalismos analizados en este trabajo. Primero, se presenta una

caracterización posible del patrón Respuesta siguiendo la notación en lenguaje natural disciplinado de Propel:

- Comportamiento Base (con pre-aridad, post-aridad e inmediatez): Una o más ocurrencias del evento acción resultarán eventualmente en una o más ocurrencias del evento respuesta.
- Frase de Nulidad: el evento acción puede no ocurrir.
- Frase de precedencia: la respuesta puede ocurrir antes que la primera ocurrencia del evento acción.
- Frase de Repetición: el comportamiento es repetible.

En la figura 5.1 se puede apreciar el mismo comportamiento modelado con los demás formalismos involucrados: un autómata, una fórmula LTL y como una regla FVS.

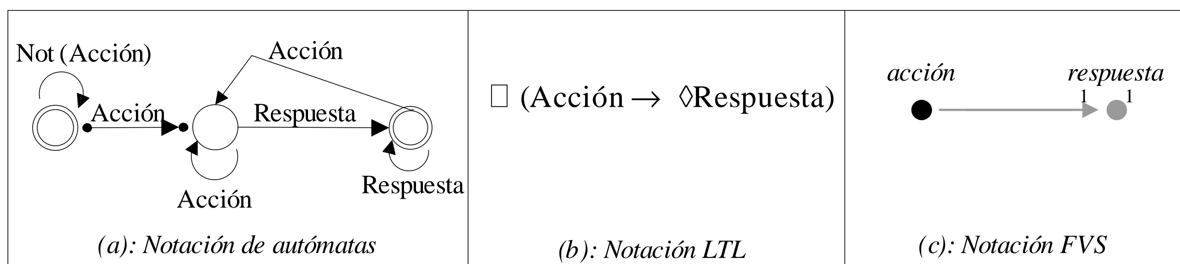

Figura 5.1: El patrón respuesta modelado bajo 3 notaciones diferentes

A continuación se procede a comparar las reglas FVS con otros formalismos bajo los atributos de calidad propuestos.

5.1.1. Sucinto

Una forma de analizar lo sucinto de las especificaciones es analizar cómo las mismas crecen a medida que se agregan las restricciones de los alcances en la especificación de los patrones, denotando así comportamiento cada vez más complejo y específico. Para poder comparar FVS contra las fórmulas LTL dadas en [42] y las notaciones de la herramienta Propel se propone la siguiente medida para estimar de alguna forma el tamaño de los artefactos expresados en los formalismos. Para las fórmulas LTL se tendrán en cuenta el nivel de anidamiento (N) y el número de operadores involucrados (O). Para la notación

de autómatas se contarán la cantidad de estados (St) y el número de transiciones (T) de los autómatas presentados en [98], y finalmente, para la notación en lenguaje disciplinado de Propel se contabilizará el número de frases (S) necesarias para expresar la propiedad, mientras que para las reglas FVS se medirán la cantidad de puntos involucrados (P) y el número de restricciones utilizadas (R), contando precedencia y restricción de comportamiento.

Como primer análisis se comparan las especificaciones del patrón Respuesta, bajo todos los alcances posibles. Los resultados se muestran en la tabla 5.1, donde las filas representan formalismos, y las columnas los distintos alcances. La primera fila corresponde al tamaño de la fórmula LTL para el patrón según la medida de tamaño definida. La segunda fila corresponde a la notación de lenguaje natural disciplinado y la tercera a la notación de autómatas. Ambas corresponden a la herramienta Propel. Finalmente, la cuarta columna muestra el tamaño de las especificaciones FVS. Las entradas en la columnas representan los alcances: Global para el alcance Global, *Antes* para el alcance *Antes de P*, *Después* para el alcance *Después de Q*, *Entre* para el alcance *Entre P y Q*, y *Hasta* para el alcance para *Después de Q hasta P*.

Tabla 5.1: Comparación del tamaño de especificaciones para el patrón Respuesta

Formalismo	Global	Antes	Después	Entre	Hasta
LTL	N=1,O=3	N=2,O=5	N=3,O=5	N=4,O=1	N=4,O=10
Automata	St=3,T=6	St=4,T=8	St=4,T=9	St=5,T=12	St=5,T=12
DNL	S=7	S=8	S=8	S=10	S=10
FVS	P=2,R=1	P=4,R=4	P=4,R=3	P=4,R=4	P=3,R=4

Como lo muestra la tabla 5.1 las fórmulas LTL dadas por [42] codificando los patrones crecen de manera significativa al agregarse alcances complejos. Los autómatas también crecen de manera importante en tamaño, en particular por el aumento en el número de transiciones. Por otro lado, la notación en lenguaje natural disciplinado tiene un crecimiento aceptable, es decir, que aumenta de manera proporcional al cambio que sufre en lenguaje natural la especificación original al agregarse los alcances. Sin embargo, esta notación tiene algunas limitaciones propias de la técnica respecto a la introducción de alcances con delimitadores. Los delimitadores deben ser distintos, y no tiene que haber

intersección entre los eventos que definen la intención y los que definen al alcance del patrón. En las reglas FVS los alcances pueden agregarse sin que el aumento del tamaño de las mismas sea relevante. Adicionalmente, la notación FVS no cuenta con ninguna limitación respecto a los eventos que forman tanto la intención como el alcance de la propiedad.

Ahora se comparan la codificación de todos los patrones de especificación en la categoría *Ocurrencia*. En esta ocasión FVS se comparará solamente contra las formulas LTL dadas en [42]. Esto se debe a que la codificación en Propel de estos patrones no está disponible [98]. La tabla 5.2 muestra el crecimiento de las fórmulas LTL mientras que la tabla 5.3 lo hace para las reglas FVS. Cada fila se corresponde con un patrón y cada columna se corresponde con un alcance. La entrada en la cuarta fila de ambas tablas etiquetada con 2-limitada se refiere al patrón 2-Existencia limitada.

Tabla 5.2: Crecimiento fórmulas LTL para patrones de ocurrencia

Patrón	Global	Antes	Después	Entre	Hasta
Ausencia	N=1,O=2	N=1,O=4	N=2,O=4	N=2,O=8	N=2,O=6
Existencia	N=1,O=1	N=1,O=4	N=2,O=6	N=3,O=8	N=3,O=8
2-Limitada	N=4,O=8	N=8,O=22	N=6,O=13	N=10,O=24	N=9,O=24
Universal	N=1,O=1	N=1,O=3	N=2,O=3	N=2,O=7	N=2,O=5

Tabla 5.3: Crecimiento reglas FVS para patrones de ocurrencia

Patrón	Global	Antes	Después	Entre	Hasta
Ausencia	P=2,R=1	P=4,R=4	P=4,R=4	P=4,R=4	P=3,R=4
Existencia	P=2,R=1	P=3,R=3	P=3,R=2	P=3,R=3	P=2,R=2
2-limitada	P=4,R=6	P=4,R=7	P=5,R=7	P=4,R=7	P=6,R=12
Universal	P=2,R=1	P=2,R=1	P=3,R=2	P=2,R=1	P=3,R=3

El análisis de las tablas 5.2 y 5.3 permite observar un importante crecimiento en las fórmulas LTL, especialmente al introducir alcances complejos cómo *Entre P y Q* o *Después de Q hasta P*. Sin lugar a dudas, el patrón 2-Existencia limitada es el más complejo de

todos. Las fórmulas LTL que lo codifican no sólo tienen muchos operadores y nivel de anidamiento para los alcances compuestos como *Entre P y Q* o *Después de Q hasta P* sino también para alcances más sencillos como el *Antes de P*, donde se propone una fórmula con 22 operadores y 8 niveles de anidamiento. Esto claramente es grave, máxime cuando estos números tienden a aumentar considerablemente cuando se consideran $k > 2$. Por otro lado, como se ve en la tabla 5.3 las reglas FVS también aumentan su tamaño, pero en una proporción mucho menor. Esto es un indicador que FVS podría manejar mejor cuestiones es escalabilidad, facilitando la especificación de patrones como el k-Existencia limitada, con valores de $k > 2$.

Finalmente, se completa el análisis del atributo *Sucinto* considerando la codificación de los patrones en la categoría *Orden*. Nuevamente, se muestra primero el crecimiento de las fórmulas LTL en la tabla 5.4 mientras que la tabla 5.5 lo hace con las reglas FVS. Como en las tablas anteriores, no se muestran la codificación en Propel ya que las mismas no están disponibles por los autores de la herramienta. En ambas tablas las entradas P-Encadenada (x,y) y R-Encadenada (x,y) representan tanto los patrones Precedencia Encadenada y Respuesta Encadenada donde x representa la cantidad de eventos que forman el estímulo e y representa la cantidad de eventos que forman la respuesta. Similarmente, la entrada E-Restringido representa al patrón Encadenamiento Restringido. El análisis del crecimiento mostrado en ambas tablas en similar al hecho previamente. Las especificaciones FVS cumplen mejor el atributo *Sucinto*, manteniendo sus especificaciones escalables y simples.

Tabla 5.4: Crecimiento fórmulas LTL para los patrones de orden

Patrón	Global	Antes	Después	Entre	Hasata
Respuesta	N=1,O=3	N=3,O=8	N=2,O=5	N=4,O=12	N=4,O=10
Precedencia	N=0,O=2	N=2,O=5	N=2,O=7	N=3,O=9	N=3,O=7
P-Encadenada(2,1)	N=3,O=10	N=4,O=11	N=4,O=16	5=3,O=13	N=6,O=13
P-Encadenada(1,2)	N=2,O=7	N=2,O=13	N=3,O=13	N=3,O=15	N=4,O=19
R-Encadenada(2,1)	N=3,O=9	N=3,O=13	N=4,O=12	N=4,O=15	N=5,O=24
R-Encadenada(1,2)	N=2,O=6	N=4,O=12	N=3,O=7	N=5,O=14	N=4,O=19
E-Restringido(2,1)	N=3,O=9	N=4,O=16	N=4,O=10	N=6,O=18	N=5,O=25

Tabla 5.5: Crecimiento FVS para patrones de orden

Patrón	Global	Antes	Después	Entre	Hasta
Respuesta	P=2,R=1	P=4,R=4	P=4,R=3	P=4,R=4	P=3,R=4
Precedencia	P=2,R=1	P=4,R=4	P=4,R=4	P=4,R=4	P=4,R=4
P-Encadenada(2,1)	P=3,R=2	P=5,R=5	P=5,R=5	P=5,R=5	P=4,R=5
P-Encadenada(1,2)	P=3,R=2	P=5,R=5	P=5,R=5	P=5,R=5	P=4,R=6
R-Encadenada(2,1)	P=3,R=2	P=5,R=5	P=5,R=4	P=5,R=5	P=4,R=6
R-Encadenada(1,2)	P=3,R=2	P=5,R=5	P=5,R=4	P=5,R=5	P=4,R=6
E-Restringido(2,1)	P=3,R=3	P=5,R=6	P=5,R=5	P=5,R=6	P=4,R=7

Una interesante observación surge de considerar las diferentes variantes para los patrones de Respuesta Encadenada y Precedencia Encadenada. Sólo agregar un estímulo o una respuesta causa un crecimiento notable en las fórmulas LTL, lo cual es un serio problema para encarar tareas de validación. Y el problema se profundiza al considerar alcances como *Entre P y Q* o *Después de Q hasta P*. Por el contrario las reglas FVS mantienen un crecimiento controlado. Pequeños cambios como agregar un estímulo o una respuesta, o alcances complejos llevan a aumentos de tamaño también pequeños. Para concluir esta sección es válido mencionar que las fórmulas LTL pueden ser expresadas de manera más natural o de manera más sucinta empleando modalidades como el manejo del tiempo pasado, u operadores como "De ahora en más"("from now on" en inglés) [70, 75], pero tales fórmulas requieren manipulaciones no triviales o incluso exponerse a problemas como explosiones de estado durante el proceso de manipulación [88].

5.1.2. Comparabilidad

Comparabilidad se refiere a la habilidad de considerar dos codificaciones en lenguajes formales de dos patrones de especificación relacionados y entender las diferencias y similitudes entre ellos, así como poder determinar si uno está embebido en el otro. Estos objetivos son difíciles de cumplir cuando se comparan dos fórmulas LTL complejas, o al comparar autómatas con varios estados y transiciones. Formalmente esto requiere de testear inclusión de lenguajes en el caso de autómatas o utilizar mecanismos deductivos de

simplificación para fórmulas lógicas. La situación es diferente al analizar especificaciones FVS. Por ejemplo, se puede observar que la regla para el patrón Respuesta Encadenada es la extensión lógica de la regla modelando el patrón Respuesta (lo único que cambia entre ambas reglas es la inclusión de los nuevos eventos en el estímulo o en la respuesta), y este análisis puede realizarse visualmente, con sólo mirar las reglas, y no requiere de otras herramientas. Otro ejemplo interesante puede apreciarse en la regla para el patrón de Encadenamiento Restringido (ver figura 4.11). La única diferencia entre la versión con restricciones y la versión sin restricciones (en la figura 4.9) es clara y salta a la vista al comparar ambas reglas: la inclusión de la restricción etiquetando la flecha correspondiente.

Las especificaciones en FVS también permiten lograr razonamientos más elaborados. Para mencionar un ejemplo, considerar la regla para el patrón Respuesta Encadenada con alcance *Antes de P* de la figura 4.9-b y la regla para el patrón Encadenamiento Restringido también con mismo alcance de la figura 4.11-b. Ambas reglas tienen antecedentes equivalentes, pero el consecuente de la figura 4.11-b es más "fuerte" que el consecuente de la regla en 4.9-b, ya que posee más restricciones. Por lo tanto, se puede afirmar que la regla en la figura 4.11-b es una especialización de la regla en la figura 4.9-b. Como se mencionó anteriormente en el presente trabajo, la relación de especialización es un concepto similar a la subsunción lógica [28]. Otros dos reglas en las que puede observarse que también se cumple esta relación es con las reglas describiendo el mismo patrón, pero con alcance Global (ver figuras 4.11-a y 4.9-b). Al mismo tiempo, al comparar las reglas de las figuras 4.9-a y 4.9-b, se puede observar que a pesar que el consecuente de la figura 4.9-b impone más restricciones que el de la figura 4.9-a, también acontece lo mismo con el antecedente. Luego en este caso entre ambas reglas no hay una relación de especialización. Por otro lado, este tipo de análisis es difícil de llevar a cabo al tener notaciones formales en forma de autómatas o lógicas LTL sin una manipulación cuidadosa. Por ejemplo, las fórmulas lógicas para los patrones de las reglas 4.9-a y 4.9-b que se presentan en [42] son: $(\Box S1 \rightarrow \Diamond(R1 \wedge X\Diamond R2))$, para la regla en la figura 4.9-a, y $\Box((P \wedge \Diamond Q) \rightarrow (S1 \rightarrow (\neg QU(R1 \wedge \neg Q \wedge X(\neg QUR2))))UQ)$, para la regla en la figura 4.9-b. Con sólo mirar a ambas fórmulas es más que difícil reconocer y entender la relación semántica entre ambas (si hubiera alguna) y además, lograr entender porqué existe (o porqué no existe en caso de no haber relación).

Se puede llegar a conclusiones similares al analizar las reglas que modelan un patrón

bajo los diferentes alcances. Considerar por ejemplo, las reglas que modelan al patrón Precedencia en la figura 4.6. La estructura principal del patrón, la cual sería que la ocurrencia de un evento estímulo debe siempre preceder a toda ocurrencia de un evento respuesta, está embebida en todas los alcances, y esto es algo que puede observarse a simple vista. En otras palabras, la estructura del patrón se mantiene siempre visible a pesar de que van cambiando los diferentes alcances. Este hecho tiene un impacto positivo a la hora de entender y razonar sobre la propiedad, y decidir cuál es el alcance requerido. En otras especificaciones como las fórmulas LTL en [42] no es tan claro visualizar la estructura principal del patrón, ya que las mismas se vuelven más complejas introduciendo operadores y anidamiento, tal como se discutió previamente en este trabajo.

5.1.3. Complemento

Este atributo de calidad se refiere a la habilidad de un lenguaje formal de generar escenarios generales, expresados en el mismo lenguaje, que expresen el comportamiento que lleva a violar una propiedad. FVS provee la capacidad de razonar por comportamiento complementario ya que para cada regla es posible construir de manera automática anti-escenarios. La figura 5.2 refleja dos anti-escenarios para el patrón Respuesta con alcance *Entre P y Q* (visto en las figura 4.2-d). La regla en la figura 5.2-a modela el caso donde la respuesta nunca ocurre en la traza luego del evento estímulo, mientras que en la figura 5.2-b, el evento respuesta sí ocurre, pero luego de la ocurrencia del delimitador Q.

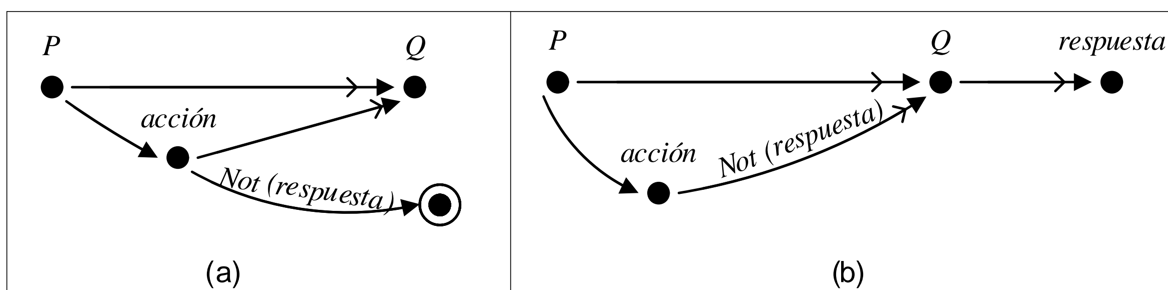

Figura 5.2: Anti-escenarios para el patrón Respuesta con alcance Entre P y Q

En lo que sigue de esta sección se mostrarán anti-escenarios para el resto de los patrones, considerando alcance *Global*. Dada la relación dual entre el patrón *Ausencia* y el patrón *Universal*, sólo se presentan anti-escenarios para el primero de ellos. El anti-escenario para el patrón *Ausencia* es un escenario muy simple. El mismo muestra la

ocurrencia del evento F, tal como lo refleja la figura 5.3).

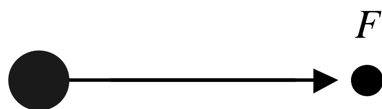

Figura 5.3: Anti-escenario para el patrón Ausencia

De manera similar, un anti-escenario para el patrón *Existencia* es un escenario donde nunca ocurre el evento E durante toda la traza (ver figura 5.4).

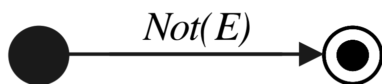

Figura 5.4: Anti-escenario para el patrón Existencia

Para el patrón *2-Existencia limitada* un escenario que representa una violación de esta propiedad es aquel donde el evento E ocurre al menos tres veces (ver figura 5.5).

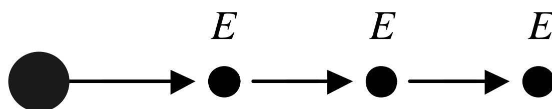

Figura 5.5: Anti-escenario para el patrón 2-Existencia limitada

Para completar la especificación de comportamiento complementario de los patrones, se presentan a continuación anti-escenarios para los patrones en la categoría *Orden*. Para el patrón *Precedencia* un escenario representando una violación consiste en un escenario donde ocurre un evento respuesta R sin que antes ocurriera un evento estímulo S. Tal escenario está reflejado en la figura 5.6.

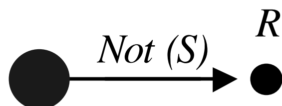

Figura 5.6: Comportamiento complementario para el patrón Precedencia

Para el patrón Precedencia Encadenada con dos estímulos (S1 y S2) y una respuesta (R) existen cuatro anti-escenarios posibles, tal como lo reflejan los escenarios en la figura 5.7. En el primero de ellos (figura 5.7-a), ocurre una respuesta, pero ninguno de los

estímulos. Los escenarios siguientes reflejan la situación donde uno de los estímulos no ocurre (el estímulo *S1* en la figura 5.7-b y el estímulo *S2* en la figura 5.7-c). Finalmente, en el cuarto anti-escenario ambos estímulos suceden, pero en el orden incorrecto. En la figura 5.7-d se observa este escenario.

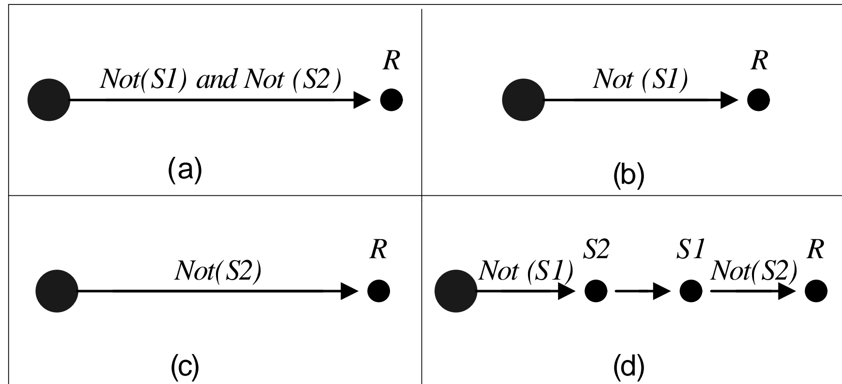

Figura 5.7: Comportamiento complementario para el patrón Precedencia Encadenada

De manera análoga, cuatro anti-escenarios se presentan en la figura 5.8 para el patrón Respuesta Encadenada con un estímulo y dos respuestas, representando situaciones sonde el escenario es violado: no ocurre ninguna respuesta (5.8-a), ocurre únicamente una respuesta (5.8-a y 5.8-a-c), o ocurren ambas pero en el orden incorrecto (5.8-d)

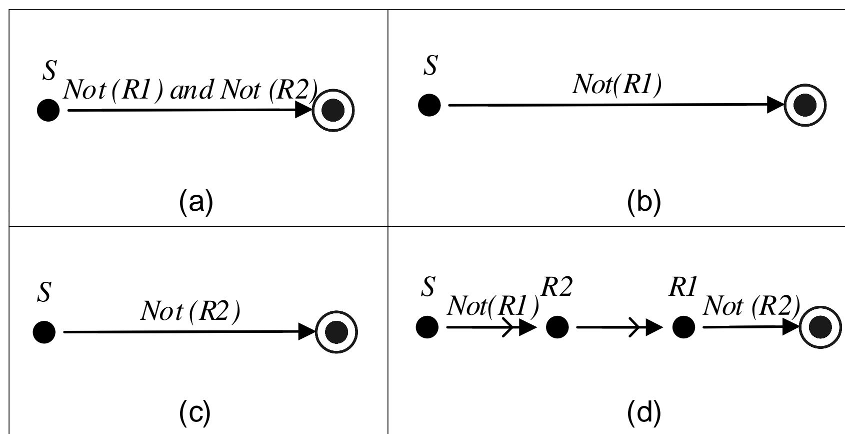

Figura 5.8: Anti-escenarios para el patrón Respuesta Encadenada

Finalmente, se presenta comportamiento complementario para el patrón Encadenamiento Restringido a través de los escenarios de la figura 5.9, en donde el evento W no debe ocurrir durante la secuencia de eventos que forman la respuesta. En este caso existen cinco anti-escenarios. Los primeros cuatro siguen un razonamiento similar al de los patrones de Respuesta Encadenada y Precedencia Encadenada. El quinto escenario en la figura

5.9-e representa un caso adicional, donde ambas respuestas ocurren en el orden correcto pero está presente el evento W entre ellas.

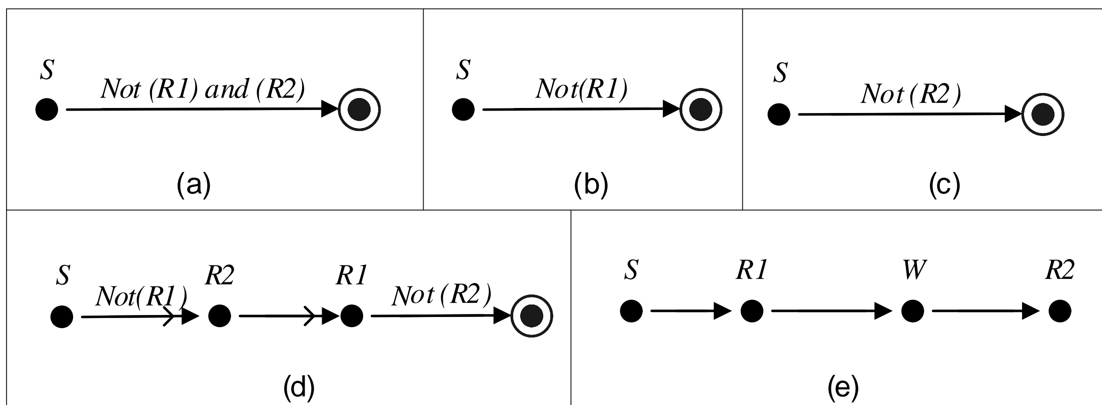

Figura 5.9: Anti-escenarios para el patrón Encadenamiento Restringido

El comportamiento complementario puede obtenerse también en otras notaciones. Para un autómata hay que recurrir a la operación de complemento mientras que para una fórmula LTL se debe obtener su negación. Sin embargo, ambos procesos no son triviales, e involucran operaciones costosas. Asimismo, no es sencillo razonar sobre el resultado obtenido. El artefacto que devuelve una herramienta al complementar un autómata o una fórmula es otro autómata o fórmula. Con sólo mirar la negación de la fórmula no es sencillo deducir las posibles situaciones que llevan a satisfacerla, violando la propiedad original. Lo mismo sucede con el complemento de un autómata.

5.1.4. Modificabilidad

El espíritu de este atributo es analizar el impacto que tienen los cambios en las especificaciones. Se busca que pequeños cambios en el contexto de aplicación lleven a cambios menores y bien localizados en las especificaciones.

Por ejemplo, un punto interesante de observar en la especificación del patrón Respuesta Encadenada es que la secuencia de eventos debe seguir un orden estricto. En el ejemplo previamente desarrollado con un estímulo *S1* y dos respuestas *R1* y *R2*, para que la propiedad valga la respuesta *R1* debe preceder a la respuesta *R2* (ver figura 4.9-a). Se podría dar el caso de querer relajar esta última condición, y permitir que las repuestas ocurran en cualquier orden. Esta modificación es fácil de llevar a cabo en FVS. El único elemento a cambiar de la especificación original es la eliminación de la relación de precedencia entre

ambas respuestas. La regla en la figura 5.10-a muestra la versión modificada del patrón Respuesta Encadenada, contemplando el cambio en los requerimientos.

