Dpto. de Tecnologías de las Comunicaciones
ETSI de Telecomunicación
Universidad de Vigo