Como segundo ejemplo, considerar un nuevo cambio sobre el patrón Respuesta Encadenada, con alcance *Entre P y Q*. En esta ocasión, se desea permitir que las respuestas puedan ocurrir también después del delimitador Q, mientras que en la especificación original se exigía que ambas ocurrieran antes de Q (ver figura 4.9-e). Con el cambio la semántica deseada es la siguiente: toda ocurrencia de un estímulo *S1* entre los eventos P y Q siempre debe ser seguido por dos respuestas *R1* y *R2*. Esta nueva versión es aplicable, por ejemplo, para modelar *condiciones de carrera*, una situación típica en sistemas concurrentes o con múltiples *threads*. La regla en la figura 5.10-b refleja esta nueva variante del patrón.

Algo para destacar sobre estos dos ejemplos es que los cambios tuvieron un impacto mínimo, ya que con pocas modificaciones se logró adaptar el patrón a nuevas situaciones. Además, dichos cambios tienen correlación directa con los cambios en los requerimientos, y este punto es fácil de distinguir con sólo ver las especificaciones.

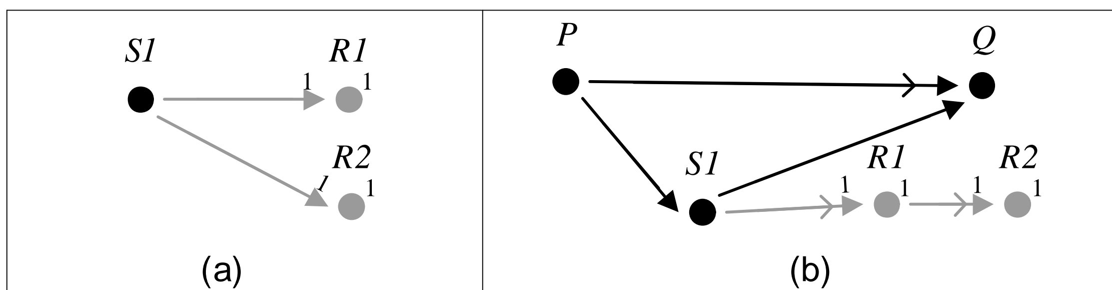

Figura 5.10: Dos variantes para el patrón Respuesta Encadenada

Finalmente se introduce un ejemplo enfocado en el patrón Encadenamiento Restringido, donde un evento W no debe ocurrir durante la secuencia de eventos que forman la respuesta. En esta variante, se modela una restricción más fuerte, y se pide que W no ocurra desde la ocurrencia del estímulo. Las reglas de la figura 5.11 muestran esta variante bajo todos los alcances posibles. Nuevamente, es importante observar que la única diferencia con la especificación original (ver figura 4.11) es simplemente la inclusión de la restricción *Not(W)* en la flecha de precedencia que conecta el estímulo *S1* con la primer respuesta *R1*.

Algo en común que revelan estos tres ejemplos es que las especificaciones FVS de los

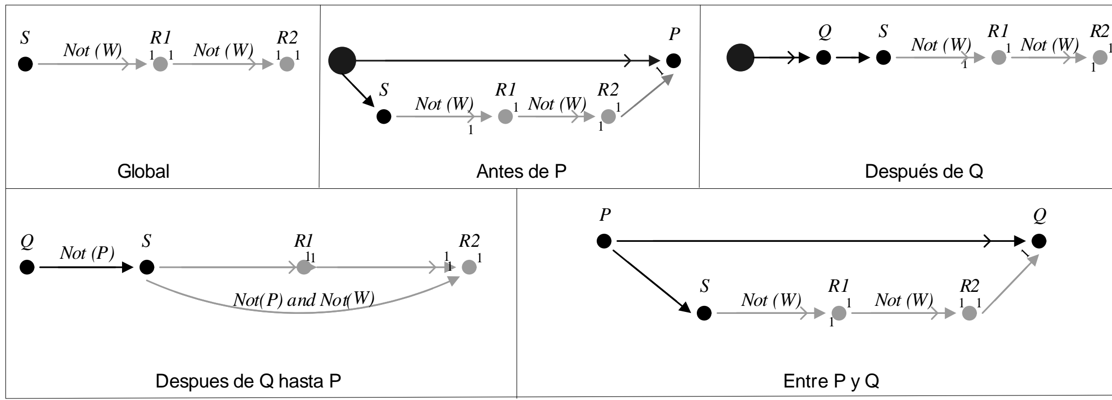

Figura 5.11: Una variante del patrón Encadenamiento Restringido

patrones pueden ser modificadas con facilidad y así poder adaptarse a distintas situaciones y contextos. Con una manipulación adecuada se pueden llevar a cabo estos cambios a autómatas o fórmulas LTL, pero este proceso puede llegar a convertirse en una tarea compleja y propensa a errores.

5.2. Propuesta en acción

El objetivo de esta sección se mostrar cómo se lleva a cabo la tarea de escribir y validar propiedades empleando tanto FVS como otras notaciones formales y poder comparar los distintos procesos. Como caso de estudio se utilizará un ejemplo muy simple y atractivo presentado en [53], donde se describen dificultades típicas que surgen al utilizar notaciones formales y patrones de especificación. El ejemplo se analizará haciendo foco en el patrón Respuesta. Los autores en [53] también consideran el patrón Ausencia, pero se encuentran problemas similares.

El objetivo principal es especificar, en un lenguaje formal, la siguiente propiedad de un sistema telefónico: *Cuando el cliente levanta el teléfono, siempre se genera la señal de tono*. El usuario podría detectar este comportamiento como una posible instancia del patrón Respuesta, donde el estímulo se corresponde con descolgar el teléfono y la respuesta con generar la señal de tono. La formula LTL propuesta para este patrón en [42] es $\Box(action \rightarrow \Diamond response)$. Luego, el usuario debe instanciar este formula general con los eventos concretos del sistema, los cuales en este caso son descolgar el teléfono y generar señal de tono. Hecho este paso el usuario obtendría la siguiente formula LTL: $\Box(descolgar \rightarrow \Diamond generarTono)$ Sin embargo, como se menciona en [53], esta especifi-

cación no refleja con exactitud el comportamiento esperado. Por ejemplo, si el cliente al descolgar el teléfono no oye la señal de tono, podría colgar e intentar de nuevo, y repetir este comportamiento hasta oir la señal de tono. Claramente este no era un comportamiento esperable, y sin embargo es aceptado por la formula LTL propuesta. Para resolver este problema los autores en [53] intentan primero modificar la formula LTL. Sin embargo, esta tarea se vuelve engorrosa y cambian el enfoque. El nuevo camino elegido es modelar la negación de la propiedad y luego probar que no es posible que sea satisfecha. Es importante notar que desde este punto y al tomar este camino, se abandona la utilización de los patrones de especificación, y el problema se resuelve puramente a través de manipulación sobre la fórmula LTL. La nueva fórmula que se obtiene es la siguiente: $\Diamond(descolgar \rightarrow (\neg generarTono \text{ U } colgar))$. Sin embargo, la misma todavía no modela el comportamiento esperado, por lo que es necesario introducir más cambios, tal como se detalla en [53]. Tras varios pasos de prueba y error la formula final obtenida por los autores en [53] es la siguiente: $\Diamond(descolgar \wedge \text{X } ((\neg generarTono \wedge \neg colgar) \text{ U } (\neg generarTono$ $\text{U } colgar))$ (donde X es el operador *Next* y U el operador *Until*).

Ahora se describe a continuación el mismo proceso pero usando FVS como el lenguaje de especificación. La regla ya instanciada con los eventos concretos del sistema para el patrón Respuesta es la que se observa en la figura 5.12-a. Para resolver el problema de que el cliente cuelgue el teléfono hasta oír el tono, se agrega una simple restricción a la regla: el evento *colgar* no debe ocurrir entre los eventos *descolgar* y *generarTono*, como lo muestra la regla de la figura 5.12-b. Yendo un pasó más, el usuario puede generar anti-escenarios para esta regla para llevar a cabo tareas de validación. Los mismos se muestran en la figura 5.12-c. El primero muestra el error cuando el usuario cuelga sin llegar a escuchar la señal de tono. Este comportamiento es el mismo que se solucionó con manipulación lógica en [53]. El segundo anti-escenario refleja una situación donde la señal de tono nunca se genera, pero al mismo tiempo el usuario nunca vuelve a colgar el teléfono. Vale la pena mencionar que el comportamiento de este último anti-escenario no está contemplado en la formula LTL final propuesta en [53]. Esto resalta la utilidad de generar anti-escenarios y razonar mediante el comportamiento complementario.

Comparación entre los métodos y conclusiones La especificación de propiedades basándose en el uso de patrones de especificación, formulas LTL y autómatas sin dudas

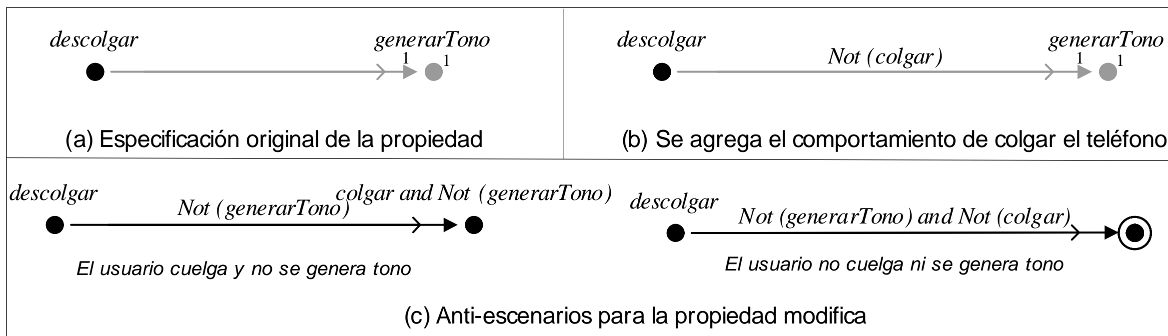

Figura 5.12: Completando el proceso de especificación de propiedades en FVS

exhiben fallas para el ejemplo mostrado. En primera instancia, para detectar el error inicial cuando la señal de tono es escuchada recién después de varios intentos, el usuario debe contar son suficiente experiencia en el manejo de formulas LTL. La especificación de patrones fue útil para ayudar al usuario a obtener la formula LTL necesaria, pero para poder validarla se requiere un conocimiento experto en el lenguaje. Cuando el error es detectado, la fórmula LTL necesita ser modificada. En este punto los patrones de especificación dejan de ser útiles para el usuario, y el problema es resuelto manipulando la formula LTL directamente. Este proceso iterativo de detección y corrección de errores en formulas LTL puede ser difícil y propenso a errores, aún para personas con amplios conocimientos en lógicas temporales. A modo de ejemplo, los autores en [53] se vieron obligados a modelar la negación de la propiedad en vez de continuar trabajando sobre la propiedad en sí misma. Si se hubiera elegido una notación con autómatas se hubieran obtenidos conclusiones similares.

El contexto es diferente al emplear a FVS como lenguaje de especificación. El error detectado es fácilmente corregido agregando una simple restricción. En este punto el usuario puede observar ambas especificaciones (la original y la que contiene el cambio) y analizar la relación de especialización entre ellas, exponiendo los beneficios del atributo *Comparabilidad*. La única diferencia entre ambas es el evento prohibido etiquetando la flecha. Este es un cambio pequeño y localizado, el cual se corresponde con el cambio expresado en lenguaje natural. La exploración de anti-escenarios permitió descubrir un comportamiento anómalo que no está contemplado en la especificación final que se muestra en [53]. Finalmente, un último comentario a destacar es que en FVS el usuario mantiene siempre la especificación *positiva* del requerimiento, en vez de moverse a razonar sobre la negación del mismo.

Capítulo 6

FVS como Lenguaje de Modelado Orientado a Aspectos

En este capítulo se mostrará la versatilidad de FVS como lenguaje declarativo de especificación mostrando cómo puede ser visto como un lenguaje de modelado orientado a aspectos [61]. En este dominio la flexibilidad y riqueza expresiva de FVS permite atacar problemas de modularización en el modelado temprano de propiedades. La estructura de esta sección es la siguiente. Primero se describe y motiva el problema a atacar, y se plantea cómo FVS puede ser visto como un lenguaje de modelado orientado a aspectos. Luego se identifican los principales desafíos que encuentra el modelado de aspectos en etapas tempranas: *falta de flexibilidad en la aplicación de aspectos*, *modelado incremental*, y *la interferencia entre aspectos*. A través de distintos ejemplos se observa cómo FVS intenta solucionar estos desafíos.

6.1. Motivación

En los últimos años la orientación a aspectos ha surgido como un enfoque interesante para manejar la complejidad de la descripción de artefactos de software. El enfoque orientado a aspectos tiene como núcleo la modularización de conceptos entrecruzados [61], es decir, conceptos que de otra manera quedarían desparramados por todo el sistema. Tal objetivo es un principio básico de la Ingeniería de Software. Cada aspecto representa y abstrae a cada uno de estos conceptos, y su comportamiento es agregado a un sistema original, denominado sistema base. Los aspectos tienen una estructura de dos partes: los

poincuts, los cuales seleccionan donde se introducirán el comportamiento de los aspectos, y los *advices*, los cuales especifican en sí el comportamiento que agregará cada aspecto. Tradicionalmente, los aspectos pueden agregarse "antes de", "luego de", o "en vez de" un instante dado del sistema base. Luego, existe un mecanismo de composición denominado *weaving*, que une el comportamiento de los aspectos a un sistema base para formar el sistema final.

La aplicación de aspectos en el modelado temprano de requerimientos parece a primera vista como algo bastante natural y directo, más teniendo en cuenta que muchas veces los requerimientos son expresados de una forma que encajan en la visión de aspectos (por ejemplo, "cada vez que llega un mensaje, el servidor es notificado"). En otras palabras, los aspectos se manifiestan en requerimientos bajo la forma de comportamiento que surge o es "disparado" por la ocurrencia de otro comportamiento [20]. Aplicar la filosofía orientada a aspectos en etapas tempranas es conocido en la comunidad científica como "aspectos tempranos" o "early aspects" en inglés, termino acuñado por [6].

Sin embargo, varios autores han señalado varias dificultades en la aplicación de aspectos en la etapa de modelado, especialmente con notaciones operacionales basadas en autómatas [57, 58]. La mayoría de las aproximaciones orientadas a aspectos terminan aplicado mecanismos sintácticos de composición denominado *weaving*, los cuales adolecen de una semántica clara y precisa [57]. Luego, a diferencia de otros operadores clásicos de modularización como procedimientos, composición paralela, o conjunción lógica en enfoques lógicos, la orientación a aspectos, a pesar de su atractivo, todavía no es considerado como operador como un elemento o cuidadano de primer nivel, sino más bien que es considerado como un mecanismo dinámico de instrumentation cuyo impacto no está caracterizado apropiadamente.

Uno de los principales problemas es la *falta de flexibilidad* para describir los *pointcuts*. Los aspectos fueron concebidos originalmente para las etapas de implementación y codificación, y no para el modelado temprano de requerimientos. Como consecuencia, el modelo para denotar los pointcuts está basado sobre abstracciones de llamadas a métodos, mientras que los *advices* se describen utilizando lenguajes Turing-completos. Este sabor implementativo para especificar aspectos complica su aplicación en etapas tempranas de modelado. En general, el comportamiento de los aspectos debe agregarse justo *antes*, *después*, o *durante* de la ocurrencia de otro comportamiento (en general, abstracciones de

llamadas a métodos, como se mencionó anteriormente). Esto resulta en un enfoque muy poco flexible para la etapa de modelado. Para mencionar dos ejemplos, expresar requerimientos que prediquen sobre comportamiento que ocurrió en el pasado es algo complejo de modelar con aspectos (y a veces hasta imposible). Una propiedad como "Si la alarma suena es porque ocurrió una falla en el pasado" no es fácil de capturar con los modelos tradicionales de aspectos. Los requerimientos que hablan sobre eventos que ocurren dentro de un cierto alcance también sufren del mismo problema, donde se requieren en algunos casos, joinpoints extremadamente complejos para poder describir el comportamiento esperado con exactitud. Esta *falta de flexibilidad* lleva a varios problemas muy conocidos en la comunidad de aspectos como la paradoja de los aspectos ("AOP paradox") [101] o el problema de la fragilidad de los aspectos [65].

Otro problema importante está relacionado con el *modelado incremental*, que constituye una característica deseable para cualquier lenguaje de modelado. El modelado incremental consiste en ir agregando gradualmente nuevas características a un sistema base [109]. Esto es particularmente útil en etapas tempranas del desarrollo de software ya que en las mismas los requerimientos todavía no están especificados por completo. Nuevamente, los aspecto surgen como candidatos aptos para el modelado incremental, ya que un aspecto puede ser visto como un nuevo comportamiento que se agrega a un sistema base cuando se cumplen ciertas condiciones. En este contexto un aspecto puede al mismo tiempo aumentar el comportamiento de un sistema y restringir su aplicación. Por ejemplo, un aspecto encargado de encriptar datos en un sistema agrega la noción de información encriptada. Pero al mismo tiempo establece cuándo y cómo se encriptará y desencriptará la información, restringiendo así su aplicación. Sin embargo, al no tener una semántica clara la utilización de la orientación a aspectos se complica el razonamiento y análisis sobre el sistema aumentado (el sistema que contiene la funcionalidad base más el comportamiento de los aspectos) [57]. Algunos trabajos han sido propuestos para solucionar este problema: introducir una capa intermedia [5, 59, 51], interfaces orientadas a aspectos ([100, 62]) o proveer constructores de aspectos más expresivos [76, 44, 49, 52, 91]. Sin embargo, la mayoría de estas aproximaciones se enfocan sobre el modelo de pointcuts, dejando de lado los advices, y están enfocadas para las etapas de implementación y diseño, dejando de lado la fase de modelado.

Un tercer problema para el modelado temprano de aspectos se conoce como *interfe-*

rencia entre aspectos [22]. La interferencia entre aspectos tiene lugar cuando el comportamiento de dos o más aspectos interactúa entre sí, entrando potencialmente en conflicto. Ya que el comportamiento de un aspecto podría llegar a cambiar el estado del sistema, el orden que los aspectos son aplicados podría ser decisivo para el correcto funcionamiento del sistema. Los siguientes puntos resumen cuestiones que deben ser atacadas para manejar el problema de la interferencia entre aspectos

- *Orden de ejecución*: ¿Qué aspecto debe ejecutarse primero? ¿Porqué? ¿Es relevante el orden? Responder estas preguntas se conoce como definir la precedencia entre aspectos. En algunas situaciones el comportamiento final del sistema aumentado (el sistema que ya tiene agregado el comportamiento de los aspectos) depende del orden de ejecución de los aspectos. Por ejemplo, suponer dos aspectos influyendo en un protocolo de comunicación entre procesos. El primero encripta la información agregando seguridad al protocolo, mientras el segundo prepara los datos en estructuras adecuadas para el canal de comunicación, fragmentando la información a enviar. Ambos actúan en el mismo punto, al enviar el sistema datos, por lo que ambos interfieren entre sí. En este contexto podría ser el caso que la información deba ser encriptada primero, y fragmentada después. Luego, si el aspecto de fragmentación es ejecutado primero, el aspecto de encriptación podría no ejecutarse como lo esperado, exponiendo a todo el sistema de comunicación a problemas de seguridad. Por lo tanto es esencial para la persona realizando la especificación del sistema detectar y entender completamente toda situación de interacción entre aspectos y determinar para cada una el orden de precedencia entre los aspectos involucrados.

- *Dependencia de comportamiento*: ¿Existe conflicto entre el comportamiento de los aspectos que interfieren o son completamente independientes? En algunos casos, un aspecto puede depender del comportamiento de otro. Esto se debe a que los aspectos comparten todo el estado del sistema, incluyendo variables, datos o invocación de métodos. Si los aspectos involucrados actúan de una manera "solo-lectura" jugando el rol de observadores, no habrá dependencia entre sus comportamientos. Por ejemplo, dos aspectos encargados de contar cuántas veces se invocan determinados métodos no tendrán dependencia entre sí, ya que no alteran el estado del programa. Este tipo de aspectos se denominan "espectadores" en la categorización de aspectos

propuesta por [57]. Sin embargo si los aspectos modifican datos, variables o alteran el flujo de ejecución, entonces existe dependencia de comportamiento que debe ser tenida en cuenta al momento de especificar los aspectos. En este caso los aspectos entran en la categoría "reguladores" o en la categoría "invasivos". Los aspectos reguladores pueden modificar el estado del sistema pero no el flujo de ejecución. En cambio, los aspectos invasivos pueden cambiar por completo la semántica del sistema, sin ninguna restricción. Esto implica que su aplicación puede invalidar cualquier propiedad que valía antes de la ejecución del aspecto. Dada esta característica los aspectos invasivos son considerados peligrosos [57], y en este trabajo se enfocará solamente en los aspectos "espectadores" y "reguladores.

- *Razonamiento en el sistema aumentado*: Una vez que el comportamiento de todos los aspectos fue agregado al sistema, ¿qué se puede afirmar del comportamiento del sistema final? ¿Fueron todos los aspectos agregados correctamente? Como se mencionó anteriormente, la falta de una semántica clara de la aplicación de aspectos puede llegar a complicar este tipo de razonamiento sobre el sistema aumentado [57]. Si los aspectos que interfieren son muchos el comportamiento del sistema aumentado puede llegar a ser complejo de estimar.

Para poder responder a estos puntos es crucial entender y razonar sobre la aplicación de los aspectos cuando los mismos interactúan entre sí. Luego, el formalismo usado para su especificación juega un rol importante. En las etapas de modelado este problema recrudece aún más, ya que no existe todavía un implementación concreta para inspeccionar. En este punto las especificaciones declarativas del comportamiento tienen bastante que aportar, ya que su naturaleza las hace aptas para capturar requerimientos de manera temprana [107]. La especificación declarativa del comportamiento se concentra en definir **qué** comportamiento debe agregarse, y no **cómo** hacerlo, lo cual si sucede con notaciones operacionales como máquinas de estado, notaciones de grafos, o state charts. Las técnicas de minería de aspectos [99, 41, 89] ayudan a identificar y detectar la interferencia entre aspectos. Sin embargo, una vez determinado el punto de conflicto, el usuario debe aún enfrentar el problema de determinar cómo interactuarán los mismos y establecer el orden correcto de ejecución. Para llevar a cabo esta tarea debe ser sencillo determinar la precedencia de los aspectos con solo mirar su especificación. Sin embargo, la falta de una

semántica clara para componer los aspectos puede dificultar seriamente este análisis [57]. Otra fuente útil de análisis es razonar por el *comportamiento complementario*. Entender y observar el comportamiento que se está excluyendo es importante al momento de explorar y definir una propiedad. Finalmente, el análisis de las trazas válidas del sistema ayuda al usuario a detectar y priorizar la interacción de aspectos.

Dado este contexto, existe la necesidad de contar con un lenguaje de modelado declarativo orientado a aspectos, con la suficiente flexibilidad para poder solucionar los problemas ya mencionados. FVS cuenta con estas características y puede adaptarse fácilmente como un lenguaje de modelado orientado a aspectos. Por un lado, su naturaleza declarativa lo hace especialmente apto para el modelado incremental, y por otro, la flexibilidad para expresar comportamiento es ideal como modelo de pointcuts. A continuación se describe la manera en que FVS puede verse como un lenguaje de modelado orientado a aspectos, y cómo intenta superar los desafíos planteados.

6.2. FVS como un lenguaje de modelado orientado a aspectos

Como se mencionó anteriormente, los aspectos se describen a través de un pointcut, que establece dónde se va a aplicar el comportamiento del aspecto, y un advice, que especifica el comportamiento del aspecto. Es usual que los requerimientos sigan esta forma de descripción, expresando qué cosas deben suceder en el sistema cuando una cierta situación tiene lugar. Un ejemplo de un requerimiento que cumple esta modalidad es para un sistema que maneja las luces interiores de un automóvil es: “Cuando una puerta se abre, las luces interiores del auto deben prenderse”. Bajo una notación de aspectos esta propiedad puede modelarse de la siguiente manera:

- **pointcut** (dónde y cuándo): *Se abre alguna puerta del auto.*
- **advice** (qué): *Prender luces interiores.*

Las reglas FVS encajan de manera directa a una perspectiva orientada a aspecto: los antecedentes toman el lugar de los pointcuts, mientras que los consecuentes toman el lugar de los advices. La tabla 6.1 refleja este mapeo.

Tabla 6.1: Mapeo entre conceptos de la orientación a aspectos a conceptos FVS

Orientación a Aspectos	FVS
Pointcut	Antecedente de una regla
Advice	Consecuente(s) de una regla
Aspecto	Regla FVS

A modo de ejemplo de este mapeo, la regla en la figura 6.1 muestra la propiedad ya mencionada especificada en FVS, donde el antecedente de la regla se comporta como un pointcut y el consecuente como un advice.

Figura 6.1: Una regla FVS comportándose como un aspecto

En las siguiente secciones se analizará el comportamiento de FVS como lenguaje de modelado orientado a aspectos considerando los desafíos ya mencionados: modelado incremental, flexibilidad del modelo de pointcuts e interferencia de aspectos.

6.3. Modelado Incremental y Flexibilidad del modelo de pointcuts

Para mostrar cómo FVS soporta el modelado incremental y la flexibilidad del modelo de pointcuts se trabajará con un caso de estudio simple presentado en [79]. El mismo consiste en un sistema de software embebido para un automóvil que controla el funcionamiento de sus luces interiores. En pocas palabras, el sistema está encargado de prender las luces interiores cuando se abre alguna puerta, y de apagarlas cuando se cierran, basándose en el estado de las puertas, las trabas de las puertas, y el mecanismo de encendido del auto. La regla anterior, presentada en la figura 6.1 forma parte de los requerimientos del sistema.

Siguiendo esta modalidad FVS-orientada a aspectos, otras funcionalidades también pueden agregarse a esta especificación inicial. Un nuevo aspecto podría utilizarse para

garantizar que las puertas sólo se abran si no tienen puesta la traba de seguridad. Esto implica controlar que siempre que se abre una puerta, la misma haya sido destrabada en el pasado, y se haya mantenido en ese estado hasta el momento de abrirse. La regla en la figura 6.2 ilustra este comportamiento. Notar que en este caso el advice está describiendo comportamiento que ocurrió en el pasado del pointcut.

Figura 6.2: Regla FVS controlando el estado de la puerta

La regla en la figura 6.3 especifica el cuándo y cómo se deben apagar las luces interiores. La regla establece que las luces deben apagarse ya sea porque se cerró una puerta (consecuente 1), o bien porque la batería del auto se quedó sin energía (consecuente 2). Nuevamente, el advice de este aspecto predica sobre comportamiento que ocurre previo al pointcut.

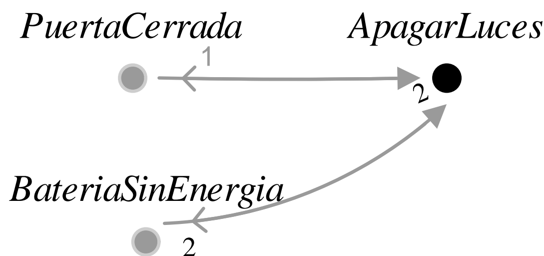

Figura 6.3: Aspectos controlando cómo y cuándo se deben apagar las luces

Finalmente, se modelan dos requerimientos más. La regla en la figura 6.1 especifica cuándo deben encenderse las luces del sistema, el cual constituye un requerimiento clave del sistema. Esta funcionalidad puede extenderse para decir que las luces deben apagarse cuando una puerta se cierra, tal como lo refleja la regla en la figura 6.4-a. Por último, la regla en la figura 6.4-b completa la especificación del comportamiento esperado. La misma establece que las luces interiores no pueden encenderse dos veces de manera consecutiva sin que se hayan apagado previamente. Como se puede apreciar, requiere que entre dos eventos consecutivos de *PrenderLuces* ocurra un evento de *ApagarLuces*. Este comportamiento impide que el evento *PrenderLuces* ocurra cuando las luces ya estén prendidas. Este caso también ilustra la flexibilidad de FVS para modelar pointcuts, ya que el comportamiento introducido por el aspecto ocurre entre los dos eventos que constituyen el pointcut.

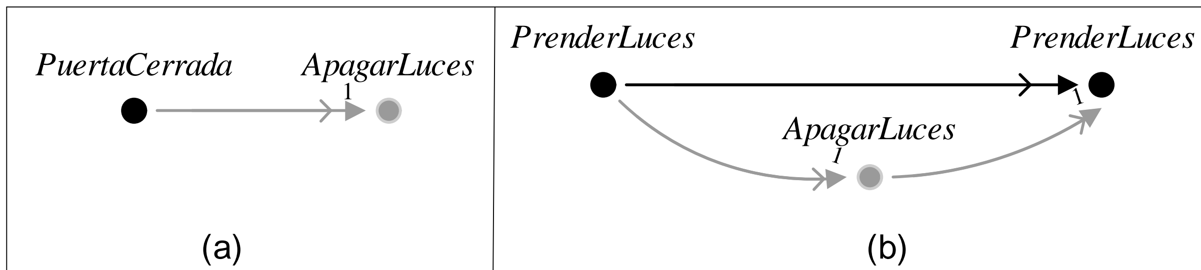

Figura 6.4: Aspectos en FVS para el sistema de luces

6.3.1. Análisis y Resultados

El ejemplo presentado permite establecer un análisis del desempeño de FVS como lenguaje orientado a aspectos, en especial considerando los desafíos del modelado incremental y la flexibilidad en el modelo de pointcuts. Respecto al modelado incremental, FVS lo implementa de una manera natural y directa. Los nuevos requerimientos son agregados simplemente incluyendo la regla que describe su comportamiento al conjunto de reglas que describe el comportamiento esperado. En el ejemplo visto se fueron agregando nuevos requerimientos de manera incremental, incorporando cada vez una nueva regla. Respecto a la flexibilidad del modelo de pointcuts, FVS logró capturar con precisión y naturalidad todos los requerimientos necesarios. Los pointcuts que especificaban comportamiento sobre eventos en el pasado fueron modelados sin problemas. En la regla de la figura 6.2 el comportamiento del consecuente (el evento *PuertaDestrabada*) ocurre antes que el comportamiento del antecedente el evento (*PuertaAbierta*). De manera similar, el comportamiento de ambos consecuentes en la regla 6.3 ocurre previo al comportamiento del antecedente. Tampoco hubo problemas al modelar pointcuts que hablaban sobre eventos ocurriendo en el medio de otros. Por ejemplo, la regla en la figura 6.4-b modela un aspecto donde el comportamiento del advice ocurre entre los dos puntos que constituyen el pointcut del aspecto (el evento inicial *LucesPrendidas* y el evento final *LucesPrendidas*). Definir este tipo de pointcuts es difícil (y hasta a veces imposible) en modelos de pointcuts que razonan sobre abstracciones de llamadas a métodos. En estos modelos tradicionales se recurre a métodos como re-escribir la propiedad en otros términos, o bien, se obtiene pointcuts complejos, donde es difícil deducir cuál es el comportamiento que denotan. En ambos casos, las soluciones no son del todo satisfactorias, al obtenerse modelos difíciles de evolucionar y entender.

6.4. Interferencia de Aspectos

Las características de FVS lo hacen un lenguaje de modelado orientado a aspectos apto para manejar los problemas que nacen a partir de que dos o más aspectos interfieren entre sí. Por un lado, su naturaleza declarativa ayuda a comprender de manera natural la interacción de aspectos. Segundo, la capacidad de generar anti-escenarios de manera automática permite razonar por el comportamiento complementario y así tener un panorama más completo para tomar una decisión cuando dos aspectos interfieren entre sí. Finalmente, la capacidad de poder observar las trazas válidas del sistema que corresponden al comportamiento de los aspectos (gracias a un mecanismo de tableau) le permite al usuario detectar y analizar posibles conflictos. Es decir FVS cuenta con la posibilidad de analizar comportamiento localizado. A continuación se explora con mayor profundidad cada una de estas características, mostrando en cada caso un ejemplo distinto. En la sección 6.4.1 se introduce el ejemplo Telecom, presente en la distribución de AspectJ [60], una de las herramientas más populares para programar con aspectos. En la sección 6.4.2, enfocada en el uso de anti-escenarios para la interferencia de aspectos, se utiliza el ejemplo JukeBox presentado en [41], mientras que en la sección 6.4.3 se analiza el conflicto de logging vs. encripción detallado en [40]. Finalmente, la sección concluye analizando los resultados obtenidos.

6.4.1. Especificaciones Declarativas y la interferencia entre aspectos

El ejemplo Telecom, que forma parte de la distribución de AspectJ, es una aplicación muy simple que simula un sistema telefónico donde los clientes pueden realizar llamadas locales, regionales y de larga distancia. Para mostrar parte de su comportamiento, se introducen algunas reglas FVS. La regla en la figura 6.5-a establece que toda conexión establecida debe eventualmente finalizar. Más concretamente, pide que todo evento *ConexionCompleta* sea seguido por un evento *ConexionFinalizada*. La regla en la figura 6.5-b considera dos comportamientos posibles luego de que el cliente levanta el teléfono. O bien la conexión se establece (consecuente 1) o ocurre un error de comunicación en caso contrario (consecuente 2). Finalmente, 6.5-c modela las condiciones que deben cumplirse para la realización de una llamada. Si el evento *ConexionCompleta* es seguido por el

evento *ConexionFinalizada*, entonces es el caso que el cliente tiene que haber levantado el teléfono antes de iniciarse la conexión, y demás, tiene que haber cortado entre que la conexión se estableció y finalizó.

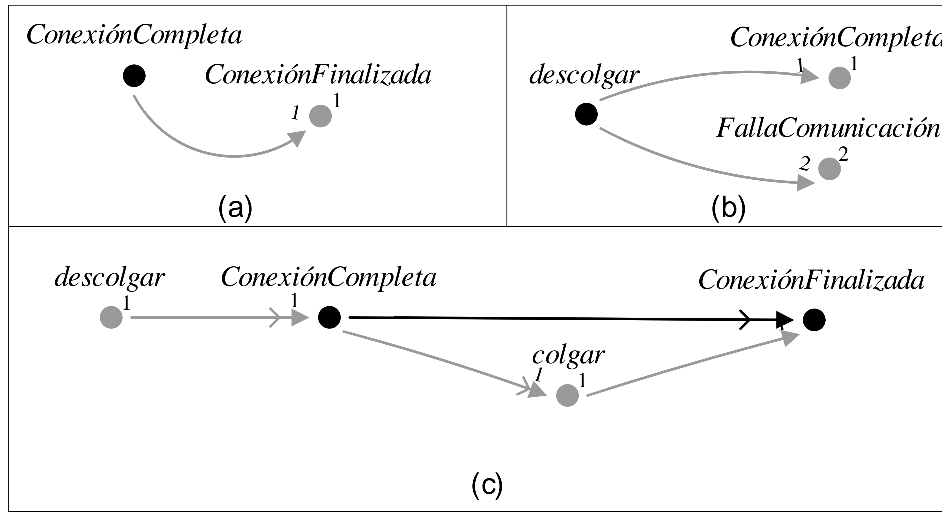

Figura 6.5: Reglas FVS modelando requerimientos del ejemplo Telecom

Ahora se procederá a agregar el comportamiento de un aspecto a esta especificación inicial. Se trata del aspecto "Tiempo", el cual se encarga de contabilizar la duración de una llamada. El comportamiento de este aspecto es simple: empieza a contar el tiempo cuando una conexión se establece y empieza la llamada, y se detendrá al finalizar la conexión. Las reglas de la figura 6.6-a y 6.6-b modelan apropiadamente el comienzo y finalización del *timer* utilizado en el aspecto para llevar constancia del paso del tiempo. Finalmente, la regla en la figura 6.6-c establece que el tiempo de la llamada (representado por el evento *TiempoTranscurrido*) será calculado una vez detenido el timer.

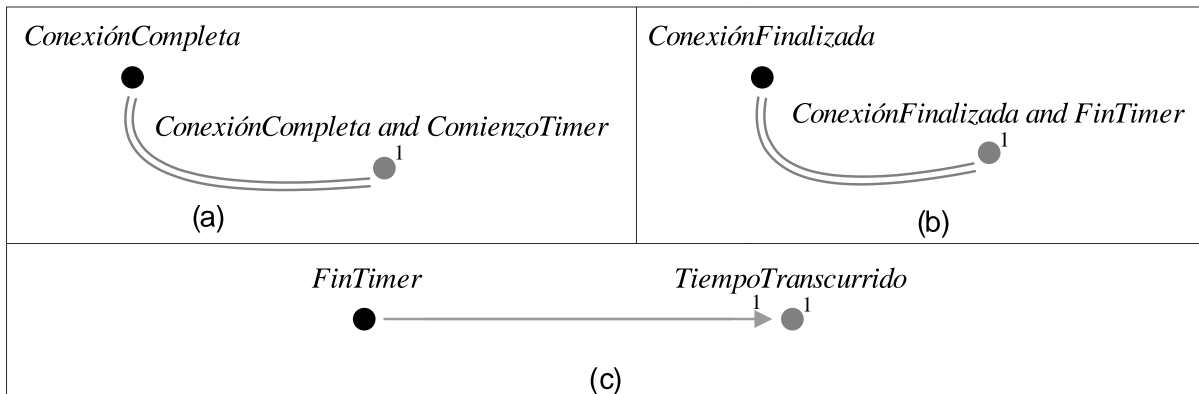

Figura 6.6: El aspecto tiempo en FVS

Ahora se agregará el comportamiento de un nuevo aspecto, denominado “Facturador”, encargado de computar el costo de la llamada según la duración de la misma. Siguiendo estos requerimientos se presenta en la figura 6.7 una regla que describe parte del comportamiento de este aspecto. En particular, la regla establece que cada ocurrencia del evento *CostoLlamada* debe ser precedida por el evento *TiempoTranscurrido*. Es decir, el costo de las llamadas no puede calcularse sin saber la duración de las mismas.

Figura 6.7: El aspecto Facturador en FVS

Como puede observarse, el aspecto “Facturador” y el aspecto “Tiempo” están en conflicto. El correcto funcionamiento del sistema depende que el aspecto “Tiempo” se ejecute primero que el aspecto “Facturador”, dado que en caso contrario éste último no tiene cómo computar el costo de la llamada. La especificación declarativa en FVS permitió modelar esta situación con naturalidad en la figura 6.7. Existe una restricción de manera explícita entre los eventos *TiempoTranscurrido* y *CostoLlamada*, al estar ambos eventos conectados por una flecha indicando precedencia. La misma situación en AspectJ, o en otras notaciones operacionales la solución radica en introducir una sentencia **declare precedence**, y luego a continuación una lista de aspectos, que el programador introduce manualmente, lo cual es claramente un proceso que lleva a errores. De acuerdo a la documentación de AspectJ, la precedencia de los aspectos está determinada por el orden de los aspectos en la lista. Así, el primer aspecto en la lista es el que tiene mayor precedencia y se ejecutará primero si hay conflicto entre dos aspectos. Luego tendrá precedencia el segundo en la lista, y así sucesivamente. Sin embargo, dada la semántica de AspectJ el comportamiento de esta lista de precedencia no es tan intuitivo como parece. Por ejemplo, para el caso de la aplicación Telecom y el conflicto entre el aspecto “Tiempo” y “Facturador” se resuelve con el siguiente comando ***declares precedence: Facturador, Tiempo***, y no con la sentencia ***declares precedence: Tiempo, Facturador***, que es a priori la más intuitiva, ya que se quiere que el aspecto “Tiempo” se ejecute primero. Sin embargo, esta última opción no es correcta, ya que la precedencia se ve afectada por el flujo de ejecución, y el orden de aplicación de los aspectos. Los aspectos cuyo comportamiento ocurre “antes

de" un instante en particular, siguen el razonamiento esperado: se ejecutan los aspectos según el orden de la lista. Sin embargo, para los aspectos que se agregan "luego de" de instante dado, el orden de ejecución es en sentido inverso de la lista, dado que el flujo de control está de alguna manera "retornando" . Luego, los aspectos "luego de" se ejecutan en sentido inverso de la lista. Dado esto, y cómo los aspectos "Tiempo" y "Facturador" se aplican en el ejemplo luego de que la llamada termine, el orden correcto es ***declares precedence: Facturador, Tiempo***. Como se observa, aún para este simple caso es confusa la utilización de la sentencia **declare precedence**, lo que lleva a cometer errores en la especificación.

Por último, es importante hacer la siguiente reflexión sobre la flexibilidad del modelo de pointcuts. En el solución provista en AspectJ para el ejemplo Telecom, el costo de la llamada debe calcularse exactamente después que que la conexión finaliza. Sin embargo, esto podría no corresponderse con la especificación actual del protocolo. Por ejemplo, podría ser el caso que las llamadas recién se facturen y obtengan su costo todas juntas al finalizar un determinado mes. Debido a la falta de flexibilidad en el modelo, las llamadas se calculan sí o sí en ese punto, llevando a tomar decisiones prematuras. En FVS es fácil dejar este comportamiento abierto, y contemplar otras opciones.

6.4.2. Anti-escenarios y la interferencia entre aspectos

Esta sección se basará en el ejemplo JukeBox presentado en [41]. Básicamente este sistema permite a un usuario seleccionar una canción, que luego será reproducida por un reproductor multimedia. El sistema incluye el comportamiento de dos aspectos: el aspecto "Recientes", el cual se encarga de almacenar en una lista los últimos temas reproducidos, y el aspecto "Contador", el cual lleva estadísticas del número de veces que se ha reproducido un dado tema. Como ambos aspectos se aplican al momento de reproducir una canción, están en conflicto. Una posible manera de resolverlo se ilustra en la figura 6.8, donde el aspecto "Recientes" se ejecuta primero.

Para explorar y validar la opción elegida, el usuario puede generar anti-escenarios e inspeccionarlos. FVS genera para la regla 6.8 cuatro anti-escenarios, que se ven en la figura 6.9. En la figura 6.9-a se modela la situación donde una canción es seleccionada, pero la misma no es ni almacenada en la lista de recientes, ni se actualiza su contador. Claramente es una situación de error. Las reglas en la figuras 6.9-b y 6.9-c contemplan

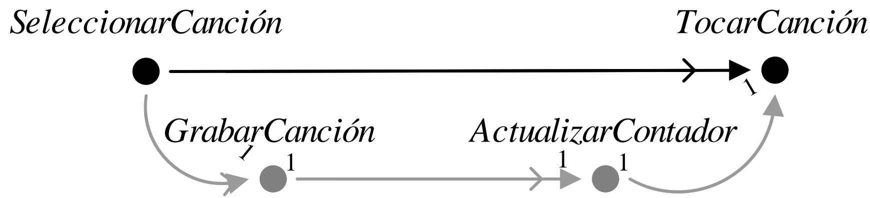

Figura 6.8: Aspectos en conflicto en el sistema JukeBox

situaciones de error donde ocurre sólo uno de los aspectos: en la primera no se actualiza el contador, y en la segunda la canción no es almacenada. Finalmente, la regla en la figura 6.9-d ilustra el comportamiento donde primero se actualiza el contador, y luego la canción es almacenada en la lista de recientes. Los anti-escenarios representan una información valiosa al momento de resolver un conflicto, ya que permiten visualizar el comportamiento que lleva a violar una propiedad.

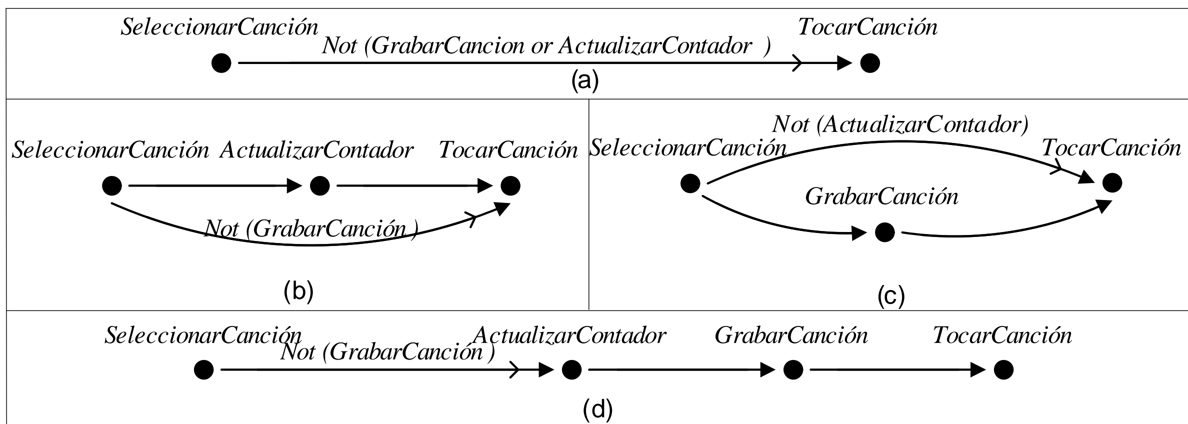

Figura 6.9: Anti-escenarios para el sistema JukeBox

Con solo inspeccionar los anti-escenarios el usuario puede validar su elección. Por ejemplo, el usuario podría inspeccionar el último caso (regla 6.9-d) y darse cuenta que en realidad el comportamiento esperado es que ambos eventos ocurran (que se actualice el contador y que se almacene la canción), pero sin importar el orden de ocurrencia. Luego es perfectamente legal para el sistema que se actualice primero el contador y luego se proceda con su almacenamiento. Tras este análisis, el usuario corrige la regla original de la figura 6.8, y arriba al comportamiento mostrado en la figura 6.10.

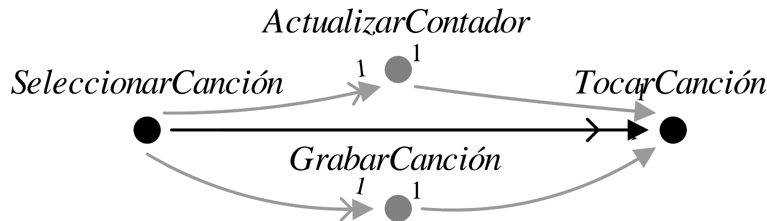

Figura 6.10: Conflicto de aspectos revisitado tras análisis de anti-escenarios

6.4.3. Modelado Localizado y la interferencia entre aspectos

FVS permite analizar distintas características del sistema siendo modelado de manera independiente, para detectar y resolver de manera temprana la interacción entre aspectos. Para ilustrar este punto, se considera el conflicto entre los aspectos de logging y de encriptación detallados en [40]. En pocas palabras se requiere que toda información a ser enviada sea logueada y encriptada antes de ser enviada. Este comportamiento se modela en FVS con las reglas en la figura 6.11-a. Cada regla establece que cada dato es logueado y encriptado antes de ser enviado.

A través del proceso de tableau detallado en [29] se pueden generar traza válidas para un conjunto dado de reglas. Gracias a esto, el usuario puede inspeccionar el resultado para detectar posibles conflictos y obtener más información acerca de la especificación del comportamiento esperado. La siguiente sucesión de eventos representa una traza válida {*ServidorListo, Encriptar, Loguear, EnviarInfo*} para la regla de la figura 6.11-a. En este caso la información es encriptada primero, y luego logueada. Analizando este comportamiento el usuario podría darse cuenta que esta traza es en realidad inválida, ya que el dato logueado ya está encriptado, resultando inútil para tareas de seguridad y auditoría. La posibilidad que brinda FVS para analizar la ejecución parcial de un sistema lleva a identificar y resolver interferencia entre aspectos. Para resolver el problema el usuario agrega una nueva regla para establecer el orden entre el aspecto de logging y el de encriptación, como se ve en la figura 6.11-b.

6.4.4. Análisis y Resultados

En el primer ejemplo visto el conflicto entre el aspecto Tiempo y el aspecto Facturador se resolvió implicitamente por la especificación declarativa de FVS. El conflicto entre ambos aspectos está definido explícitamente por una flecha de precedencia, la cual

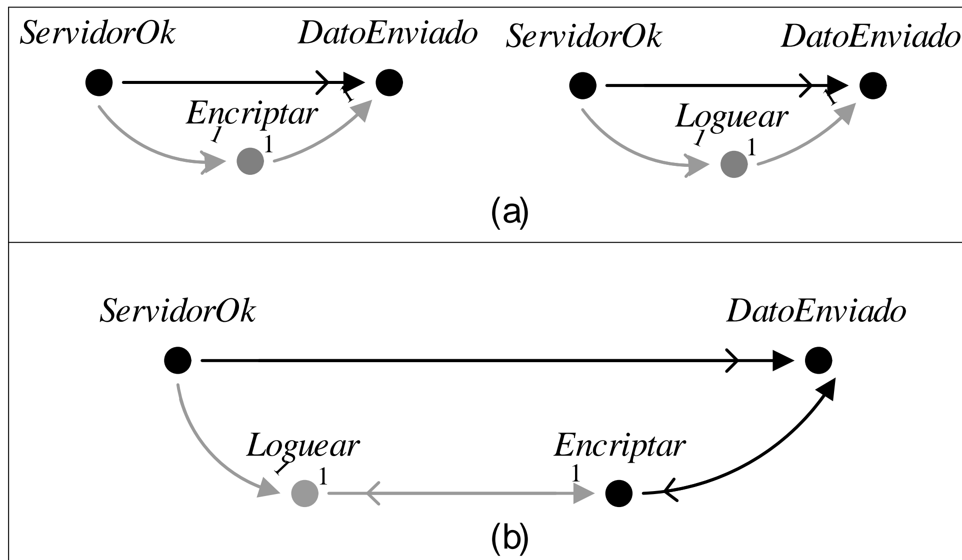

Figura 6.11: Conflicto logging-encriptación

representan una información clara y simple de entender. Esta capacidad no es tan sencilla de lograr en notaciones operacionales. En estos casos la precedencia se establece usando operadores basados en UML, por lo que el usuario requiere de entender una composición de artefactos para resolver el conflicto, donde muchas veces la composición no tiene una semántica clara. En el segundo ejemplo, el usuario emplea razonamiento complementario para entender y resolver la interferencia entre dos aspectos que actuaban sobre un sistema que reproduce temas musicales. Inspeccionando anti-escenarios el usuario obtiene información valiosa y puede corregir la especificación con la información obtenida (la especificación original era demasiado restrictiva, ambos aspectos podían suceder en cualquier orden). De manera similar en el tercer ejemplo el usuario al analizar trazas válidas para la aplicación de los aspectos logra detectar una interacción entre aspectos y un error en la especificación original. Esta vez, la especificación original era muy permisiva, por lo que se agrega una nueva regla para establecer un orden concreto de ejecución entre aspectos. Esta capacidad es particularmente útiles en etapas tempranas del desarrollo, donde todavía no se cuenta con una implementación concreta.

Capítulo 7

Aumentado el poder expresivo: lenguajes ω regulares

En el presente capítulo se describen las modificaciones a FVS que permiten dar un salto en el poder expresivo, dando lugar a ω-FVS, un lenguaje de modelado capaz de describir propiedades ω-regulares. Se presenta un análisis del poder expresivo de ω-FVS, y se muestran como casos de estudio el modelado de dos protocolos de comunicación.

7.1. ω-FVS

En esta sección se introduce ω-FVS, detallando cómo el poder expresivo de FVS puede aumentarse de manera sencilla para poder denotar propiedades ω-regulares, logrando así cubrir de una manera más completa la especificación de comportamiento. El salto en el nivel expresivo se da gracias a introducir nociones de abstracción en el razonamiento de propiedades. La abstracción en ω-FVS estará dada por la presencia de un nuevo tipo de eventos, a través de los cuales el usuario puede abstraer comportamiento y predicar sobre eventos que no están presentes en la ejecución del sistema, sino que representan un nivel de abstracción mayor. Este nuevo tipo de eventos recibe el nombre de eventos "fantasma", en contraste a los eventos "actuales", que son los que están presentes en el vocabulario del sistema. De esta forma los escenarios ω-FVS estarán compuestos de eventos "fantasma" y "actuales". Más formalmente, un escenario ω-FVS estará compuesto por una tupla $\langle \Sigma, P, \ell, \equiv, \not\equiv, <, \gamma \rangle$, de manera casi idéntica a un escenario FVS, pero con la salvedad que ahora $\Sigma = \Sigma_a \uplus \Sigma_g$, donde Σ_g denota los eventos fantasma y Σ_a los eventos actuales.

Para facilitar la manipulación formal de ambos tipos de eventos, se introduce un operador simple $\downarrow_{\Sigma_a}$ para especificar que los eventos actuales son aquellos definidos en Σ_a, donde $\Sigma_a \subseteq \Sigma$ y Σ_g=$\Sigma - \Sigma_a$ se intepretan cómo los eventos fantasma. De esta manera una regla ω-FVS r denotada como $r{\downarrow_{\Sigma_a}}$ indica que los eventos actuales en la regla son aquellos en Σ_a. Luego, los eventos fantasma de la regla estarán definidos implícitamente por $\Sigma - \Sigma_a$.

La semántica de ω-FVS se define de manera muy similar a la vista para FVS, pero teniendo en cuenta la presencia de los eventos fantasma. En primer término se define una operación que determina cuándo un escenario ω-FVS incluyendo eventos fantasmas puede ser proyectado en una traza del sistema (es decir, en una traza que contiene sólo eventos actuales). La idea central de esta operación es descartar los eventos fantasmas a través del clásico proceso de eliminación existencial. Con la presencia de esta operación es sencillo definir cuándo una dada traza satisface una regla o un conjunto de reglas, las cuales contienen eventos fantasmas y actuales, y así establecer la semántica del lenguaje ω-FVS.

Morfismo de Proyección

El *Morfismo Proyección* es una operación clave encargada de manejar el salto de abstracción que introduce la presencia de los evento fantasmas. En pocas palabras, esta operación define cuándo un escenario ω-FVS, conteniendo eventos fantasmas, puede ser proyectado en una traza del sistema, la cual incluye sólo eventos actuales. Notar en este punto que esta proyección es análoga a la proyección estándar de trazas, donde se eliminan variables que no están incluidas en un dado alfabeto.

Definition 7.1.1 (Morfismo de Proyección)**.** *Dado un escenario traza $\mathcal{S}_\sigma$ $\langle \Sigma_\sigma, P_\sigma, \ell_\sigma, \equiv_\sigma, \not\equiv_\sigma, <_\sigma, \gamma_\sigma \rangle$, un escenario ω-FVS $\mathcal{S}$ $\langle \Sigma, P, \ell, \equiv, \not\equiv, <, \gamma \rangle$ y eventos $\Sigma_a \subseteq \Sigma$ (asumiendo un universo común de proposiciones de eventos y etiquetas), y g una función total entre P y P_σ se dice que g es una morfismo proyección de $\mathcal{S}$ a $\mathcal{S}_\sigma$ para Σ_a (denotado $g : \mathcal{S} \to \mathcal{S}_\sigma$) si y solo si:*

M1: $\Sigma\sigma = \Sigma_a$ *y los eventos fantasmas* $\Sigma_g = \Sigma - \Sigma\sigma$

M2: $\ell_\sigma(g(\mathsf{p})) \Rightarrow \exists\ \upsilon_1, \upsilon_2 \ldots \upsilon_n\ \ell(\mathsf{p})$ *es una tautología para todo* $\mathsf{p} \in P$ *donde* $\Sigma_g = \{\upsilon_1, \upsilon_2 \ldots \upsilon_n\}$

M3: $\gamma_\sigma(g(\mathsf{p}), g(\mathsf{q})) \Rightarrow \exists\ \upsilon_1, \upsilon_2 \ldots \upsilon_n\ \gamma(\mathsf{p}, \mathsf{q})$ *es una tautología para todo* $\mathsf{p}, \mathsf{q} \in P$ *donde* $\Sigma_g = \{\upsilon_1, \upsilon_2 \ldots \upsilon_n\}$

M4: *si*$\mathsf{p} \equiv \mathsf{q}$ *entonces* $g(\mathsf{p}) \equiv_\sigma g(\mathsf{q})$ *para todo* $\mathsf{p}, \mathsf{q} \in P$

M5: *si* $\mathsf{p} \not\equiv \mathsf{q}$ *entonces* $g(\mathsf{p}) \not\equiv_\sigma g(\mathsf{q})$ *para todo* $\mathsf{p}, \mathsf{q} \in P$

M6: *si* $\mathsf{p} < \mathsf{q}$ *entonces* $g(\mathsf{p}) <_\sigma g(\mathsf{q})$ *para todo* $\mathsf{p}, \mathsf{q} \in P$.

Esta definición es muy similar a la operación de morfismos entre escenarios definida previamente (ver definición 3.3.2), con el agregado de condiciones extras encargadas de llevar a cabo la eliminación existencial de los eventos fantasmas (**M2:** y **M3:**). Finalmente, la próxima definición establece la semántica de ω-FVS. **La semántica de un conjunto de reglas R es el conjunto de todas las trazas que satisfacen *R***. Formalmente se obtiene la siguiente definición:

Definition 7.1.2 (Semántica de trazas para una regla ω-FVS)**.** *Un escenario de trazas* $\mathcal{S}_\sigma$*, satisface un conjunto de reglas* $\mathrm{R}{\downarrow_{\Sigma_a}}$ *si y solo si: existe un* $\mathcal{S}$ *tal que:* $\forall\ r \in \mathrm{R}\ \mathcal{S} \models r$ *and* $\exists\ g$*, un morfismo proyección* $g : \mathcal{S} \to \mathcal{S}_\sigma$ *para* Σ_a.

Dicho en otras palabras, una traza satisface un conjunto de reglas si existe un escenario que puede ser proyectado en la traza, y que al mismo tiempo satisface todas las reglas en el conjunto.

Un breve ejemplo: Sistema de luces interiores

Para ilustrar la utilización de eventos fantasma se retoma el ejemplo introducido previamente, basado en un sistema para el manejo de las luces interiores de un automóvil presentado en [79]. Los eventos disponibles en el vocabulario del sistema incluyen eventos como *Encendido*, *Apagado*, *PrenderLuces*, *ApagarLuces*, *AbrirPuerta*, *CerrarPuerta*, *PuertaTrabada* y *PuertaDestrabada*.

En este punto se desea agregar un nuevo requerimiento, el cual apunta a mejorar el rendimiento de la batería del auto. En particular, se desea evitar un gasto excesivo de la batería que se da cuando se prenden las luces del auto sin que esté encendido. En estos casos se dice que la batería trabaja en un modo “caro”. Dado este contexto se desea restringir esta modalidad, y una forma de lograrlo es exigir que entre dos períodos consecutivos de un uso caro de la batería el auto se encienda para poder recargar la batería. Sin embargo, este modo caro no forma parte del vocabulario del sistema. Es decir, no es un evento incluido en la especificación. Esto lleva a que el usuario escriba esta modalidad

en términos de los eventos disponibles, el cual es un proceso propenso a errores y difícil de validar.

Este tipo de propiedades pueden formularse en ω-FVS empleando eventos fantasmas. Esto se logra introduciendo el evento fantasma “ModoCaro”, capturando el momento donde la batería funciona con exceso de capacidad, que se da cuando las luces son encendidas y el auto está apagado. Este comportamiento está reflejado en la regla de la figura 7.1. De manera similar, cuando las luces se encienden con el auto encendido se emplea la fórmula “Not (ModoCaro)”. Una vez definidos estos eventos, se pueden expresar propiedades en base a ellos, como la muestra la última regla de la figura 7.1. Esta última regla refleja el comportamiento pedido para restringir el uso excesivo de las luces que provocaba que se descargue la batería.

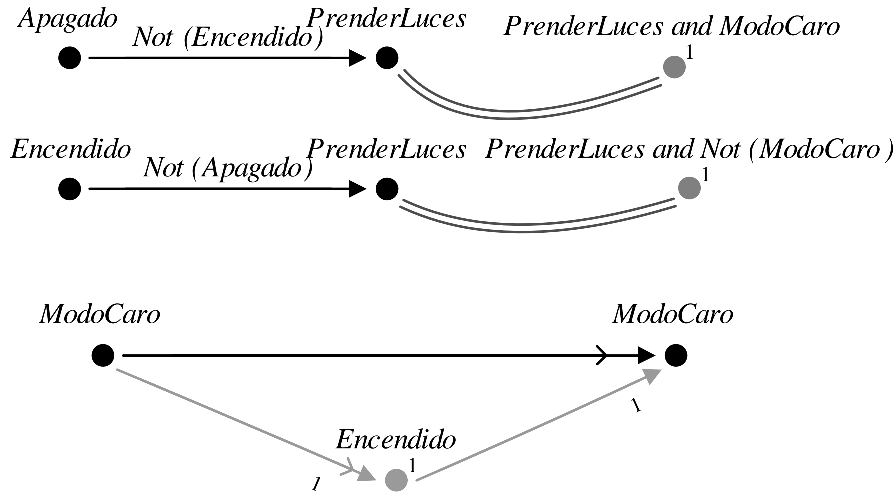

Figura 7.1: Reglas que denotan el modo caro del sistema de luces

Para ilustrar cómo funciona la operación de *morfismo proyección*, considerar la figura 7.2. En la misma se encuentra una traza del sistema en la parte superior ($\mathcal{S}_\sigma$), y un escenario $\mathcal{S}$ incluyendo el evento fantasma “ModoCaro”en la parte inferior. Es decir, $\Sigma = \Sigma_a \uplus \Sigma_g$, donde $\Sigma_a = \{Apagado, PrenderLuces\}$ and $\Sigma_g = \{ModoCaro\}$. El morfismo desde $\mathcal{S}$ a $\mathcal{S}_\sigma$ se muestra en la figura con flechas punteadas. Dada la definición de la operación de morfismo proyección (definición 7.1.1) se puede observar qué $\Sigma\sigma = \Sigma_a = \{Apagado, PrenderLuces\}$ y que también se satisfacen las demás condiciones. En particular para la eliminación existencial en **M2:** de la definición 7.1.1 se tiene que $LucesPrendidas \Rightarrow \exists ModoCaro\ LucesPrendidas \wedge ModoCaro$ es una tautología considerando $ModoCaro = true$. De la misma forma se observa que $Apagado \Rightarrow Apagado$

también es una tautología. Luego, y viendo que se satisfacen todas las condiciones, la traza puede proyectarse en el escenario. Es decir, existe g, un morfismo proyección de $\mathcal{S}$ a $\mathcal{S}_\sigma$. Además, dado que $\mathcal{S}$ satisface *r1*, la primera regla en la figura 7.1, y que además existe g, se puede concluir que $\mathcal{S}_\sigma \models r1$, siguiendo la definición 7.1.2.

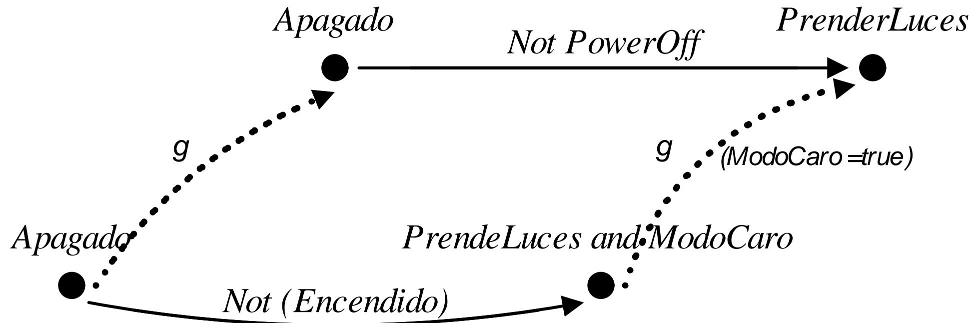

Figura 7.2: Un morfismo proyección en ω-FVS

7.2. El poder expresivo de ω-FVS

El objetivo de esta sección es dar una idea intuitiva de cómo ω-FVS puede representar cualquier lenguaje ω-regular codificando cualquier autómata de Buchi a través de un conjunto de reglas. Básicamente, dado un autómata de Buchi $\mathcal{B}$ es posible construir un conjunto de reglas ω-FVS que imiten el comportamiento de $\mathcal{B}$. En las mismas se emplean eventos fantasmas para representar la entrada y salida de cada estado de $\mathcal{B}$. Es decir, para cada estado $\mathcal{S}$ de $\mathcal{B}$ existirán eventos fantasmas "Ingreso-$\mathcal{S}$" y "Salida-$\mathcal{S}$". Con estos eventos fantasmas se definen reglas para establecer cuándo y cómo es válido ingresar y abandonar un estado, siguiendo el comportamiento definido por la función de transición de $\mathcal{B}$. Finalmente, también se introducen reglas para modelar las condiciones de aceptación. Más precisamente, existirán reglas ω-FVS que indiquen que los eventos fantasmas que representan el ingreso a un estado aceptador ocurren en toda traza, y que además, siempre que un estado aceptador es abandonado, entonces en el futuro la traza exhibe nuevamente un evento fantasma representando el ingreso a un estado aceptador, cumpliendo así la parte de que los estados aceptadores ocurren infinitamente frecuente.

A continuación se muestra un ejemplo basado en un fragmento de un autómata de Buchi $\mathcal{B}$ (ver figura 7.3), donde el objetivo final es definir reglas que imiten el comportamiento de $\mathcal{B}$.

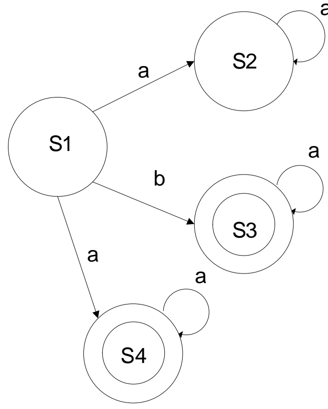

Figura 7.3: Un autómata de Buchi

Por razones de simplicidad sólo se muestran a continuación reglas para entrar y salir de *S1* (reglas muy similares pueden definirse para el resto de los estados) y reglas que modelan las condiciones de aceptación. Tales reglas se muestran en la figura 7.4. La regla 1 modela el cambio de estado que ocurre al llegar un evento *a* desde *S1* a *S2* o *S4*, tal como está establecido en las transiciones de salida de *S1* en la figura 7.3 etiquetadas con el evento *a*. Si estando en el estado *S1* ocurre un evento *a*, entonces el autómata abandona el estado *S1* para ingresar al estado *S2* (reflejado en el consecuente 1) o al estado *S4* (reflejado en el consecuente 2). Análogamente se podría definir una regla que modele la salida del estado *S1* al ocurrir un evento de tipo *b*.

Cómo las reglas pueden usarse para fijar cuándo y cómo un estado es ingresado o abandonado se muestra en reglas 2 y 3. La regla 2 especifica el contexto válido para entrar al estado *S2*: el estado previo tiene que haber sido *S1*, y tiene que haber ocurrido un evento *a*. Análogamente, la regla 3 detalla el comportamiento esperado para abandonar el estado *S1*: el autómata debe estar en el estado *S1* y ocurrir un evento *a* o *b*. Finalmente, las reglas 4 y 5 tratan con las condiciones de aceptación, tal como se mencionó previamente, tomando como estados de aceptación al conjunto {S3,S4}. En particular, la regla 4 establece que al menos un estado aceptador es alcanzado alguna vez en la traza, mientras que la regla 5 especifica el comportamiento esperado al abandonar el autómata el estado aceptador *S3*: o bien en el futuro se llegan nuevamente a estados aceptadores (*S3* o *S4*), como es reflejado en los consecuentes 1 y 2, o el autómata hace un *loop* en *S3* (consecuente 4), o se mueve al estado *S4* (consecuente 3). Vale destacar que algunos de estos consecuentes no son topológicamente posibles dado el autómata de la figura 7.3, pero son igualmente

incluidos para mostrar todos los consecuentes que son construidos. De manera similar se pueden introducir reglas para modelar el comportamiento del autómata al abandonar el estado aceptador *S4*.

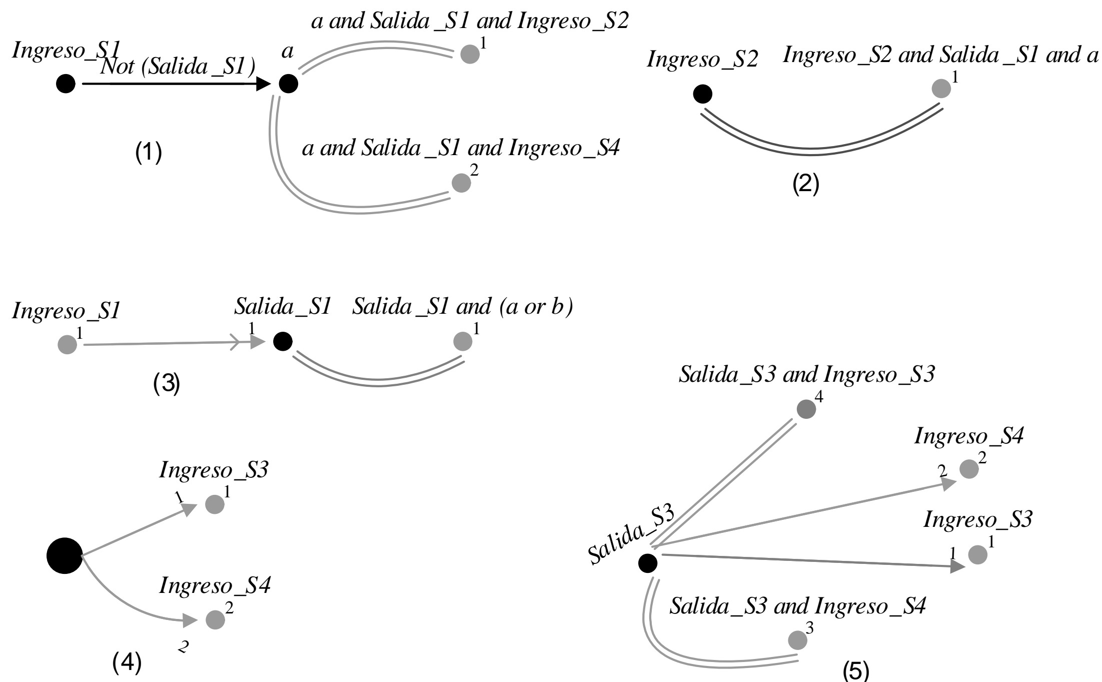

Figura 7.4: Reglas ω-FVS para modelar un autómata de Buchi

Las reglas se construyen de tal forma que las corridas aceptadoras de $\mathcal{B}$ representan al conjunto de trazas que satisfacen esas reglas. Más precisamente, cualquier corrida aceptadora de $\mathcal{B}$ puede embeberse en un escenario traza que satisface las reglas tal como se planteó en la definición 3.3.7. De manera análoga, cualquier escenario traza que satisface la regla puede embeberse en una corrida aceptadora de $\mathcal{B}$.

Para ilustrar el poder expresivo que introducen los eventos fantasmas se analizan a continuación tres ejemplos. La sección 7.2.1 muestra cómo ω-FVS puede especificar comportamiento fuera del alcance de LTL mientras que las secciones 7.2.2 y 7.2.3 muestran los eventos fantasmas en acción modelando protocolos de comunicación de software.

7.2.1. Yendo más allá del poder expresivo de LTL

Como se mencionó anteriormente, los eventos fantasmas no constituyen meramente un atajo sintáctico, sino que agregan expresividad al lenguaje. Considerar por ejemplo una propiedad como "*p debe ocurrir en los pasos pares*". Este tipo de propiedades no son

expresables en lógicas temporales como LTL, y se conocen como "LTL no sabe contar" [113, 92]. En ω-FVS se pueden utilizar los eventos fantasmas para "contar". Para modelar esta propiedad se introduce un evento fantasma "Par" para registrar si el número de pasos actual en la ejecución es par o no. Los pasos impares se representarán con la formula "Not Par". Las reglas en la figura 7.5 proveen una definición para establecer la paridad. La primera aparición del evento "Paso" se marcará como "Not Par". Desde ese momento, las ocurrencias sucesivas del evento "Paso" se modelarán con "Par" o "Not Par", según corresponda y de manera alternada.

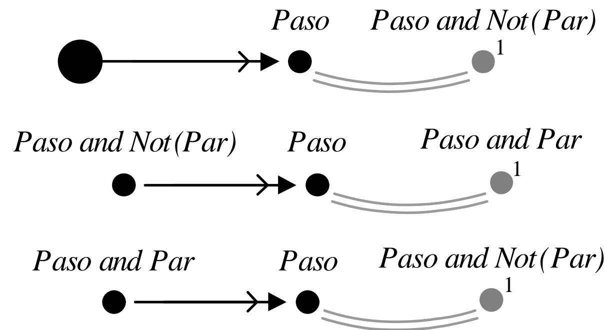

Figura 7.5: Contando en ω-FVS

Dada esta definición la propiedad deseada, que razonaba sobre las ocurrencias pares, pueden ser especificada, tal como se ve en la regla de la figura 7.6. La misma establece que p debe ocurrir en cada paso par.

Figura 7.6: p debe vale en cada paso par

7.2.2. Ejemplo 1: Protocolo de Comunicación SMB2

En esta sección se analizan algunas características del protocolo SMB2. Este protocolo, como está detallado en [109], es un sucesor del protocolo de Windows Cliente-Servidor para compartir archivos denominado SMB, el cual se emplea para compartir archivos entre plataformas con Windows Vista con otras plataformas. En términos generales, este protocolo sigue el esquema clásico cliente-servidor, donde los clientes realizan pedidos

al servidor, quien les responde. Siguiendo la estrategia que se viene empleando en este trabajo con los demás ejemplos, se especifica primero una versión inicial, a la cual se le agregan luego características avanzadas. En este caso la propiedad avanzada a introducir es un mecanismo de *ventana deslizante* para organizar y sincronizar los pedidos por parte del cliente. Al modelar este mecanismo se verá en acción el poder de los eventos fantasmas, y se analizará la especificación resultante teniendo en cuenta la versión propuesta en [109].

De toda la funcionalidad del protocolo esta sección se enfocará en el pedido de servicios por parte del cliente. En este contexto se puede pensar en dos eventos concretos disponibles: **Pedido**, representando un pedido por parte de un cliente, y **Respuesta**, representando la respuesta del servidor al pedido del cliente. En principio se modelarán dos requerimientos:

- *Todo pedido recibirá una respuesta*
- *Toda respuesta debe ser precedida por un pedido*

Tales requerimientos son especificados con las reglas de la figura 7.7.

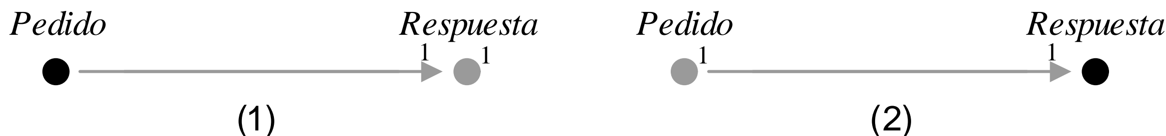

Figura 7.7: Todo pedido recibe una respuesta (1) y toda respuesta es precedida por un pedido(2).

Ahora se procederá a mejorar este esquema de pedidos y respuestas en el protocolo introduciendo un algoritmo de *ventana deslizante*. El cliente puede pedir por créditos al enviar un pedido, para poder aumentar el tamaño de la ventana. El servidor puede otorgar créditos en su respuesta. El número de créditos otorgado en una respuesta determina cómo el tamaño de la ventana crece o disminuye a medida que pasa el tiempo. Para su modelado en ω-FVS se abstraerá la noción de ventana deslizante considerando la cantidad de créditos, que indica el tamaño de la misma. Luego, el comportamiento queda determinado de la siguiente manera. El cliente puede efectuar tantos pedidos como créditos tenga. En cada respuesta, el servidor puede otorgar más créditos. Cada vez que el cliente efectúa un pedido, su cantidad de créditos disminuye en uno. Tomando en cuenta un caso simple con un tamaño máximo de 2 créditos, el evento respuesta ahora es modelado con dos eventos:

RespuestaOtorga1, donde el servidor responde otorgando un crédito, y *RespuestaOtorga2*, donde el servidor responde otorgando dos créditos. Para modelar el comportamiento de los créditos se introducen los siguientes eventos fantasmas: *Credito0*, *Credito1* and *Credito2*, cada uno representando las distintas cantidades de crédito el cliente puede llegar a tener durante la ejecución. También existe un evento *Comienzo* para representar el inicio del protocolo de comunicación. Dado este contexto, las reglas de la figura 7.8 modelan el comportamiento de la ventana deslizante siguiendo los siguientes requerimientos:

R1: *Los clientes tienen un crédito al comienzo del protocolo*

R2: *Los pedidos sólo pueden ocurrir si el cliente tiene al menos un crédito.* Este requerimiento implica que el cliente no puede efectuar pedido si tiene cero créditos.

R3: *Los créditos del cliente disminuyen en uno cada vez realiza un pedido.*

R4: *Los créditos del cliente aumentan sólo de acuerdo a la cantidad de créditos otorgada por el servidor.* Este requerimiento sigue el siguiente razonamiento: si se produce una respuesta con 1 crédito y como resultado el cliente resulta con n créditos, entonces obligatoriamente el cliente tenía antes de recibir la respuesta *n-1* créditos. Por simplicidad, se considera únicamente el caso cuando el servidor otorga un crédito (las reglas para cuando otorga cero o dos créditos son muy similares).

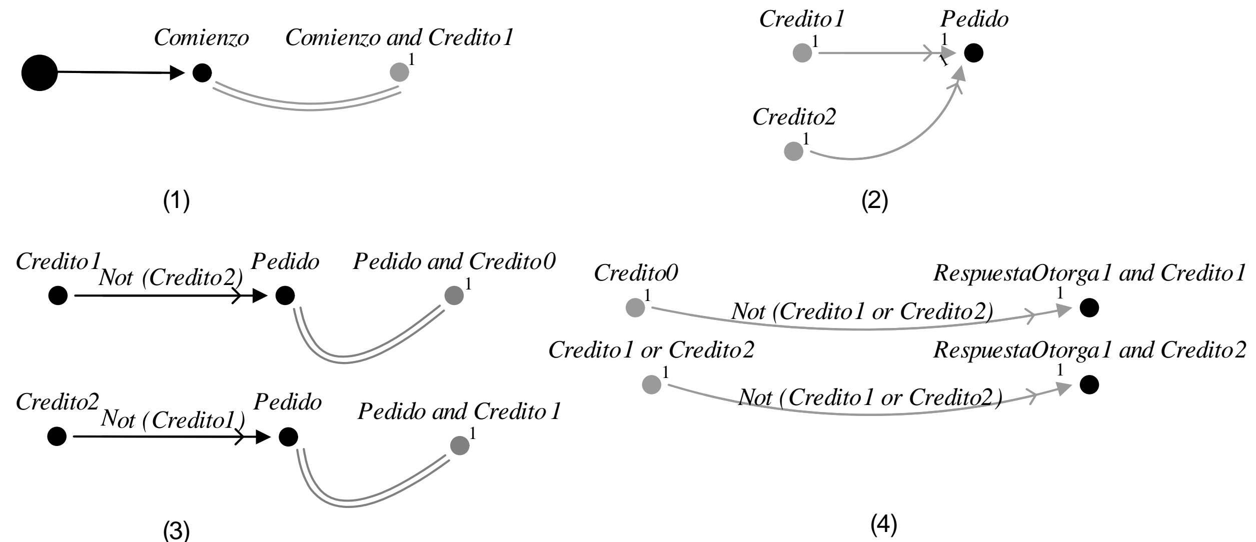

Figura 7.8: Reglas para el protocolo de Ventana Deslizante

Evitando la inanición del cliente

La figura 7.9 se concentra en un problema clásico de los protocolos distribuidos, como es la inanición o *starvation* en inglés. En el protocolo siendo modelado este problema ocurre cuando el cliente obtiene cero créditos y no tiene oportunidad de recibir más créditos en el futuro. Para evitar la inanición se propone el siguiente requerimiento:
R5: *si el cliente envía un pedido y como resultado se queda sin créditos, entonces eventualmente recibirá más créditos en el futuro*

Este nuevo requerimiento está modelado por la regla en la figura 7.9, como se mencionó anteriormente. Es importante notar que esta regla elimina aquellas trazas donde se envía una respuesta con cero créditos a un cliente que no tiene pedidos pendientes. Este hecho se modela implícitamente por la regla, mientras que en la especificación propuesta en [109] primero se tuvo que detectar este comportamiento inapropiado y luego agregar una nueva propiedad para solucionarlo.

Figura 7.9: Evitando la inanición

7.2.3. Ejemplo2: Protocolo de audio Philips

Como segundo ejemplo se analizará parte del comportamiento del protocolo de audio Philips [26]. Este protocolo fue tomado también como caso de estudio en [27], donde una propiedad interesante del protocolo fue modelada. En esta sección se especifica la misma propiedad combinando la capacidad de ω-FVS para modelar comportamiento directamente de la descripción en lenguaje natural de los requerimientos con la habilidad de ω-FVS para imitar el comportamiento de autómatas. A continuación se describe el protocolo de audio, y luego se especifica la propiedad de interés en ω-FVS.

El protocolo de audio se compone de un emisor y un receptor conectados por un cable, donde streams de bits son enviados utilizando *Codificación Manchester*. En esta codificación se envía un *1* subiendo el voltaje en el medio del slot del bit, mientras que un *0* se envía de la manera opuesta: el voltaje es alto en la primera mitad y bajo en la segunda. Las variaciones en los voltajes se modelan con los eventos *up* and *down*. De

manera similar, los eventos *in* y *out* modelan respectivamente el pedido para transmitir un stream de bits y su correspondiente salida. Como en [27, 26] se asume en lo que sigue que cualquier evento *in* que ocurre antes del evento *out* que le corresponde a un mensaje anterior lleva al protocolo un *estado caótico* donde se permite cualquier comportamiento. Luego, el comportamiento a ser especificado es únicamente válido cuando el protocolo se encuentra trabajando en un modo *no caótico*.

La propiedad sobre el comportamiento del protocolo a modelar es la siguiente: *cuando el protocolo está en modo no caótico, todo stream de bits recibido es igual al enviado*. Para satisfacer esta propiedad se deben cumplir dos condiciones:

- Condición 1: siempre que el receptor produce un evento *out* en un estado no caótico entonces necesariamente tiene que haber ocurrido un evento *in* en el pasado cuando el emisor empezó a transmitir el bit de streams.

- Condición 2: desde la última ocurrencia del evento *in* el sistema tiene que haber transmitido de manera correcta el bit de streams siguiendo la codificación Manchester.

Ambas condiciones son modeladas en [27] usando una lógica temporal denominada $BTATL_p$, la cual combina lógicas temporales con notaciones de autómatas en las fórmulas. Para mostrar la versatibilidad de ω-FVS se modelará la primera condición introduciendo reglas directamente a partir de los requerimientos expresados en lenguaje natural, mientras que para la segunda condición se imitará el comportamiento de un autómata de Buchi propuesto en [27] para describir su comportamiento. Finalmente, ambas condiciones serán combinadas para especificar la propiedad.

Para modelar la primera condición se deben caracterizar los estados caóticos y no caóticos del protocolo. Para esto se introduce el evento fantasma *caótico*. Cada ocurrencia de los eventos *in* y *out* se catalogará como caótica o no caótica, de acuerdo a lo establecido en los requerimientos, que exigen una alternancia estricta entre los eventos. Como se mencionó anteriormente, el protocolo entra en un modo caótico cuando dos eventos consecutivos *in* ocurren sin la ocurrencia de un evento *out* en el medio, y viceversa. Las reglas en la figura 7.10 caracterizan el modo caótico para los eventos *in* mientras que los eventos en la figura 7.11 lo hacen para los eventos *out*. Como lo ilustra la figura 7.10 la primera ocurrencia del evento *in* se asocia con la formula *Not caótico*. De ahí en más las

próximas ocurrencia de *in* serán asociadas con *caótico* o *Not caótico* según haya ocurrido el correspondiente evento *out* en el medio o no.

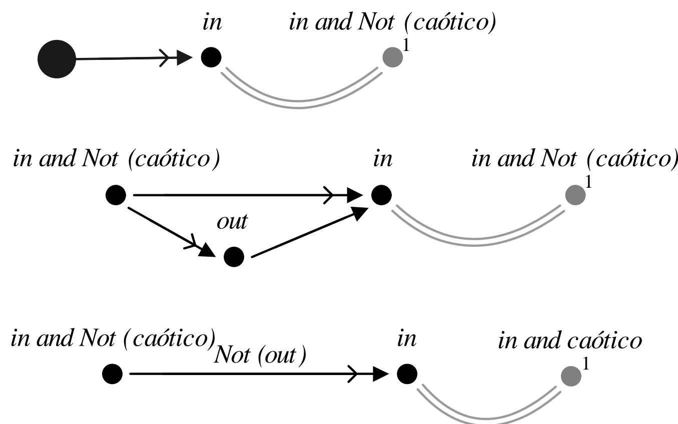

Figura 7.10: Caracterizando el modo no caótico de acuerdo a los eventos in

Por otro lado, las reglas de la figura 7.11 caracterizan el modo caótico teniendo en cuenta los eventos *out*. La primera ocurrencia de este evento será *caotico* o *Not caotico* según haya ocurrido antes o no en la traza un evento *in*. De manera similar se procede con las demás ocurrencia del evento *out*, clasificándolas como *caotico* o *Not caotico*.

Con estas reglas de definen por completo los modos de ejecución del protocolo. Luego, el paso siguiente es modelar una transmisión Manchester correcta, tal como es requerido por la condición 2.

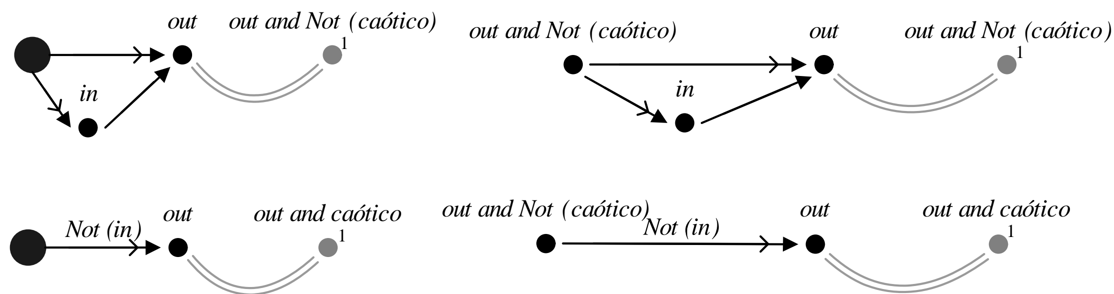

Figura 7.11: Caracterizando los modo caótico y no caótico para los eventos out

Modelar el comportamiento de una transmisión correcta de acuerdo a la codificación Manchester implica asegurarse que el stream de bits ha sido enviado siguiendo la codificación y validar que la misma ha sido correctamente recibida. Este comportamiento se modeló [27] con el automata de Buchi en la figura 7.12.

En pocas palabras, el comportamiento del automata es el siguiente (para una descrip-

ción más completa dirigirse a [27]). El etiquetado en cada estado S_i debe interpretarse como un conjunto de eventos que deben ocurrir de manera simultánea. Se utiliza un reloj para medir el tiempo transcurrido entre el último cambio de voltaje en el cable de comunicación, representado por los eventos *clock2* (dos unidades de tiempo) y *clock4* (cuatro unidades de tiempo). Los eventos *sen0*, *sen1*, *rec0* and *rec1* representan respectivamente el envío de unos y ceros. Para poner en concreto un ejemplo del comportamiento del autómata, cuando en el estado $S1$ ocurre un evento *down* luego de cuatro unidades de tiempo desde la última vez que cambió el voltaje la codificación indica que debe enviarse un 0. Luego, el autómata se mueve al estado $S2$, y se envía un *0* por el cable, representado por el evento *rec0*.

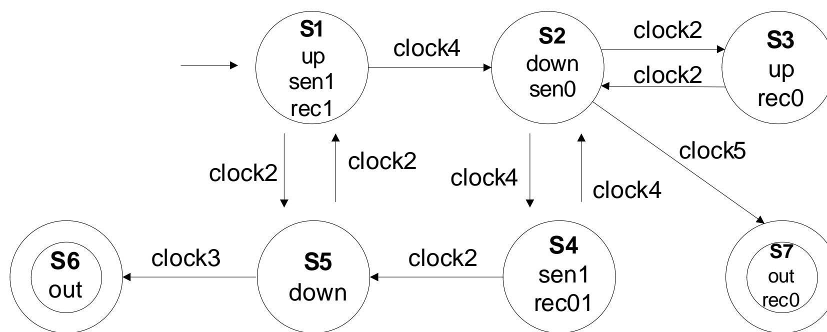

Figura 7.12: Un autómata de Buchi modelando una codificación Manchester

Como se explicó en el comienzo de esta sección la presencia de los eventos fantasmas permite que ω-FVS pueda imitar el comportamiento de un autómata. En este sentido, las reglas de la figura 7.13 modelan parte del autómata de la figura 7.12, enfocándose principalmente sobre el estado $S2$.

Por razones de simplicidad, se utiliza la notación $Etiquetado_S_i$ para representar el etiquetado en cada estado. Por ejemplo, *Etiquetado_S2* simboliza la ocurrencia simultánea de los eventos *sen0* y *down0*.

El modo no caótico se garantiza en las reglas ya que existe una restricción presente en cada regla que representa un cambio de estado en el autómata. Las reglas 1 y 2 describen las transiciones de los estados S1 y S4 al estado S2. Las reglas 3 y 4 reflejan las condiciones válidas para entrar al estado S2, mientras que la regla 5 describe las condiciones para abandonar el estado S2.

Con lo modelado anteriormente ya se encuentran especificadas las dos condiciones requeridas por la propiedad a modelar. Por lo tanto, ya es posible describir su comporta-

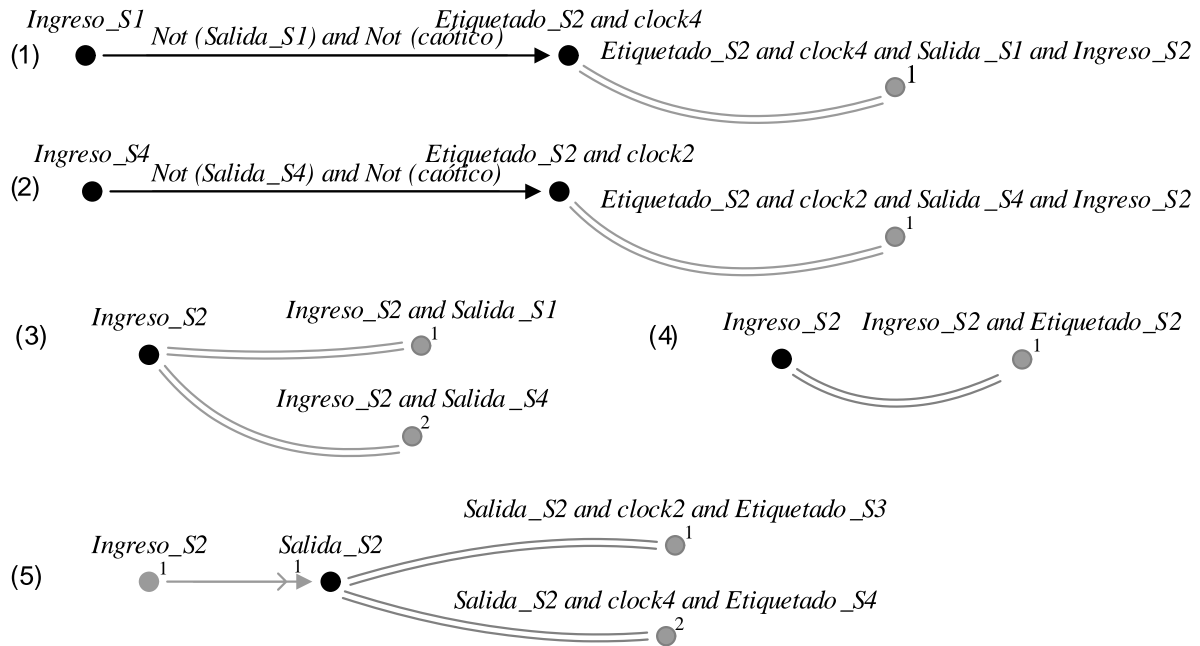

Figura 7.13: Reglas que modelan el comportamiento del autómata propuesto

miento, tal como lo expresa las reglas de la figura 7.14. Básicamente la misma dice que cada ocurrencia de un evento *in* en un modo no caótico debe ser seguido por una transmisión correcta. Esto se refleja en la primera regla de la figura 7.14 incluyendo el evento $Ingreso_S1$ en el consecuente de la regla. Una vez ingresado en el estado inicial, las reglas de la figura 7.13 aseguran que una transimisión correcta es recibida apropiadamente. La segunda regla en la figura 7.14 establece que todo evento *out* en un modo no caótico debe ser precedido en la traza por un evento *in*. De esta manera queda especificada la propiedad.

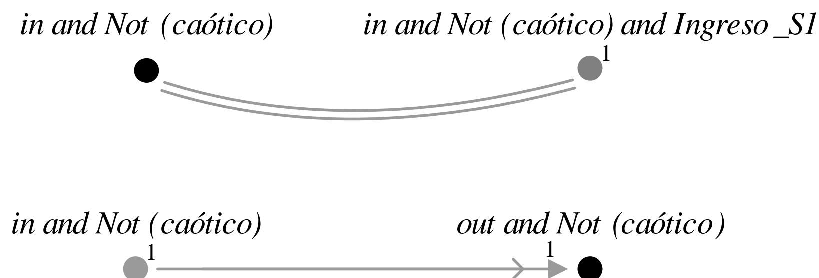

Figura 7.14: Reglas para modelar la propiedad deseada

Capítulo 8

Proceso de Síntesis

Este capítulo se concentra en describir un proceso de síntesis de comportamiento, donde se exhibe el algoritmo de tableau que traduces reglas de ω-FVS a autómatas de Buchi, se demuestra la correctitud del algoritmo y se analiza la interacción entre reglas y autómatas.

8.1. Síntesis

En esta sección se describe el proceso de síntesis que traduce reglas en autómatas de Buchi. En pocas palabras el funcionamiento del proceso es el siguiente. Suponer que se tiene la especificación de una regla que describe el comportamiento del sistema. Aplicando el algoritmo de tableau (que será descrito en la sección siguiente), dicha regla es traducida a un autómata de Buchi, donde no se distinguen elementos fantasmas. Luego, se aplica eliminación existencial al autómata generado para descartar los eventos fantasmas, como se vio en la definición 2.2.3. Cuando se tiene un conjunto de reglas en vez de una única regla, los autómatas generados para cada regla se componen, y luego se aplica eliminación existencial al autómata resultado de la composición. En cualquier caso, el autómata final obtenido es tal que acepta los escenarios traza que satisface la o las reglas. Esto será explicado con mayor profundidad en la sección 8.2, al analizar de manera integral el proceso de síntesis.

El resto de la sección se organiza como sigue. Luego de introducir algunas definiciones necesarias el algoritmo de tableau se presenta en la sección 8.1.2. La sección 8.1.3 discute la correctitud y completitud del algoritmo de tableau mientras que la sección 8.2 analiza

el proceso de sínteis de punta a punta.

8.1.1. Definiciones

Para poder describir el algoritmo de tableau es necesario introducir previamente algunas definiciones. Una *configuración* θ de un escenario $\mathcal{S}$ es un subconjunto de $P \uplus \{\infty\}$ tal que θ es cerrado a izquierda bajo la relación $< \cup \equiv$. Dado un morfismo $e : A^{'} \rightarrow A$, se dirá que un escenario $A^{'}$ es una *configuración* de A si y sólo si $e(A^{'})$ es una configuración de A. La intuición detrás de estas definiciones es que $A^{'}$ representa una isntanciación parcial del escenario A.

$\mathcal{F} \subseteq P$ es un paso simple en una configuración θ por un minterm $\mathcal{A}$ denotado $\theta \xrightarrow{\mathcal{A}} \theta \cup \mathcal{F})$ si y sólo si $\theta \cup \mathcal{F}$ es una configuración y $(\forall \mathsf{p} \in F - \{\infty\})\mathcal{A} \Rightarrow \ell(\mathsf{p})$ es una tautología. En otras palabras, un paso en una configuración representa un avance o progreso en el matching del escenario.

Se define el conjunto de *eventos activos restringidos* de una configuración θ, $AR(\theta)$ como sigue. $\langle \mathsf{p}, \mathsf{q} \rangle \in AR(\theta)$ si y solo si $\mathsf{p} \in \theta$ y $\mathsf{q} \notin \theta$. Se define $\tau(\theta) = \bigwedge \gamma(\mathsf{p}, \mathsf{q})$ para todo $\langle \mathsf{p}, \mathsf{q} \rangle \in AR(\theta)$ y $\mathcal{R}(\theta)$ a aquellos minterms m sobre Σ tal que $m \wedge \tau(\theta)$ es satisfacible. Más aún, el conjunto de *restricciones estrictamente activas* para una configuración θ con respecto a conjunto de puntos $\mathcal{F}$ se define como $SAR(\theta, \mathcal{F}) = \{\langle \mathsf{p}, \mathsf{q} \rangle \in AR(\theta) \wedge \mathsf{q} \notin \mathcal{F}\ \}$. Se denomina $\xi_{\mathcal{F}}(\theta) = \bigwedge \gamma(\mathsf{p}, \mathsf{q})$ para todo $\langle \mathsf{p}, \mathsf{q} \rangle \in SAR(\theta, \mathcal{F})$. Luego, se define el conjunto de *formulas estrictamente activas* $\mathcal{R}_{\mathcal{F}}(\theta)$ a aquellos minterms m sobre Σ tal que $m \wedge \xi_{\mathcal{F}}(\theta)$ es satisfacible. Intuitivamente todas estas restricciones establecen avances válidos en una configuración.

A continuación se define la noción de **situación**, un concepto clave que representa para una dada regla todas las posibles combinaciones de matching parciales del antecedente a cada consecuente.

Definition 8.1.1 (Situación)**.** *Dados*

- $f_j : A \rightarrow C_j$, $j \in [1..n]$, *representando una regla con antecedente* A *y consecuentes* $C_1, C_2, ..., C_n$.
- $e : A^{'} \rightarrow A$, *un morfismo representando un matching parcial* $A^{'}$ *del antecedente* A
- *:* $e_j^m : C_j^m \rightarrow C_j$, $m \in [1..k_j]$, *morfismos representando matchings parciales de cada consecuente* C_j

una ***situación*** η *es un conjunto indexado de morfismos tal que* η_i *retorna todos los matching parciales para el consecuente* C_i*. Más formalmente, una* ***situación*** η *es un conjunto indexado de morfismos* $g_j^i : A' \to C_j^i$*,* $i \in [1..k_j]$ *tal que se cumplen las siguientes condiciones:*

- *Condición 1:* A' *es una configuración para* A *y* $e_j^i(C_j^i)$ *es una configuración para* $C \setminus \{f_j(e(A'))\}$*. Esto es,* A' *y cada* $e_j^i(C_j^i)$ *representan instanciaciones parciales del antecedente y consecuentes de la regla.*

- *Condición 2:* $f_i \circ e = e_j^i \circ g_j^i$*. Esto implica que los morfismos* g_j^i *son consistentes con los matchings parciales del antecedente y consecuente de la regla.*

- *Condición 3:* $e_j^i(C_j^i) \cap f_j(A) = f_j(e(A'))$*. Los matching parciales del antecedente y los consecuentes son completos. Esto es, ningún punto ha quedado "afuera" de cualquiera de los matchings parciales.*

Para poner un ejemplo de una *situación* η considerar la figura 8.1. En este caso se muestra un ejemplo con una regla con dos consecuentes, y donde existen tres matchings parciales para el consecuente uno, y dos para el consecuente dos. Por lo tanto η_1 consiste de los tres morfismos de la primer columna (g_1^1, g_1^2, g_1^3) mientras que η_2 consiste de los dos morfismos en la segunda columna (g_2^1, g_2^2).

$$g_1^1 : A' \longrightarrow C_1^1 \qquad g_2^1 : A' \longrightarrow C_2^1$$
$$g_1^2 : A' \longrightarrow C_1^2 \qquad g_2^2 : A' \longrightarrow C_2^2$$
$$g_1^3 : A' \longrightarrow C_1^3$$

Figura 8.1: Un ejemplo de una situación

En este punto ya se cuentan con todas las definiciones necesarias para presentar el algoritmo de tableau, el cual traduce reglasω-FVS en autómatas de Buchi.

8.1.2. Algoritmo de Tableau

Dada una regla R, el algoritmo de tableau construye un autómata de Buchi $\mathcal{B} = \langle \Sigma, S, S^0, \Delta, F \rangle$ tal que Σ representan minterms sobre Σ_R y el conjunto de etados S son

3-uplas $(\Upsilon_R \times \mathit{bool} \times \mathcal{PL}(\Sigma_R))$. El conjunto Υ_R asociado a un estado S (un conjunto de situaciones η) denotado *situaciones*(S) representa simbólicamente todas las posibles combinaciones de matchings parciales obtenidos hasta ese estado desde el antecedente a cada consecuente. El segundo término de la 3-upla identifica los estados aceptadores. Esta variable booleana se seteará en **true** cuando el comportamiento denotado en la regla esté completamente matcheado y convertirá al estado en un estado aceptador efímero. Más adelante se detalla en profundidad el manejo de los estados aceptadores. Finalmente, el tercer término se necesita para mantener las obligaciones a futuro de la traza. Este elemento es particularmente útil cuando las reglas predican sobre condiciones que deben mantenerse hasta el final de la traza.

El pseudo código del algoritmo 1 computa los estados sucesores para la relación de transición Δ. Partiendo del estado inicial $(\langle \emptyset, false, true \rangle)$, el autómata intentará de manera incremental “construir” el comportamiento denotado en la regla a medida que los eventos, representados por minterms, vayan ocurriendo. Para cada minterm, el algoritmo 1 computa todos los posibles matchings considerado los matchings en el antecedente y también en cada consecuente. Esto se lleva a cabo a través de dos algoritmos auxiliares en *avanzarAntecedente*, invocado en la línea 5 y *avanzarConsecuente*, invocado en la línea 6. La línea 7 analiza si algún estado sucesor llega a una situación *trampa*, una situación donde el antecedente ha sido matcheado por completo (es decir, existe un morfismo tal que $A^{'} = A$), pero es imposible matchear alguno de los consecuentes (es decir, todo C_j^i no es una configuración de C_j)

Las líneas 8 y 9 controlan si alguno de los consecuentes ha sido completado en alguno de los sucesores. Esto implica que $goalmatched[i] = true$ si y sólo si se ha completado el consecuente C_i.

La línea 10 analiza si el próximo estado es un estado aceptador. Para serlo debe cumplir que no está en una situación de trampa, y haber completado uno de los consecuentes de la regla.

Finalmente, la línea 11 retorna la salida esperada. Aquellos consecuentes que tienen restricciones hasta el final de la traza, restricciones contra ∞, merecen un tratamiento especial. Para entender su comportamiento considerar una propiedad que predica sobre la última ocurrencia de un dado evento. Este evento tendrá en consecuencia una restricción hasta el final de la traza (es decir, hasta ∞) diciendo que el evento no volverá a ocurrir

hasta el final de traza. En cada ocurrencia del evento el algoritmo no tiene forma de saber si efectivamente esa ocurrencia será la de interés (la última de la traza), o si en cambio la debe ignorar y volverá a ocurrir. La manera en que el algoritmo resuelve esto es través del no determinismo. En un estado se considerará la situación en donde se toma el evento cómo el último a ocurrir, estado en el cual se introducirá la restricción hasta ∞ diciendo que el evento no debe ocurrir nuevamente, mientras que en un segundo estado se considerará la posibilidad de que el evento vuelva a ocurrir. En otras palabras, en aquellos casos donde un consecuente que mantiene restricciones hasta ∞ es completado se introduce no determinismo. En un camino se considera al evento cómo el último y las restricciones se agregan como obligaciones futuras que deben ser satisfechas en la traza hasta el final de la ejecución. Y por el otro camino, no se considerará por completo al escenario, esperando una ocurrencia del evento de interés. Si más de un consecuente es matcheado, se consideran todas las posibles situaciones. Concretamente en el algoritmo el no determinismo se introduce por la condición $GM \rightarrow goalMatched$ (línea 11). Cuando esta condición es verdadera dos estados son agregados, uno con *GM=true* y otro con *GM=falso*. En el primer caso el consecuente es completado y las obligaciones del consecuente se agregan como futuras obligaciones de la traza. El segundo caso se corresponde con el caso cuando se decide esperar por una nueva ocurrencia.

Este algoritmo puede optimizarse para aquellos casos donde el consecuente completado no posee restricciones hasta ∞. El cálculo de la salida en estas circunstancias es simple: a la variable *GM* le es asignado el valor *true* (el consecuente se ha completado), así como también es asignado el mismo valor a la variable *Obligations* (el consecuente no posee restricciones). La siguiente línea $\exists$ j $(goalmatched[j])) \wedge \mathcal{R}(C_j) = true$ realiza este control. Luego, si la misma resulta verdadero se procede con las asignaciones como recién se describió, y en caso de ser falsa, se repiten los pasos en la línea 11 del algoritmo 1.

Los algortimos 2 y 3 describen los procedimientos auxiliares *avanzarAntecedente* y *avanzarConsecuente*.

Estados aceptadores efímeros

Cuando el comportamiento de una regla es encontrado, el autómata construido por el tableau entra en un estado aceptador *efímero*. Este comportamiento efímero se introduce para hacer explícito el hecho de que se ha completado un consecuente. Una vez llegado a

1 *Algoritmo Sucesores*$(S : State, m : minterm)$: set of states;

2 *Precondición*: $m \wedge obligations(S)$ es satisfacible;

3 $newSits := \emptyset$;

4 **foreach** $\eta \in Situations(S)$ **do**

5 $\quad newSits := add(newSits, avanzarAntecedente(\eta, m))$;

6 $\quad newSits := add(newSits, avanzarConsecuente(\eta, m))$;

7 $trapSituation : \exists \eta \in newSits\ \forall i \forall j \in [1..n]\ g^i_j : A' \rightarrow C^i_j \in \eta \wedge A' = A \wedge C^i_j$ no es una configuración de C_j;

8 **foreach** $j \in [1..n]$ **do**

9 $\quad goalmatched[j] := \exists \eta \in situations(S) \wedge$
$\quad g^i_j : A' \rightarrow C^i_j \in \eta \wedge C^i_j \xrightarrow{m} C_j \cup \mathcal{F}^i_j \wedge m \in (R_F(C^i_j)) \wedge C^i_j \cup \mathcal{F}^i_j = C_j$;

10 $goalMatched := (\exists$j $(goalmatched[j])) \wedge (\neg trapSituation)$;

11 return { $\langle newSits, GM, Obligations \rangle$ tal que
$GM \rightarrow goalMatched \wedge GM = true \rightarrow \exists j goalmatched[j] \wedge Obligations =$
$Obligations(S) \wedge \bigwedge_{j \in I} \mathcal{R}(C_j) \wedge GM = false \rightarrow Obligations = Obligations(S)$

Algorithm 1: Algoritmo Sucesores

1 *Algoritmo avanzarAntecedente*$(\eta : situation, m : minterm)$: η situation ;

2 $return\ \{z^i_j : A' \cup \mathcal{F}^i_j \rightarrow C^i_j \cup \{[f(\mathcal{F}^i_j)]_\equiv\}$ tal que: $g^i_j : A' \rightarrow C^i_j \in \eta \wedge A' \xrightarrow{m} A' \cup \mathcal{F}^i_j \wedge$
$C^i_j \xrightarrow{m} C^i_j \cup \{[f(\mathcal{F}^i_j)]_\equiv\} \wedge g^i_j(A') = z^i_j(A') \wedge z^i_j(\mathcal{F}) = f(\mathcal{F}) \wedge m \in \mathcal{R}_{[f(\mathcal{F}^i_j)]_\equiv}(C^i_j)\} \cup$
$\{g^i_j : A' \rightarrow C^i_j$ tal que $m \in \mathcal{R}(A')\}$

Algorithm 2: Algoritmo avanzarAntecedente

1 *Algoritmo avanzarConsecuente*$(\eta : situation, m : minterm)$: η situation;

2 $return\{z^i_j : A' \rightarrow C^i_j \cup \mathcal{F}^i_j$ tal que: $g^i_j : A' \rightarrow C^i_j \in \eta \wedge C^i_j \xrightarrow{m} C^i_j \cup \mathcal{F}^i_j \wedge g^i_j(A') = z^i_j(A') \wedge$
$m \in \mathcal{R}_{\mathcal{F}^i_j}(C^i_j)\} \cup \{g_i : A' \rightarrow C^i_j$ tal que $m \in \mathcal{R}(C^i_j)\}$

Algorithm 3: Algoritmo avanzarConsecuente

este estado la ejecución continúa con una transición interna para que ninguna acción del sistema quede sin contemplar. Para lograr este efecto efímero sobre los estado aceptadores se introducen nuevas reglas de transición: $\langle \Upsilon, true, O\rangle \xrightarrow{\lambda} \langle \Upsilon, false, O\rangle$. Finalmente, se define a un estado como aceptador si y sólo si ∃j ($goalMatched[j]$) o para cada situación $\eta \in$ Situations(S) tal que $A^{'} = A \rightarrow \exists i\ \exists j\ g^i_j : A^{'} \rightarrow C^i_j \in \eta\ (C^i_j = C_j) \wedge obligations(S) \rightarrow \mathcal{R}(C_j)$. Es decir, si se ha completado el comportamiento de la regla (una regla que se acaba de completar), o si cada situación verifica que siempre que se completó el antecedente, también se completó el consecuente (una regla ya completada en el pasado).

Ejemplo

A continuación se brinda un ejemplo que ilustra el funcionamiento del algoritmo y por ende, cómo se construye el autómata de Buchi a partir de una regla. La regla de entrada será en este caso la regla de la figura 7.7-1, que modelaba el caso básico del patrón respuesta, donde todo pedido debe corresponderse con una respuesta. En la figura se utilizará la letra P para representar el evento *Pedido* y la letra Q el evento respuesta. La figura 8.2 muestra la construcción parcial e incremental del autómata considerando sólo dos posibles transiciones. En el estado inicial existe únicamente la transición vacía. Desde este estado, si el evento P ocurre, entonces la situación inicial avanza, ya que tanto el antecedente como el consecuente de la regla lo hacen. Surge entonces una nueva situación, reflejando el hecho de que el evento P ha sido matcheado o visto en la traza. En esta situación se tiene que se ha completado el antecedente de la regla (que es simplemente la ocurrencia del evento P), pero no se ha completado el consecuente (la respuesta todavía no ha sido vista). Por esto es que el estado es un estado no aceptador. Desde este segundo estado, si ocurre un evento Q, entonces la situación avanza al completarse el consecuente. Dado que el algoritmo considera todas los posibles matchings la situación contempla dos posibles casos: este se trata efectivamente del evento Q deseado, por lo que P y Q ya han sido matcheados, o se decide ignorar al evento Q, por lo que sólo se matchea hasta el momento el evento P. Este último caso considera que esta ocurrencia de Q no es la indicada, y espera por una siguiente. En esta punto es importante notar que este nuevo estado contiene una situación donde la regla es satisfecha: tanto el antecedente como el consecuente han sido completados. Luego, este estado es considerado aceptador.

Finalmente, la figura 8.3 completa el ejemplo mostrando el autómata construido por

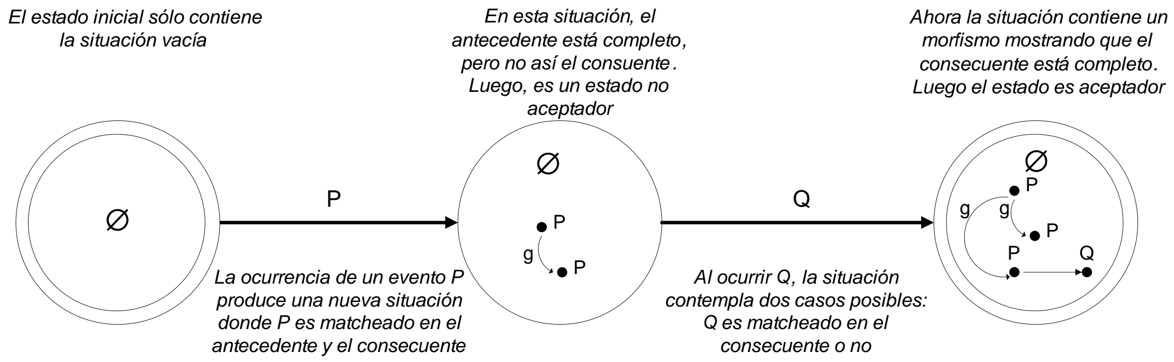

Figura 8.2: La situación inicial evoluciona contemplando todos los casos

el tableau para la regla mencionada.

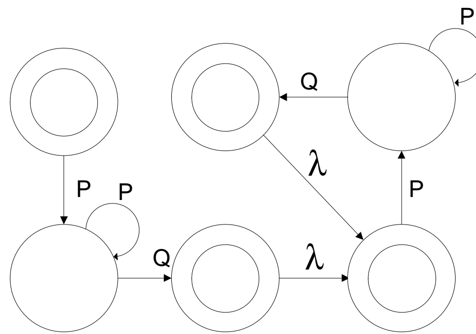

Figura 8.3: El autómata construido por el tableau para el patrón Respuesta

8.1.3. Correctitud y Completitud del Tableau

En esta sección se brindan detalles sobre la correctitud y completitud del algoritmo de tableau presentado anteriormente. El objetivo es demostrar, dada una regla R, que el lenguaje aceptado por el autómata que construye el tableau para R es exactamente igual al lenguaje denotado por las trazas que satisfacen R. Esto implica que la semántica de ω-FVS y el algoritmo de tableau conmutan, como será visto en la sección 8.2.

Para cumplir con el objetivo planteado se introduce primero el lema 8.1.2 sobre el cual se basará la prueba de correctitud. Este lema relaciona las trazas del sistema y los estados del autómata. En esencia, el lema dice que cualquier traza que lleva a un estado dado puede ser matcheada (embebida), como mínimo, en una de las situaciones de ese estado. Además, el matching es maximal respecto a los eventos involucrados.

Lemma 8.1.2 (Caracterización Estados del Tableau)**.** *Dada una regla* R*:* $f : A \to C$*,*

morfismos $e : A^{'} \rightarrow A$, $e^{i} : C^{i} \rightarrow C$, $i \in [1..k]$, *un estado* S *en el autómata construido por el tableau, y una traza* t *que lleva al estado* S*, entonces* $\exists$ *una situación* η , $\eta \in S$, *incluyendo morfismos* $g_i : A^{'} \rightarrow C^{i}$, $i \in [1..k]$ *y* $\exists h : A^{'} \rightarrow t$, $h_i : C_i \rightarrow t (i \in [1..k])$ *tal que se cumplen las siguiente condiciones:*

- *Condición 1:* $h_i \circ g_i = h$

- *Condición 2:* $\forall w : A^{'} \rightarrow C^{'}, e^{'} : C^{'} \rightarrow C$, *donde* w,e,$e^{'}$ *y* f *satisfacen las condiciones 1, 2 y 3 de la definición 8.1.1, si* $\exists h^{'} : C^{'} \rightarrow t$ *tal que* $h^{'} \circ g^{'} = h$ *entonces* $\exists i$ $(h^{'} = g_i)$.

En el lema 8.1.2 los morfismos h y cada h_i son los morfismos en una situación del estado que muestran el matching a la traza. Las dos condiciones del lema caracterizan las trazas que llevan a un dado estado del autómata.

La primera une la traza que lleva al estado con un morfismo incluido en una de las situaciones de ese estado. La segunda requiere que dicho matching sea maximal respecto a los eventos involucrados, ya que garantiza que cualquier matching posible a la traza está incluido en uno de los morfismos de la situación. Este lema se satisface por construcción, considerando la definición de situación (definición 8.1.1) y cómo las situaciones evolucionan cuando se produce un cambio de estado en el autómata según lo describe el algoritmo de la sección 8.1.2. Para favorecer la legibilidad de esta sección la demostración detallada de este lema se encuentra en el apéndice A.

A continuación se esquematiza la prueba de correctitud del tableau, la cual se basa en el lema 8.1.2, recién presentado. Formalmente, dada una regla R y un escenario traza $\mathcal{S}_\sigma$ se debe probar que $\mathcal{S}_\sigma \models R$ si y solo si $\mathcal{S}_\sigma$ es una traza aceptada por B (es decir, $\mathcal{S}_\sigma \in \mathcal{L}(B)$), donde B es el autómata de Buchi construido por el tableau para la regla R.

A continuación se prueba la primer parte: Si $\mathcal{S}_\sigma \models R$, entonces $\mathcal{S}_\sigma$ (visto como una traza) $\in \mathcal{L}(B)$. Para lograr esto alcanza con probar los siguientes puntos:

- 1: $\mathcal{S}_\sigma$ es reconocible por B (siempre hay transiciones habilitadas),

- 2: $Visited_\infty(\mathcal{S}_\sigma) \subsetneqq Accepted(B)$ (los estados aceptadores se visitan de manera infinita)

Dado que $\mathcal{S}_\sigma \models R$ y empleando el lema 8.1.2 es fácil ver que el punto 1) se cumple, ya que cada prefijo *sp* de $\mathcal{S}_\sigma$ que lleva a un estado dado, por el lema 8.1.2 puede ser embebido en al menos una de las situaciones de ese estado. Esto indica que para cada estado habrá transiciones representando un avance, por lo que eventualmente $\mathcal{S}_\sigma$ será reconocida.

Para probar 2) se debe mostrar que para todo prefijo de una traza, o bien B está en un estado aceptador o eventualmente en el futuro se alcanzará un estado aceptador. Si ocurre el primer caso el punto 2) ya está probado ya que el estado ya es aceptador. En el segundo caso, dado que se conoce que $\mathcal{S}_\sigma \models R$, se tiene que el prefijo siempre podrá ser aumentado para así lograr completar la regla y llegar a un estado aceptador. Esto concluye la primera parte de la demostración.

Lo que queda demostrar entonces es la segunda parte. Es decir, probar que dada un escenario traza $\mathcal{S}_\sigma$, si $\mathcal{S}_\sigma$ (visto como una traza) $\in \mathcal{L}(B)$ entonces $\mathcal{S}_\sigma \models R$, donde R es una regla $f : A \to C$. Esto es, dado un morfismo m: $A \to \mathcal{S}_\sigma$, se debe probar que $\exists\ morfismo$ $m^{'} : C \to \mathcal{S}_\sigma \wedge m = m^{'} \circ f$. En otras palabras, debe existir un morfismo en el estado aceptador que pueda ser embebido en los morfismos de la regla.

Se define en primera instancia aquellos índices en $\mathcal{S}_\sigma$ tal que tienen menor precedencia que aquellos puntos definidos en el antecedente: $Index = \{i \in \mathcal{N} : \exists P_a \in A \wedge i = \#\{p \in S : p < m(P_a)\ \}\ \} + 1$. Se denomina k al máximo de la estructura *Index*. En este punto se conoce por 8.1.2 que $reach(\mathcal{S}_\sigma[1..k], B)$ incluye un estado conteniendo una situación η, la cual contiene un morfismo $m_i : A^{'} \to C_i$, $A^{'} = A, C_i \subsetneq C$ (notar que si el consecuente es matcheado, entonces el estado estará en un estado de trampa. Dado que una vez que se arriba a esa condición de trampa la condición del estado no varía, la traza no será aceptada, contradiciendo las asunciones iniciales). Por definición, este estado es un estado no aceptador. Dado que $\mathcal{S}_\sigma \in \mathcal{L}(B)$ $\exists\ r$ tal que $reach(\mathcal{S}_\sigma[1..k + r], B)$ incluye un estado aceptador. Luego, en el estado que incluye a η se puede realizar el siguiente análisis considerando las representaciones de m_i y sus extensiones en η. La secuencia extendida $\mathcal{S}_\sigma$[1..k+r] puede avanzar otros matchings posibles además de m_i. Si esto sucede, por cada nuevo elemento en la secuencia extendida se puede obtener $m_q : A^{'} \to C_i + \{e\}$, donde es el nuevo elemento matcheado, mientras que m_i se mantiene sin cambios. Sin embargo, todos estos otros posibles matchings serán eventualmente completados. Dado esto último, junto al hecho de que el conjunto de situaciones que incluyen al antecedente son finitas y limitadas, implican que en algún punto en el futuro el morfismo m_i será finalmente

completado (es decir, el consecuente será completado) y arribando así a un estado aceptador. También se conoce que en un estado aceptador el consecuente está completo, lo cual indica que el morfismo buscado $m^{'} : C \rightarrow \mathcal{S}_{\sigma}$ estará presente en una situación en el estado aceptador. Con esta afirmación queda demostrada la segunda parte y con ella la prueba de correctitud.

8.2. Esquema punta a punta del proceso de síntesis

En esta sección se provee un esquema general del proceso de síntesis del lenguaje. Este esquema se ilustra gráficamente en el figura 8.4, donde se muestran 3 diagramas integrados, mostrando distintos niveles del proceso de síntesis.

El diagrama 1 en la figura 8.4 muestra la semántica de FVS y ω-FVS tal cómo se detalló en las secciones 3.3 y 7.1: el comportamiento se especifica a través de reglas, y la semántica está dada por el conjunto de reglas que satisface las reglas. Adicionalmente, dadas las definiciones involucradas en la semántica de ω-FVS (ver sección 7.1), incluyendo nociones como morfismos, morfismo proyección y eventos fantasmas, se puede observar que el diagrama 1 conmuta.

El diagrama 2 de la figura 8.4 trae a escena el algoritmo de tableau, que traduce reglas en autómatas de Buchi. Tal como se vió en la sección 8.1, donde se analizó la completitud y corrección del tableau, este diagrama también es conmutativo.

Los diagramas 1 y 2 muestran como el autómata Buchi construido por el tableau para una regla ω-FVS produce trazas que vistas como escenarios y proyectadas en Σ_a resultan los escenarios trazas que satisfacen la regla . Todavía es necesario obtener un autómata que reconozca las trazas sobre Σ_a sin recurrir a la proyección de trazas. Sin embargo, esto se logra a partir del autómata $\mathrm{B}\!\downarrow_{\Sigma'}$, definido en la sección 2.2, teniendo que $\Sigma_a = \Sigma^{'}$. Como lo establece el lema 2.2.4 este autómata produce exactamente las trazas proyectadas sobre Σ_a. Luego, se puede establecer que también el diagrama 3 de la figura 8.4 conmuta. Notar que las trazas de autómatas y los escenarios trazas son en esencia el mismo objeto matemático debido a la existencia de mapeos inyectivos de trazas de autómatas a escenarios trazas como se vio en la sección 2.2.

Finalmente, como todos los diagramas describiendo el proceso de síntesis de la figura 8.4 conmutan se puede establecer que, dada una regla ω-FVS R el procedimiento de

tableau construye un autómata de Buchi tal que su proyección acepta cada escenario traza que satisface R. Este razonamiento puede extenderse al tratar con un conjunto de reglas. Dado un conjunto de reglas $S = \{r_1, r_2, \ldots, r_n\}$ se procede como se relata a continuación. Para S$\downarrow_{\Sigma^a}$ el autómata de Buchi que reconoce sus trazas es $\beta\downarrow_{\Sigma^a}$, donde $\beta \stackrel{\text{def}}{=} \parallel B_i$, tal que B_i, 1 $\leq$ i $\leq$ n es el autómata construido por el tableau para cada regla r_i y $\parallel$ representa el clásico operador de composición paralela.

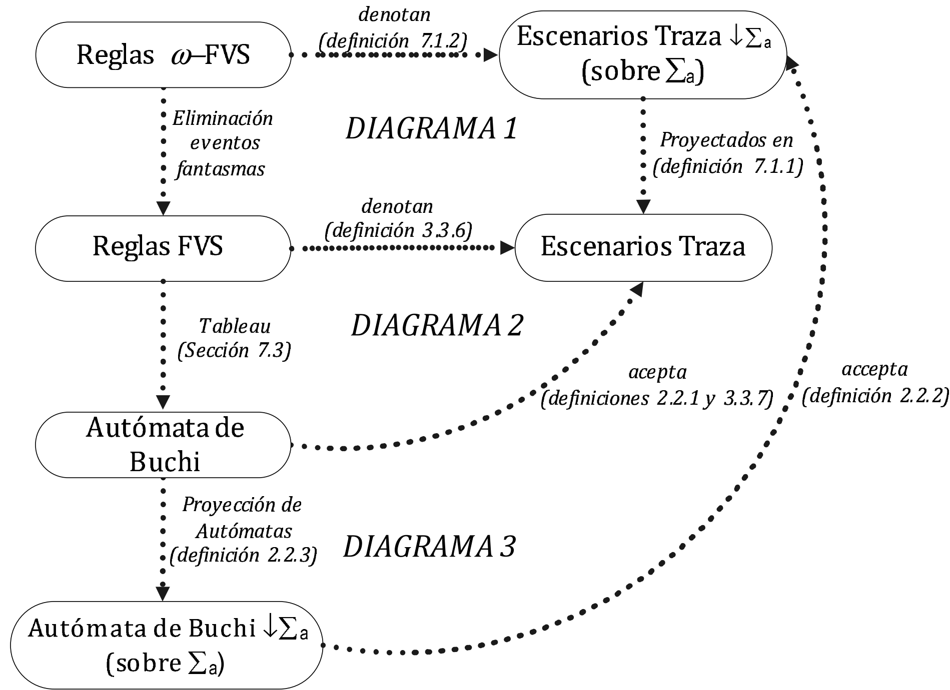

Figura 8.4: Esquema punta a punta del proceso de síntesis

Capítulo 9

Validación Construcción de Autómatas

El objetivo de esta sección es validar la generación de autómatas de Buchi generados por el tableau presentado en la sección 8.1. Para tal fin se comparan la complejidad (en términos de tamaño) de los autómatas generados contra otras autómatas propuestos en la literatura.

9.1. Caso de Estudio

Se realizó el siguiente caso de estudio para comparar los autómatas generados por el tableau con autómatas encontrados en la literatura aceptando el mismo lenguaje. En particular, se busca establecer si la traducción llevada a cabo por el algoritmo de tableau está en el mismo orden de complejidad que traducciones conocidas. Como medida de complejidad de un autómata se utiliza su tamaño, contando su cantidad de estados y transiciones, tal como fue empleada en [103]. En particular, se compara el tamaño de los autómatas generados con otros autómatas propuestos en la literatura. La traducción propuesta no es todavía un proceso optimizado, por lo que la comparación contra otras traducciones y autómatas conocidos es un punto de partida razonable.

En este sentido se decidió trabajar con un conjunto altamente representativo de propiedades LTL como lo son aquellas propuestas para los patrones de especificación en [43]. Anteriormente en este trabajo se demostró no sólo que FVS puede modelar todos los patrones de especificación sino que también que el modelado de dichos patrones es más su-

cinto y fácil de manipular y modificar que empleando otros formalismos (ver secciones 4.1 y 5.1). En esta sección se realiza un análisis similar comparando los autómatas generados por el tableau con las versiones en autómatas codificando patrones de especificación.

Los autómatas contra los cuales se comparan los generados por el tableau se obtuvieron de un extenso y completo repositorio de autómatas de Buchi, el cual se encuentra disponible de manera online (ver [1]), y fue presentado por los autores de la herramienta *GOAL* (Graphical Tool for Omega-Automata and Logic) [102]. Este repositorio incluye varios tipos de autómatas manejando lenguajes infinitos como Buchi no determinísticos, Buchi determinísticos, Co-Buchi determinísticos, Muller no determinísticos, por citar alguno de ellos. Es un repositorio libre, y cualquier usuario puede subir su propia autómata indicando qué formula lógica acepta. Para los própositos del presente caso de estudio se seleccionaron autómatas de Buchi no determinísticos propuestos por los autores de la herramienta.

Como se mencionó al principio de la sección, siguiendo lo propuesto en [103] se decidió contar el número de estado y transiciones como una manera posible de estimar la complejidad de un autómata. Esta medida resulta simple y también interesante ya que tiene correlato con los costos involucrados en los processo de verificación como model checking. En [103] esta medida también es utilizada como benchmark para comparar la herramienta *GOAL* contra otras herramientas y traducciones. Dado este contexto dicha medida es una buena aproximación para medir la complejidad de un autómata. Existen otros factores también importantes como por ejemplo el grado de no determinismo, los cuales se encuentran incluidos en el trabajo futuro que continua la presente tesis.

Para resumir, el caso de estudio compara en tamaño autómatas codificando patrones de especificación. Por un lado están los autómatas generados por el tableau de la sección 8.1 a partir de reglas FVS modelando patrones de especificación. Por otro lado, se obtuvieron las formulas LTL propuestas en [42] para cada patrón de especificación. Luego, se buscó en el repositorio online [1] los autómatas de Buchi que se correspondieran con eas fórmula.

9.1.1. Resultados

En esta sección se presentan y analizan los resultados obtenidos. La figura 9.1 muestra dos ejemplos incluyendo el patrón Respuesta con alcance global y el patrón Existencia con alcance *Antes*. Las fórmulas LTL para estos patrones en [42] son $\Box(P \rightarrow \Diamond Q)$ para el

patrón Respuesta y $\neg P\ U(Q \wedge \neg P)$ para el patrón Existencia. La columna de la derecha en la figura 9.1 muestra los autómatas de GOAL para estas formulas mientras que la columna de la izquierda presenta los autómatas generados por el tableau a partir de las reglas ω-FVS cubriendo los mencionados patrones. Aplicando la medida de complejidad (contar estados y transiciones) se puede observar que los autómatas de GOAL tienen dos estados y cuatro transiciones para el patrón Respuesta y dos estados y tres transiciones para el patrón Existencia. Por el otro lado, los autómatas generados por el tableau poseen seis estados y ocho transiciones para el patrón Respuesta y cinco estados y seis transiciones para el patrón Existencia.

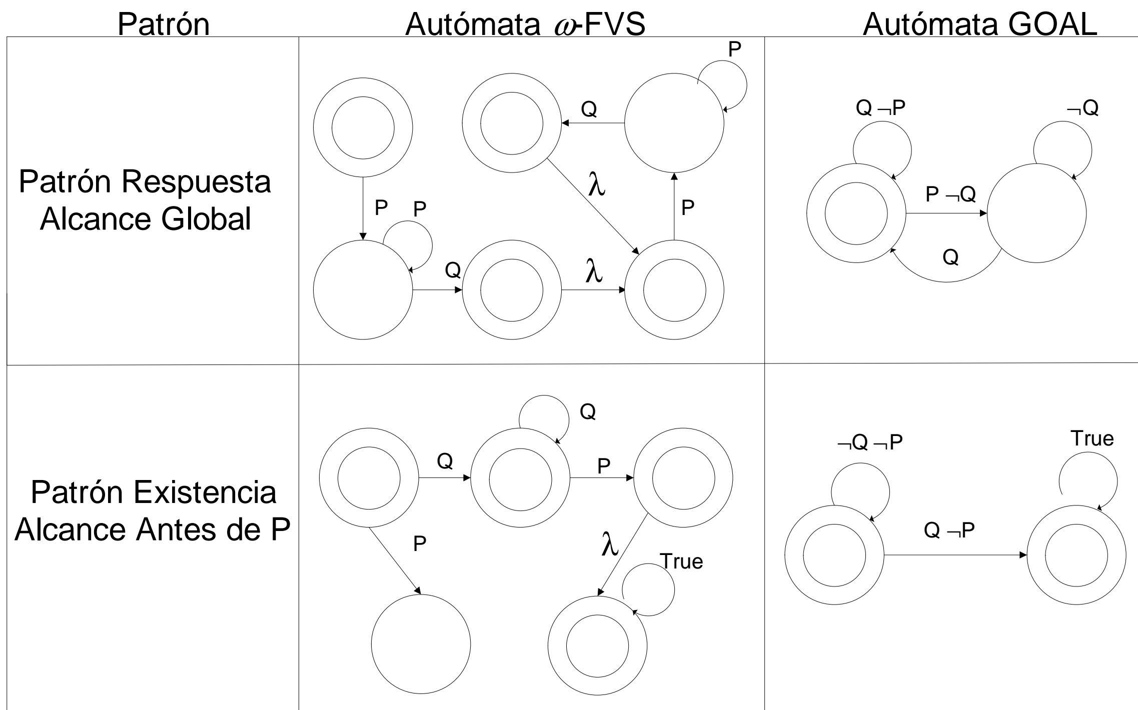

Figura 9.1: Comparación de tamaño entre autómatas

La tabla 9.1 resume la comparación del tamaño de autómatas para el resto de los patrones seleccionados, los cuales incluyen patrones como el patrón Respuesta, el patrón Existencia, el patrón Precedencia Encadenada, el patrón Respuesta Encadenada y el patrón Encadenamiento Restringido y el patrón Existencia limitada con k=2. También se incluyen variantes de algunos de estos patrones, cambiando el alcance o la cantidad de eventos que formal el estímulo y la respuesta en los patrones encadenados. El criterio de selección fue seleccionar todos los patrones de especificación descriptos en ([42]) que tuvieran una versión codificada en autómatas de Buchi en el repositorio de autómatas de Buchi ([1]).

Los patrones seleccionados representan un interesante subconjunto, cubriendo tantos los patrones de *ocurrencia* como los de *orden*. Las entradas de la tabla 9.1 siguen la notación *(número de estados, número de transiciones)*. La notación *(x,y)* acompañando los patrones encadenados representan la cantidad de eventos formando el estímulo, dada por el valor x, y la cantidad de eventos formando la respuesta, dada por el valor y.

Tabla 9.1: Comparación de tamaño de autómatas

Patrón	Alcance	GOAL-Buchi	ω-FVS Buchi
Existencia	Antes	(2,3)	(5,6)
Precedencia	Antes	(3,5)	(8,9)
Precedencia	Después	(5,8)	(6,10)
Respuesta	Global	(2,4)	(6,8)
Respuesta	Después	(3,7)	(5,7)
Precedencia Encadenada (1-2)	Global	(5,8)	(6,9)
Precedencia Encadenada (2-1)	Before	(6,12)	(11,15)
Respuesta Encadenada (1-2)	Global	(5,17)	(8,17)
Respuesta Encadenada (2-1)	Global	(6,15)	(9,14)
Encadenamiento Restringido (1-2)	Global	(6,15)	(11,15)
2-Existencia Limitada	Global	(5,9)	(7,10)

9.1.2. Análisis de los resultados

En los casos analizados los autómatas generados por el tableau de la sección 8.1 tienen mayor tamaño medido en cantidad de estados y transiciones que aquellos provistos por la herramienta *GOAL* en [1]. Sin embargo, la diferencia no llega a ser significativa para varios patrones. Asimismo, es posible reducir el tamaño y complejidad de los autómatas generados desde dos perspectivas diferentes. Por un lado es factible introducir cambios en el algoritmo de tableau para detectar estados y transiciones redundantes y así obtener autómatas de menor tamaño. Como un segundo frente, los autómatas generados pueden ser entrada para herramientas que minimizan autómatas como [102]. Ambos son objetivos del trabajo futuro de la presente investigación.

Capítulo 10

Trabajo Relacionado

El trabajo relacionado de la presente tesis se encuentra dividido en distintas categorías, de acuerdo a los distintos tópicos que se fueron desarrollando. Por un lado, se encuentran la comparación y análisis con otras notaciones gráficas basadas en escenarios, una de las características principales de FVS y ω-FVS. También se analizan otros enfoques centrados en la utilización de patrones de especificación. La tercer categoría de trabajo relacionado estudia en otras notaciones creadas con el objetivo de tener un poder expresivo más allá de LTL. Finalmente, se compara la visión de FVS como lenguaje de modelado orientado a aspectos con otras aproximaciones orientadas a aspectos.

10.1. Notaciones basadas en Escenarios

TimeEdit [97] y GIL (Graphical Interval Logic) [36] son dos lenguajes gráficos de especificación basados en diagramas de tiempo donde no los eventos no siguen un orden parcial. TimeEdit está principalmente enfocada en capturar cadenas complejas de eventos [18], mientras que FVS y ω-FVS apuntan a un esquemas más general para la especificación de propiedades. TimeEdit posee una noción de comportamiento que lleva a la ocurrencia de más comportamiento, pero con cierta limitaciones. Esto se logra a través del uso de lo que se denominan eventos *requeridos*, que son eventos que deben ocurrir si un cierto grupo de eventos ha ocurrido en el pasado). Las limitaciones en la notación está dada por una estructura rígida, donde no es simple de especificar comportamiento que ocurrió en el pasado, o eventos que ocurren dentro de un alcance acotado. Este tipo de restricciones hace que las propiedades sean más difíciles de especificar y entender. GIL provee operadores

de búsqueda para poder especificar intervalos, conceptos similares a los operadores *next* y *previous* en ω-FVS y FVS. Sin embargo, en GIL el operador *previous* no puede aplicarse de manera tan libre como en ω-FVS o FVS: el reconocimiento de intervalos empieza siempre hacia adelante de un cierto punto (o desde ese mismo punto). Luego, situaciones fácilmente expresables en ω-FVS y FVS como restricciones de correlato o aserciones sobre comportamiento pasado no puede ser expresado en GIL. Finalmente, la especificación en GIL de expresiones complejas con varios eventos y relaciones de ocurrencia lleva a anidar o apilar operadores, obteniéndose especificaciones difíciles de comprender, modificar, manipular y validar.

PSC (Property Sequence Chart) [18] es también una notación relevante, inspirada en los diagramas de Secuencia e Interacción de UML 2.0. La usabilidad de PSC también se valida utilizando patrones de especificación. Como establecen los autores de este enfoque, puede llegar a ser difícil expresar algunas propiedades directamente en el lenguaje, por lo que se requiere de herramientas automáticas que proveen asistencia [19]. El manejo de restricciones complejas se lleva a cabo empleando anotaciones en lenguaje natural. Otra diferencia con FVS y ω-FVS es que las propiedades se describen no por el comportamiento en sí que exhiben, sino por el razonamiento complementario, pensando en su negación, como si fueran anti-escenarios.

Otros formalismos visuales basados en *Message Sequence Charts* como [94, 104, 50] han sido propuestos para la especificación de propiedades basada en escenarios. Tanto FVS como ω-FVS comparten con estos enfoques la idea de utilizar órdenes parciales de eventos para describir escenarios. Sin embargo, existen diferencias desde varios ángulos que es importante notar. En primer término, en FVS y ω-FVS el comportamiento expresado en el antecedente no tiene porqué necesariamente preceder en el tiempo al comportamiento expresado en el consecuente. Adicionalmente, en FVS y ω-FVS los consecuentes pueden expresar comportamiento sobre eventos que ocurren antes del antecedente, o incluso comportamiento que ocurre de entre el antecedente y el consecuente.

Finalmente, hasta donde se pudo tener conocimiento, ninguna de las aproximaciones nombradas anteriormente está equipada con mecanismos que permitan llevar a cabos tareas claves de validación como comparar especificaciones, o razonar por comportamiento complementario. Similarmente, en cuanto al poder expresivo, ninguno es capaz de modelar propiedades ω-regulares.

10.2. Aproximaciones basadas en Patrones de Especificación

Aproximaciones como [86, 87] proponen un lenguaje de especificación basado en lenguaje natural (en inglés) con restricciones, siguiendo una dada gramática que le da forma a la notación, la cual está inspirada en un sistema de especificación de patrones definido en [64]. Las propiedades son luego traducidas a un formalismo basado en LTL. A diferencia de FVS y ω-FVS este lenguaje no provee mecanismos intuitivos para llevar a cavo tareas de validación. Otra diferencia importante es que en FVS y ω-FVS las propiedades son denotadas empleando una notación gráfica en vez de una gramática sobre lenguaje natural, obteniendo así un enfoque más comprensible y flexible.

La idea de definir eventos en término de ocurrencia de otros eventos también ha sido explotada en diferentes áreas. Mayormente empleado en el campo de la inteligencia artificial el formalismo denominado *Situation Calculus* [77, 106] también permite razonar sobre acciones disponibles y describir el comportamiento que tendrán estas acciones en el mundo que está siendo modelado. Otro enfoque interesante está dado por Complex Event Processing (CEP) [73], donde el objetivo principal es detectar y reaccionar ante patrones de comportamiento lo antes posible. Estas ideas se han aplicado principalmente para bases de datos y otras aplicaciones basadas en el intercambio intensivo de datos. Más cercano a los objetivos de FVS y ω-FV, en [47] se utiliza el concepto de *fluent* para relacionar e interconectar la ocurrencia de eventos. Un *fluent* denota un comportamiento complejo, donde se debe expresar las acciones que lo inician y finalizan. Esta estructura de modelado de comportamiento es más rígida que la que presentan FVS y ω-FVS.

El lenguaje PSL [54, 45] es ampliamente utilizado en la comunidad orientada al hardware para el diseño y verificación de chips electrónicos. Propiedades de hardware pueden ser especificadas en PSL para así luego verificar el comportamiento esperado. Este lenguaje nació primero bajo la denominación *Sugar* para luego evolucionar al nombre actual y convertirse en un estándar IEEE. Está enfocado en LTL, con la inclusión de expresiones regulares para poder expresar más tipos de propiedades. Si bien está orientado a la verificación en hardware comparte con FVS algunos objetivos. En el mismo espíritu que FVS reconocen también la necesidad de validar la propiedad y controlar que esté expresando el comportamiento correcto. Sin embargo, esta validación en PSL se logra a través de he-

rramientas construidas por sobre PSL como se observa en [35, 24]. Luego, por el contrario que en FVS y ω-FVS, los mecanismos de validación requieren herramientas por fuera del lenguaje.

10.3. Más allá del poder expresivo de LTL

Varios autores han señalado la necesidad de contar con lenguajes de especificación más expresivos. El trabajo en [23] propone introducir operadores de *agregación* (como ser promedio, mínimo, máximo, etc.) a los lenguajes de especifcación. También [25] extiende formalismos de especificación con aspectos cuantitativos. En la misma línea, [8] y [82] extienden lógicas de primer orden y lógicas modales para contemplar operadores de *agregación*.

Más relacionados con FVS y ω-FVS, varias propuestas han surgido como alternativas para superar el limitado poder expresivo de las lógicas temporales, e incluso algunas de ellas logran el mismo poder expresivo que los autómatas de Buchi [23, 114, 108, 68, 115, 27]. Entre ellas, lógicas como [114, 108, 27] incorporan características operacionales como expresar comportamiento mediante autómatas para aumentar el poder expresivo. Sin embargo, en ocasiones el resultado de aumentar la expresividad termina resultando una seria amenaza a la usabilidad del lenguaje, lo cual termina reduciendo muchas aproximaciones a tener impacto únicamente desde aspectos teóricos [92, 115].

10.4. Orientación a aspectos

Visto como una notación orientada a aspectos FVS se encuadra en la categoría de aproximaciones "simétricas". En esta categoría los aspectos son tratados como cualquier otra funcionalidad. En cambio, en las aproximaciones "asimétricas" existe un sistema base, usualmente representando requerimientos funcionales, al cual se agregan los aspectos, modelando requerimientos no funcionales. Trabajos como [79, 30, 74, 2, 63] se encuadran dentro de las notaciones asimétricas, utilizando notaciones basadas en UML, como diagramas de secuencia, diagrama de actividades, etc. Otros trabajos como [7] también toman una postura simétrica, pero la aplicación de aspectos se resuelve de manera operacional a través de un mecanismos de composición basado en la sintaxis del lenguaje.

Como se mencionó, FVS propone una visión completamente diferente, al proponer una especificación declarativa del comportamiento, cercana a la manera en que los requerimientos son expresados [107]. Mientras FVS se enfoca en eventos, la mayoría de las demás aproximaciones se basan en la noción de estados o interacciones. En este contexto, existen alternativas asimétricas que también proponen la idea de mapear comportamiento sobre trazas [90, 39, 4], las cuales priorizan fundamentalmente el modelado de protocolos de comunicación. A diferencia de FVS, el uso de patrones de eventos (por ejemplo, gramáticas libres del contexto) está limitado para expresar pointcuts (en FVS el mismo mecanismo se aplica para modelar advices). Adicionalmente, la noción de eventos en FVS es abstracta, ya que se busca definir por completo el sistema en términos de reglas, sacrificando así operacionalidad en el lenguaje para ponderar su capacidad declarativa.

En cuanto a la interferencia de aspectos, varias aproximaciones han intentado atacar este problema desde diferentes puntos de vista y concentrándose en diferentes fases del desarrollo de software [41, 99, 111, 78, 93]. Los trabajos en [78] y [99] proponen herramientas para detectar conflictos entre aspectos. Ambas herramientas se basan en transformaciones sintácticas sobre grafos. En cambio FVS propone una solución enfocada en la descripción declarativa del comportamiento. Finalmente, [93] introduce una interesante herramienta denominada MEDIATOR para el manejo de conflictos entre aspectos, la cual también incluye nociones de comportamiento que dispara la ocurrencia de otro comportamiento. A diferencia de FVS, enfocado en el modelado temprano, este framework está orientado a la fase de implementación

Es importante mencionar que, hasta donde fue posible averiguar, ninguno de los trabajdo mencionados tiene mecanismos para razonar sobre el comportamiento complementario.

Capítulo 11

Trabajo Futuro

El trabajo futuro de la presente investigación plantea extender ω-FVS para cubrir de una manera más completa el amplio aspecto de la especificación de comportamiento y la construcción de sistemas a partir de la misma. Esta extensión puede pensarse desde dos puntos de vista, contemplando otras visiones para especificar comportamiento: los sistemas abiertos y especificaciones parciales basadas en estructuras "branching time". Las siguientes secciones cubren estos aspectos, para luego finalmente mostrar cómo pueden ser combinadas.

11.1. Sistemas Abiertos

ω-FVS está enfocado en sistemas cerrados, dejando de lado una concepción del software como *sistemas abiertos*. En muchos contextos la especificación del software es un conjunto de objetivos de un sistema abierto (que controlaría y monitorearía eventos) y que debe cumplirlos en el contexto de un ambiente que cumple ciertos supuestos [69, 21, 67]. Esto se contrapone con la visión de la especificación como la descripción directa del software a construir, que en este contexto de sistema abierto debería ser sintetizada [37].

Sistemas abiertos implica por un lado el diseño y manejo de algoritmos de control en Ingeniería de Software, y por otro, la posibilidad de contemplar el comportamiento esperado del ambiente, visión común dentro de la Ingeniería de Requerimientos [107, 55]. Para mencionar un pequeño ejemplo de interacción con un ambiente, basta considerar el ejemplo del sistema de las luces interiores de un auto visto en la sección 7.1. En el mismo, se busca evitar que la batería se descargue cuando funciona en un modo "caro",

que ocurre cuando las luces interiores se encienden y el auto está apagado. Para cumplir con este requerimiento la solución recae en pedir que el auto se encienda (y así la batería se recargue) entre dos funcionamientos consecutivos de la batería en modo "caro". Sin embargo, que el auto se encienda o no está fuera del control del sistema de luces, ya que forma parte de un comportamiento controlado por el ambiente o entorno. Este es sólo un pequeño ejemplo de la necesidad de extender FVS para especificar y sintetizar comportamiento en sistemas abiertos.

La síntesis de comportamiento ha sido atacada metodológicamente y algorítmicamente desde la síntesis y generación automática de controladores [83, 56, 33, 38]. Se proveen mecanismos para diferenciar y distinguir el comportamiento que un componente puede hacer y controlar, y aquellas cosas que están fuera de su alcance: el comportamiento que proviene del entorno o ambiente que interactúa con el sistema. Sin embargo, existen limitaciones desde el punto de vista del lenguaje de especificación. Estas aproximaciones están fuertemente basadas en lógicas temporales (LTL) y algunas extensiones como fluents [47]. La expresividad de estos lenguajes ya ha sido desafiada por la comunidad, por lo que existe la necesidad de basarse en lenguajes de especificación con mayor poder expresivo. Asimismo, en estas técnicas no es posible determinar el controlador más general cuando se evalúan propiedades de tipo "liveness", lo cual constituye una limitación conceptual de estas herramientas.

11.2. Especificaciones parciales en estructuras "branching time"

Existen otras aproximaciones basadas en especificaciones parciales que también han atacado el problema de expresividad en los lenguajes de especificación [71, 96]. Estas aproximaciones se alejan de las estructuras lineales para enfocarse en estructuras "branching time". Las mismas buscaron extender los modelos de comportamiento introduciendo dos tipos de transiciones: requeridas y posibles. Luego, un modelo parcial describe posibles conjuntos de implementaciones y donde existe la posibilidad de manipular este universo de especificaciones operacionales. Una característica de esta generación de especificaciones parciales es que posibilitan la descripción, manipulación y cómputo de posibles implementaciones que satisfacen los requerimientos planteados. En este contexto, es posible también

el uso de MTS (Modal Transition Systems) para tomar decisiones sobre la especificación operacional que más se ajuste a los requerimientos, particularmente en aquellas situaciones donde existe más de una implementación posible y se debe guiar al usuario para llegar a una única solución. Si bien el aporte de las estructuras “branching time” es importante a la hora de especificar comportamiento, existen limitaciones a la hora de trasladarlas a sistemas abiertos, donde se describen propiedades a ser satisfechas por un ambiente o entorno. En estas estructuras se describe el comportamiento de un componente, pero sin hacer distinción entre lo que el componente puede hacer y lo que puede controlar.

11.3. Combinación de los enfoques

Por sí solos, los tres enfoques sufren de distintas debilidades. Por ejemplo, ω-FVS es declarativo y expresivo, pero se basa en la derivación de trazas y la especificación de sistemas cerrados, sin tener en cuenta la interacción con un ambiente. Las propuestas sobre sistemas abiertos [38, 69] sufren problemas de expresividad, además de otras dificultades conceptuales que impondrían un bias accidental para sintetizar especificaciones operacionales. En concreto no es posible obtener el controlador más general para propiedades que no sean de tipo “safety”. En el caso de aproximaciones basadas en especificaciones parciales como [67, 95] logran manejar adecuadamente nociones de parcialidad basadas en estructuras “branching time”, pero nuevamente, cuentan con poco poder expresivo y tampoco tienen en cuenta la interacción con un ambiente.

En este contexto el trabajo futuro consiste en generar una sinergia entre tres líneas de investigación: descripción declarativa de propiedades, especificaciones parciales y síntesis de comportamiento para sistemas abiertos a partir de objetivos. Más concretamente, el objetivo principal para el trabajo futuro de esta investigación es el desarrollo de un nuevo lenguaje declarativo con el suficiente poder expresivo para especificar el comportamiento de sistemas abiertos, y la capacidad para operacionalizar las especificaciones. En el mismo se combinará la posibilidad de describir comportamiento parcial, permitiendo el modelado incremental, junto con la posibilidad de especificar el comportamiento de artefactos describiendo su interacción con un ambiente o contexto externo, habilitando las especificaciones de sistemas abiertos. Estas son las premisas para extender ω-FVS, manteniendo su naturaleza declarativa y flexibilidad a la hora de especificar propiedades.

11.3.1. Metodología

Se buscará extender ω-FVS para incluir el modelado de sistemas abiertos, buscando combinar y potenciar las áreas de síntesis de controladores y especificaciones parciales basadas en estructuras “branching time”. Se proveerán nuevas técnicas y algoritmos de síntesis de comportamiento con foco en lenguajes de especificación más expresivos de manera de poder generar y sintetizar comportamiento a partir de propiedades. Los desafíos de esta extensión incluyen lo posibilidad de definir el controlador más general tanto para propiedades de tipo “safety” como también de tipo “liveness”.

Para atacar la síntesis de comportamiento, se propone extender ω-FVS para distinguir eventos controlables y no controlables, adaptando su sintaxis y semántica, así como también el algoritmo de síntesis que traduce escenarios FVS en autómatas de Buchi, prestando especial atención a la complejidad algorítmica involucrada. Para incluir especificaciones parciales se explorarán lógicas de tipo branch, como CSSL (Conditional Scenario Specification Language) propuesta en [36], atendiendo la necesidad de contemplar especificaciones parciales. Finalmente, se estudiarán alternativas para sumar y combinar especificaciones operacionales parciales a los escenarios usuales de ω-FVS.

La extensión a ω-FVS incluye también una herramienta gráfica y su integración con otras herramientas como LTSA [66] y GOAL [102]. Como principio de diseño del lenguaje se buscará contemplar distintas fuertes de especificaciones: basadas en lógicas, en autómatas, en lenguajes naturales restringidos, etc. Entre los casos de estudio planeados a realizar se buscará priorizar protocolos de comunicación en sistemas reales. Casos de estudio en estos dominios son de especial interés, ya que los requerimientos son complejos y existen interacciones desafiantes desde un punto de vista del modelado de comportamiento.

Capítulo 12

Conclusiones

La especificación formal de propiedades es una actividad crucial para la verificación formal de software. Luego, es fundamental contar con un lenguaje declarativo formal para llevar adelante este proceso que cumpla ciertas características deseables. En particular, se busca contar con especificaciones sucintas, comparables, modificables, y bajo las cuales sea posible razonar por razonamiento complementario. Además, el poder expresivo del formalismo debe ser suficiente como para poder expresar todas las propiedades de interés.

Teniendo esto en cuenta en la presente tesis se presentó el lenguaje declarativo FVS, y su posterior versión ω-FVS para facilitar el proceso de especificación de propiedades. FVS es un lenguaje gráfico basado en escenarios, con una sintaxis y semántica clara y bien definida. El comportamiento se modela a través de reglas, y la semántica está dada por el conjunto de reglas que las satisfacen. La usabilidad de FVS fue validada modelando todos los patrones de especificación. Además, se compararon sus especificaciones contra otras notaciones formales como lógicas temporales, autómatas y lenguaje natural disciplinado. La comparación se llevó a cabo teniendo en cuenta los atributos de calidad anteriormente mencionados: *sucinto*, *comparabilidad*, *complemento* y *modificabilidad*. Las especificaciones de FVS para los patrones de especificación resultaron ser más concisas, y simples de comparar y modificar, además de proveer mecanismos para razonar por comportamiento complementario. Como conclusión se puede afirmar que FVS maneja adecuadamente tareas de validación cuando el usuario está explorando los requerimientos del sistemas y tratando de determinar su comportamiento con precisión.

También se exploró la versatibilidad de FVS proponiéndolo como un lenguaje de modelado orientado a aspectos. La flexibilidad de la notación en FVS permite obtener un

modelo de pointcuts y advices muy expresivo, donde el comportamiento de los aspectos puede ser modelado con naturalidad. Las características de FVS resultaron más que adecuadas para el modelado de aspectos, e incluso atacan problemas claves de la orientación a aspectos como la falta de semántica clara para el modelado incremental, y la interferencia de aspectos.

Finalmente, se presentó una extensión a FVS para poder describir el comportamiento bajo propiedades ω-regulares, obteniéndose así el lenguaje ω-FVS. La presencia de eventos fantasmas en ω-FVS permite dar el salto expresivo y poder introducir nociones de abstracción en etapas tempranas de modelado. El poder expresivo de ω-FVS se vio en acción modelando distintos protocolos de comunicación de software. Se presentó también en esta etapa un algoritmo de tableau, que traduces las reglas ω-FVS un autómatas de Buchi, posibilitando así la capacidad de realizar análisis automático (por ejemplo, validar la consistencia de un conjunto de reglas, o verificar interferencia entre reglas) y facilitar la integración con otras notaciones. Se comparó el tamaño de los autómatas generados contra autómatas propuestos en la literatura aceptando el mismo lenguaje.

Por todo lo expuesto, se puede afirmar que FVS y ω-FVS representan una alternativa sólida y atractiva para la especificación declarativa de sistemas.

Apéndice A

Demostración Lema Caracterización Estados del Tableau

En este apéndice se demostrará el lema 8.1.2 detallado en la sección 8.1.3. Como se vió anteriormente este lema es fundamental para la demostración del algoritmo de tableau presentado en el capítulo 8. Más precisamente, este lema relaciona las trazas del sistema con los estados del autómata que construye el algoritmo de tableau. En pocas palabras, el lema dice que toda traza que lleva a un estado dado S, puede ser embebida, al menos, en una de las situaciones de S de manera maximal, es decir, incluyendo todos los puntos involucrados.

Lema (Caracterización Estados del Tableau)**.** *Dada una regla* R*:* $f : A \to C$*, morfismos* $e : A' \to A$*,* $e^i : C^i \to C$*,* $i \in [1..k]$*, un estado* S *en el autómata construido por el tableau, y una traza* t *que lleva al estado* S*, entonces* $\exists$ *una situación* η *,* $\eta \in S$*, incluyendo morfismos* $g_i : A' \to C^i$*,* $i \in [1..k]$ *y* $\exists h : A' \to t$*,* $h_i : C_i \to t (i \in [1..k])$ *tal que se cumplen las siguiente condiciones:*

- *Condición 1:* $h_i \circ g_i = h$
- *Condición 2:* $\forall w : A' \to C', e' : C' \to C$*, donde* w*,*e*,*e' *y* f *satisfacen las condiciones 1, 2 y 3 de la definición 8.1.1, si* $\exists h' : C' \to t$ *tal que* $h' \circ w = h$ *entonces* $\exists i$ $(h' = g_i)$*.*

Demostración. Notar que el lema es equivalente a $\forall$ traza t y $\forall$ estado S tal que t lleva a S, las condiciones 1 y 2 del lema se satisfacen. Esto se debe a que todos los estados del

autómata son alcanzables. De esta manera, el lema se probará usando una prueba por inducción sobre la longitud de una traza t.

Para el caso base se parte de una traza de longitud 0. Se debe probar que $\exists$ una situación η , $\eta \in S$, incluyendo morfismos $g_i : A' \rightarrow C^i$, $i \in [1..k]$ y $\exists h : A' \rightarrow t$, $h_i : C_i \rightarrow t(i \in [1..k])$ tal que se cumplen las condiciones del lema. A continuación se demuestra que el estado inicial del autómata, S^0 es el estado S buscado. El estado inicial S^0 contiene, por construcción, una situación incluyendo el morfimo vacío (esto es, un morfismo: $\emptyset \rightarrow \emptyset$). Esta situación es la situación η buscada. De la misma forma, el morfismo vacío consituye los morfismos h y h_i buscados, ya que la traza tiene longitud 0. Luego, se puede ver trivialmente que $h_i \circ g_i = h$, satisfaciendo así la condición 1 del lema. En sencillo ver que la condición 2, que establece la maximalidad, también se satisface, ya que el único morfismo valido para una traza t de longitud 0 es el morfismo vacío, el cual se encuentra incluido en el estado inicial S^0.

Ahora se probará el lema para una traza de longitud *n+1*, asumiendo que las condiciones se satisfacen para una traza de longitud n, lo cual constituye la Hipótesis Inductiva (HI). Se expresará como t_{n+1} una traza arbitraria de longitud n+1, y como t_n la traza t_{n+1} sin su último elemento (referido como el minterm m).

Para probar el lema para t_{n+1} se necesita ver que $\forall$ estado S', tal que t_{n+1} lleva a S', $\exists$ una situación η', incluyendo morfismos g_i', tal que se cumplen las dos condiciones del lema. Dada la equivalencia entre trazas y escenario trazas (ver definición 3.3.7) se puede ver a una traza t_{n+1} como una sucesión de minterms. Sabemos por HI que para t_n $\exists$ una situación η , $\eta \in S$, incluyendo morfismos: $g_i : A' \rightarrow C^i$, $i \in [1..k]$, y $\exists h : A' \rightarrow t_n$, $h_i : C_i \rightarrow t_n$ tal que las condiciones del lema se cumplen. La situación candidata η' que se propone es la situación que el algoritmo de tableau construye a partir de η, al calcular los avances que surgen del procesamiento del minterm m. Al calcular todos los sucesores del estado S en las líneas 5 y 6 del algoritmo 1 todas las situaciones en ese estado serán avanzadas considerando todas las combinaciones. En particular, se avanzará la situación η. Cuando se avanza una situación, se avanzan todos sus morfismos tal como se ve en las línea 2 de los algortimos 2 y 3, nuevamente considerando todas las combinaciones según el minterm m, tanto en el antecedente como en el consecuente. Es decir, los morfismos $g_i : A' \rightarrow C^i$ serán avanzados considerando el minterm m. Los algoritmos 2 y 3 computan los avances calculando avances en una configuración, tal como se definió en la sección 8.1.1. Sea A''

y $C^{''i}$ las configuraciones que se obtienen mediante $A^{'} \xrightarrow{m} A^{''}$ y $C^{i} \xrightarrow{m} C^{''i}$ reespectivamente. Realizado este paso, los morfismos $g_i^{'} \in \eta^{'}$ son iguales a los morfismos $g_i \in \eta$ para aquellos elementos $\in A^{'}$, y para aquellos nuevos elementos del antecedente (aquellos en $A^{''}$-$A^{'}$) los morfismos $g_i^{'}$ los relacionan con los nuevos elementos del consecuente (aquellos en $C^{''i} - C^{i}$).

Dado un estado $S^{'}$ conteniendo una situación $\eta^{'}$ construida como se explicó recien, ahora se contruyen los morfismos candidatos $h^{'}$ y $h_i^{'}$ a partir de los morfismos h y h_i. $h^{'}(x) = h(x), \forall x \in A^{'}$ y $h^{'}(x) = n+1, \forall x \in A^{''} - A^{'}$. Notar que n+1 es la última posición de la traza t_{n+1}, la cual está etiquetada con el minterm m. De manera similar $h_i^{'}(x) = h_i(x), \forall x \in C^{i}$ y $h^{'}(x) = n+1, \forall x \in C^{''i} - C^{i}$.

A continuación se prueba que los morfismos candidatos $g_i^{'}$, $h^{'}$ y $h_i^{'}$ satisfacen las dos condiciones del lema. Para ver que se cumple la condición 1 se debe probar que $h_i^{'} \circ g_i^{'} = h^{'}$. Por definición, todos los candidatos $g_i^{'}$, $h^{'}$ $h_i^{'}$ se comportan exactamente igual a g_i, h y h_i para aquellos elementos en $A^{'}$. También se conoce por HI que $h_i \circ g_i = h$. Luego, sólo resta probar que $h_i^{'} \circ g_i^{'} = h^{'}$ se cumple para los nuevos elementos agregados por el procesamiento del minterm m. Esto es, para aquellos puntos en $A^{''}$-$A^{'}$. Por definición, $h^{'}$ relaciona estos puntos con n+1. Siguiendo la definición de los morfismos $g_i^{'}$ se sabe que los nuevos elementos del antecedente son relacionados con los nuevos elementos del consecuente. Pero también se conoce que, por definición, $h^{'}$ relaciona estos puntos (los nuevos elementos del consecuente) con n+1. Luego, $h_i^{'} \circ g_i^{'} = h^{'}$ y la condición 1 del lema está demostrada.

Para probar la condición 2 se procede de la siguiente manera. Se asume por el absurdo que la condición de maximalidad no se cumple para t_{n+1}. Luego, existe una situación $\eta^{'}$ tal que $\exists$ un estado $S^{'}$, t_{n+1} lleva a $S^{'}$, y $\eta^{'} \notin S^{'}$. Se sabe que la imagen de $\eta^{'}$ incluye el minterm m, el último minterm de t_{n+1}. De otra manera, $\eta^{'}$ hubiera sido una situación para t_n, lo cual es un absurdo ya que sabemos que la condición de maximalidad se cumple para t_n. Considerar ahora la situación η que surge de eliminar n+1 de la situación $\eta^{'}$. Es decir, remueve todo los pares que están relacionados a n+1. Es fácil ver que η es una situación para t_n. Luego, se sabe por HI que $\forall$ estado S, donde t_n lleva a S, $\eta \in S$. Sin embargo, dada la naturaleza exhaustiva del algoritmo de tableau (avanza todas las situaciones en cada estado), $\eta^{'}$ estaría incluida en uno de los sucesores de S al realizar el procesamiento del minterm m. Esto es un absurdo, ya que contradice las asunciones iniciales que decían

que $\eta^{'}$ no estaba incluida en ningún estado $S^{'}$, tal que t_{n+1} lleva a $S^{'}$. Como este absurdo provino de suponer que la condición de maximalidad no se cumple para t_{n+1}, se puede afirmar la condición de maximalidad se cumple para t_{n+1}. Como ambas condiciones están demostradas, concluye la demostración del lema.

$\square$

Bibliografía

[1] Buchi online Store. In *http://buchi.im.ntu.edu.tw/index.php/browse/index/*.

[2] O. Aldawud, A. Bader, T. Elrad, et al. Weaving with statecharts. 2002.

[3] A. Alfonso, V. Braberman, N. Kicillof, and A. Olivero. Visual timed event scenarios. In *Proceedings of the 26th International Conference on Software Engineering*, pages 168–177. IEEE Computer Society, 2004.

[4] C. Allan, P. Avgustinov, A. S. Christensen, L. Hendren, S. Kuzins, O. Lhoták, O. De Moor, D. Sereni, G. Sittampalam, and J. Tibble. Adding trace matching with free variables to aspectj. *ACM SIGPLAN Notices*, 40(10):345–364, 2005.

[5] R. Altman, A. Cyment, and N. Kicillof. On the need for SetPoints. In *First European Interactive Workshop on Aspects in Software (EIWAS)*, 2004.

[6] J. Araújo, A. Moreira, I. Brito, and A. Rashid. Aspect-oriented requirements with uml. In *Workshop on Aspect-oriented Modeling with UML*, volume 7, 2002.

[7] J. Araujo, J. Whittle, and D.-K. Kim. Modeling and composing scenario-based requirements with aspects. In *Requirements Engineering Conference, 2004. Proceedings. 12th IEEE International*, pages 58–67. IEEE, 2004.

[8] C. Areces, G. Hoffmann, and A. Denis. Modal logics with counting. In *Logic, Language, Information and Computation*, pages 98–109. Springer, 2010.

[9] F. Asteasuain. Un enfoque declarativo para el modelado de requerimientos. In *Encuentro de Doctorandos CACIC-2011*, 2011.

[10] F. Asteasuain and V. Braberman. FVS: A declarative aspect oriented modeling language. *EJS Electronic Journal SADIO (1514-6774)-Volume 10,Number 1, pages 20-37, 2011.*

[11] F. Asteasuain and V. Braberman. Specification patterns: formal and easy. In *2nd Revision Submitted Feb 2013 to: International Journal of Software Engineering and Knowledge Engineering.*

[12] F. Asteasuain and V. Braberman. Exploring visual scenarios as an aspect oriented modeling language. In *Proceedings of ASSE 2010 (Argentinian Simposium of Software Engineering)*, 2010.

[13] F. Asteasuain and V. Braberman. Specification patterns can be formal and also easy. In *Software Engineering and Knowledge Engineering (SEKE)*, pages 430–436, 2010.

[14] F. Asteasuain and V. Braberman. Visual scenarios for addressing the aspect interference problem. In *Proceedings of ASSE 2011 (Argentinian Simposium of Software Engineering)*, 2011.

[15] F. Asteasuain and V. Braberman. Declaratively building behavior by mean of scenario clauses. In *Submitted to: Requirement Engineering Journal ISSN: 0947-3602*, 2013.

[16] F. Asteasuain and V. Braberman. Declaratively building behavior by means of scenario clauses. Technical Report 05-13, Computer Science Department – School of Science - UBA, 2013.

[17] F. Asteasuain and V. Braberman. On the need for a declarative and incremental behavioral modeling. In *Proceedings of 3st LA WASP (Workshop on Aspect-Oriented Software Development*, Brasil.2009.

[18] M. Autili, P. Inverardi, and P. Pelliccione. Graphical scenarios for specifying temporal properties: an automated approach. *ASE*, 14(3):293–340, 2007.

[19] M. Autili and P. Pelliccione. Towards a graphical tool for refining user to system requirements. *Electronic Notes in Theoretical Computer Science*, 211:147–157, 2008.

[20] E. Baniassad and S. Clarke. Theme: An approach for aspect-oriented analysis and design. In *Proceedings of the 26th International Conference on Software Engineering*, pages 158–167. IEEE Computer Society, 2004.

[21] S. Ben-David, M. Chechik, A. Gurfinkel, and S. Uchitel. Cssl: a logic for specifying conditional scenarios. In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*, pages 37–47. ACM, 2011.

[22] L. Bergmans. Towards detection of semantic conflicts between crosscutting concerns. *Analysis of Aspect-Oriented Software - European Conference on Object-Oriented Programming (ECOOP)*, 2003.

[23] D. Bianculli, C. Ghezzi, C. Pautasso, and P. Senti. Specification patterns from research to industry: a case study in service-based applications. In *Proceedings of the 2012 International Conference on Software Engineering*, pages 968–976. IEEE Press, 2012.

[24] R. Bloem, R. Cavada, C. Eisner, I. Pill, M. Roveri, and S. Semprini. Manual for property simulation and assurance tool (deliverable 1.2/4-5). Technical report, PROSYD Project, Tech. Rep, 2004.

[25] U. Boker, K. Chatterjee, T. A. Henzinger, and O. Kupferman. Temporal specifications with accumulative values. In *Logic in Computer Science (LICS), 2011 26th Annual IEEE Symposium on*, pages 43–52. IEEE, 2011.

[26] D. Bosscher, I. Polak, and F. Vaandrager. Verification of an audio control protocol. In *Formal Techniques in Real-Time and Fault-Tolerant Systems*, pages 170–192. Springer, 1994.

[27] A. Bouajjani, Y. Lakhnech, and S. Yovine. Model checking for extended timed temporal logics. In *Formal Techniques in Real-Time and Fault-Tolerant Systems*, pages 306–326. Springer, 1996.

[28] V. Braberman, D. Garbervestky, N. Kicillof, D. Monteverde, and A. Olivero. Speeding up model checking of timed-models by combining scenario specialization and

live component analysis. In *Formal Modeling and Analysis of Timed Systems*, pages 58–72. Springer, 2009.

[29] V. Braberman, N. Kicillof, and A. Olivero. A scenario-matching approach to the description and model checking of real-time properties. *Software Engineering, IEEE Transactions on*, 31(12):1028–1041, 2005.

[30] W. Cazzola and S. Pini. Join point patterns: a high-level join point selection mechanism. In *Models in Software Engineering*, pages 17–26. Springer, 2007.

[31] E. M. Clarke, O. Grumberg, and D. A. Peled. *Model checking*. MIT press, 1999.

[32] R. Cobleigh, G. Avrunin, and L. Clarke. User guidance for creating precise and accessible property specifications. In *Proceedings of the 14th ACM SIGSOFT international symposium on Foundations of software engineering*, page 218. ACM, 2006.

[33] D. M. Cohen, S. R. Dalal, J. Parelius, and G. C. Patton. The combinatorial design approach to automatic test generation. *Software, IEEE*, 13(5):83–88, 1996.

[34] S. Dalal, A. Jain, N. Karunanithi, J. Leaton, C. Lott, G. Patton, and B. Horowitz. Model-based testing in practice. 1999.

[35] S. David and A. Orni. Property-by-example guide: a handbook of psl/sugar examples-prosyd deliverable d1. 1/3, 2005.

[36] L. Dillon, G. Kutty, L. Moser, P. Melliar-Smith, and Y. Ramakrishna. A graphical interval logic for specifying concurrent systems. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 3(2):131–165, 1994.

[37] N. D'Ippolito, V. Braberman, N. Piterman, and S. Uchitel. Synthesis of Live Behaviour Models. In *Proceedings of the 18th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM SIGSOFT, 2010.

[38] N. D'Ippolito, V. A. Braberman, N. Piterman, and S. Uchitel. Synthesizing nonanomalous event-based controllers for liveness goals. *ACM Trans. Softw. Eng. Methodol.*, 22(1):9, 2013.

[39] R. Douence, P. Fradet, and M. Südholt. Composition, reuse and interaction analysis of stateful aspects. In *Proceedings of the 3rd international conference on Aspect-oriented software development*, pages 141–150. ACM, 2004.

[40] P. Durr, L. Bergmans, and M. Aksit. Reasoning about semantic conflicts between aspects. In *ECOOP 2006*. Citeseer, 2005.

[41] P. Durr, T. Staijen, L. Bergmans, and M. Aksit. Reasoning about semantic conflicts between aspects. In *2nd European Interactive Workshop on Aspects in Software (EIWAS?05)*. Citeseer, 2005.

[42] M. Dwyer, G. Avrunin, and J. Corbett. Specification Patterns Web Site. In *http://patterns.projects.cis.ksu.edu/documentation/patterns.shtml*.

[43] M. Dwyer, G. Avrunin, and J. Corbett. Patterns in property specifications for finite-state verification. In *Proceedings of the 21st International Conference on Software Engineering ICSE*, volume 99, 1999.

[44] M. Eichberg, M. Mezini, and K. Ostermann. Pointcuts as functional queries. *Programming Languages and Systems*, pages 366–381, 2004.

[45] C. Eisner and D. Fisman. *A Practical Introduction to PSL (Series on Integrated Circuits and Systems)*. Springer-Verlag New York, Inc., Secaucus, NJ, 2006.

[46] M. R. Gary and D. S. Johnson. Computers and intractability: A guide to the theory of np-completeness, 1979.

[47] D. Giannakopoulou and J. Magee. Fluent model checking for event-based systems. In *Proceedings of the 9th European software engineering conference*, page 266. ACM, 2003.

[48] M. Grohe and N. Schweikardt. The succinctness of first-order logic on linear orders. In *Logic in Computer Science, 2004. Proceedings of the 19th Annual IEEE Symposium on*, pages 438–447. IEEE, 2004.

[49] K. Gybels and J. Brichau. Arranging language features for more robust pattern-based crosscuts. In *Proceedings of the 2nd international conference on Aspect-oriented software development*, pages 60–69. ACM, 2003.

[50] D. Harel and R. Marelly. Playing with time: On the specification and execution of time-enriched lscs. In *Modeling, Analysis and Simulation of Computer and Telecommunications Systems, 2002. MASCOTS 2002. Proceedings. 10th IEEE International Symposium on*, pages 193–202. IEEE, 2002.

[51] S. Herrmann. Object teams: Improving modularity for crosscutting collaborations. *Objects, Components, Architectures, Services, and Applications for a Networked World*, pages 248–264, 2009.

[52] K. Hoffman and P. Eugster. Bridging Java and AspectJ through explicit join points. In *Proceedings of the 5th international Symposium on Principles and Practice of Programming in Java*, pages 63–72. ACM, 2007.

[53] G. Holzmann. The logic of bugs. *ACM Software Engineering Notes*, 27(6):87, 2002.

[54] IEEE-Commission et al. Ieee standard for property specification language (psl). Technical report, Technical report, IEEE, 2005. IEEE Std 1850-2005, 2005.

[55] M. Jackson. The world and the machine. In *Software Engineering, 1995. ICSE 1995. 17th International Conference on*, pages 283–283. IEEE, 1995.

[56] B. Jobstmann, S. Galler, M. Weiglhofer, and R. Bloem. Anzu: A tool for property synthesis. In *Computer Aided Verification*, pages 258–262. Springer, 2007.

[57] S. Katz. Aspect categories and classes of temporal properties. In *Transactions on aspect-oriented software development I*, pages 106–134. Springer, 2006.

[58] S. Katz and H. Israel. Diagnosis of harmful aspects using regression verification. *FOAL: Foundations Of Aspect-Oriented Languages*, pages 1–6, 2004.

[59] A. Kellens, K. Mens, J. Brichau, and K. Gybels. Managing the evolution of aspect-oriented software with model-based pointcuts. *ECOOP 2006–Object-Oriented Programming*, pages 501–525, 2006.

[60] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An overview of aspectj. In *ECOOP 2001 Object-Oriented Programming*, pages 327–354. Springer, 2001.

[61] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. *Aspect-oriented programming.* Springer, 1997.

[62] G. Kiczales and M. Mezini. Separation of concerns with procedures, annotations, advice and pointcuts. *ECOOP 2005-Object-Oriented Programming*, pages 195–213, 2005.

[63] J. Klein, L. Hélouët, and J.-M. Jézéquel. Semantic-based weaving of scenarios. In *Proceedings of the 5th international conference on Aspect-oriented software development*, pages 27–38. ACM, 2006.

[64] S. Konrad and B. Cheng. Real-time specification patterns. In *Proceedings of the 27th ICSE*, pages 372–381. ACM, 2005.

[65] C. Koppen and M. Storzer. PCDiff: Attacking the fragile pointcut problem. In *First European Interactive Workshop on Aspects in Software (EIWAS)*, 2004.

[66] J. Kramer and J. Magee. Concurrency: State models and java programs. *ISBN: 0-471-98710-7, John Wiley and Sons*, 1999.

[67] I. Krka, Y. Brun, G. Edwards, and N. Medvidovic. Synthesizing partial component-level behavior models from system specifications. In *Proceedings of the the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, pages 305–314. ACM, 2009.

[68] O. Kupferman, N. Piterman, and M. Vardi. Extended temporal logic revisited. *CONCUR 2001?Concurrency Theory*, pages 519–535, 2001.

[69] O. Kupferman and M. Y. Vardi. Module checking. In *Computer Aided Verification*, pages 75–86. Springer, 1996.

[70] F. Laroussinie, N. Markey, and P. Schnoebelen. Temporal logic with forgettable past. In *Proceedings- Symposium on Logic in Computer Science. pp. 383-392. 2002*, 2002.

[71] K. G. Larsen and B. Thomsen. A modal process logic. In *Logic in Computer Science, 1988. LICS'88., Proceedings of the Third Annual Symposium on*, pages 203–210. IEEE, 1988.

[72] G. T. Leavens, A. L. Baker, and C. Ruby. Preliminary design of jml: A behavioral interface specification language for java. *ACM SIGSOFT Software Engineering Notes*, 31(3):1–38, 2006.

[73] D. Luckham. *Event processing for business: organizing the real-time enterprise*. Wiley, 2011.

[74] M. Mahoney, A. Bader, T. Elrad, and O. Aldawud. Using aspects to abstract and modularize statecharts. *Proc. 5th Wsh. Aspect-Oriented Modeling, Lisboa*, 2004.

[75] N. Markey. Temporal logic with past is exponentially more succinct. *EATCS Bull*, 79:122–128, 2003.

[76] H. Masuhara and K. Kawauchi. Dataflow pointcut in aspect-oriented programming. *Programming Languages and Systems*, pages 105–121, 2003.

[77] J. McCarthy and P. Hayes. *Some philosophical problems from the standpoint of artificial intelligence*. Stanford University, 1968.

[78] K. Mehner, M. Monga, and G. Taentzer. Interaction analysis in aspect-oriented models. 2006.

[79] N. Noda and T. Kishi. An aspect-oriented modeling mechanism based on state diagrams. In *9 th Int?l Workshop on Aspect-Oriented Modeling (AOM?06)*, 2006.

[80] D. L. Parnas. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 15(12):1053–1058, 1972.

[81] D. Paun and M. Chechik. Events in linear-time properties. In *Proceedings of 4th International Conference on Requirements Engineering*. Citeseer, 1999.

[82] N. Pelov, M. Denecker, and M. Bruynooghe. Well-founded and stable semantics of logic programs with aggregates. *Theory and Practice of Logic Programming*, 7(3):301, 2007.

[83] N. Piterman, A. Pnueli, and Y. Sa'ar. Synthesis of reactive (1) designs. *Lecture notes in computer science*, 3855:364, 2006.

[84] A. Pnueli. The temporal logic of programs. In *Foundations of Computer Science, 1977., 18th Annual Symposium on*, pages 46–57. IEEE, 1977.

[85] A. Pnueli. Applications of temporal logic to the specification and verification of reactive systems: a survey of current trends. *Current trends in Concurrency*, pages 510–584, 1986.

[86] A. Post and J. Hoenicke. Formalization and analysis of real-time requirements: a feasibility study at bosch. In *Verified Software: Theories, Tools, Experiments*, pages 225–240. Springer, 2012.

[87] A. Post, I. Menzel, J. Hoenicke, and A. Podelski. Automotive behavioral requirements expressed in a specification pattern system: a case study at bosch. *Requir. Eng.*, 17(1):19–33, 2012.

[88] M. Pradella, P. San Pietro, P. Spoletini, and A. Morzenti. Practical model checking of LTL with past. In *ATVA03: 1st Workshop on Automated Technology for Verification and Analysis*. Citeseer, 2003.

[89] J. Pryor and C. Marcos. Solving conflicts in aspect-oriented applications. *Proceedings of the Fourth ASSE*, 32, 2003.

[90] R. W. R. and K.Viggers. Implementing protocols via declarative event patterns. In *ACM Sigsoft International Symposium on FSE(FSE-12)*, pages 158–169, 2004.

[91] H. Rajan and G. Leavens. Ptolemy: A language with quantified, typed events. *ECOOP 2008–Object-Oriented Programming*, pages 155–179, 2008.

[92] C. Sánchez and M. Leucker. Regular linear temporal logic with past. In *Verification, Model Checking, and Abstract Interpretation*, pages 295–311. Springer, 2010.

[93] S. Sandra I. Casas, J. J. Baltasar García Perez-Schofield, and C. Claudia A. Marcos. MEDIATOR: an AOP Tool to Support Conflicts among Aspects. *International Journal of Software Engineering and Its Applications (IJSEIA)*, 3(3):33–44, 2009.

[94] B. Sengupta and R. Cleaveland. Triggered message sequence charts. *ACM SIGSOFT Software Engineering Notes*, 27(6):167–176, 2002.

[95] G. Sibay, S. Uchitel, and V. Braberman. Existential live sequence charts revisited. In *Software Engineering, 2008. ICSE'08. ACM/IEEE 30th International Conference on*, pages 41–50. IEEE, 2008.

[96] G. E. Sibay, S. Uchitel, V. Braberman, and J. Kramer. Distribution of modal transition systems. In *FM 2012: Formal Methods*, pages 403–417. Springer, 2012.

[97] M. Smith, G. Holzmann, and K. Etessami. Events and constraints: A graphical editor for capturing logic requirements of programs. In *Proceedings of the 5th IEEE RE*, pages 14–22. Citeseer, 2001.

[98] R. Smith, G. Avrunin, L. Clarke, and L. Osterweil. Propel: An approach supporting property elucidation. In *ICSE*, volume 24, pages 11–21, 2002.

[99] M. Storzer, R. Sterr, and F. Forster. Detecting precedence-related advice interference. In *ASE*, pages 317–322. IEEE, 2006.

[100] K. J. Sullivan, W. G. Griswold, H. Rajan, Y. Song, Y. Cai, M. Shonle, and N. Tewari. Modular aspect-oriented design with xpis. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 2009.

[101] T. Tourwé, J. Brichau, and K. Gybels. On the existence of the AOSD-evolution paradox. *SPLAT*, 2003.

[102] Y. Tsay, Y. Chen, M. Tsai, K. Wu, and W. Chan. Goal: A graphical tool for manipulating büchi automata and temporal formulae. *Tools and Algorithms for the Construction and Analysis of Systems*, pages 466–471, 2007.

[103] Y. Tsay, M. Tsai, J. Chang, and Y. Chang. Büchi store: an open repository of büchi automata. *Tools and Algorithms for the Construction and Analysis of Systems*, pages 262–266, 2011.

[104] S. Uchitel, J. Kramer, and J. Magee. Negative scenarios for implied scenario elicitation. In *Proc. of FSE '02*, pages 109–118. ACM Press, 2002.

[105] M. Utting and B. Legeard. *Practical model-based testing: a tools approach*. Morgan Kaufmann, 2007.

[106] F. Van Harmelen, V. Lifschitz, and B. Porter. *Handbook of knowledge representation*, volume 1. Elsevier Science, 2008.

[107] A. Van Lamsweerde. Goal-oriented requirements engineering: A guided tour. In *Requirements Engineering, 2001. Proceedings. Fifth IEEE International Symposium on*, pages 249–262. IEEE, 2001.

[108] M. Vardiy and P. Wolperz. Reasoning about infinite computations. 1994.

[109] M. Veanes and W. Schulte. Protocol Modeling with Model Program Composition. *LECTURE NOTES IN COMPUTER SCIENCE*, 5048:324, 2008.

[110] J. Vlissides, R. Helm, R. Johnson, and E. Gamma. Design patterns: Elements of reusable object-oriented software. *Reading: Addison-Wesley*, 49:120, 1995.

[111] N. Weston, R. Chitchyan, and A. Rashid. Formal semantic conflict detection in aspect-oriented requirements. *Requirements engineering*, 14(4):247–268, 2009.

[112] T. Wilke. CTL+ is exponentially more succinct than CTL. In *Foundations of Software Technology and Theoretical Computer Science*, pages 110–121. Springer, 1999.

[113] P. Wolper. Temporal logic can be more expressive. *Information and control*, 56(1-2):72–99, 1983.

[114] P. Wolper, M. Vardi, and A. Sistla. Reasoning about infinite computation paths. In *Foundations of Computer Science, 1983., 24th Annual Symposium on*, pages 185–194. IEEE, 1983.

[115] Z. Wu. On the expressive power of qltl. In *Proceedings of the 4th international conference on Theoretical aspects of computing*, pages 467–481. Springer-Verlag, 2007.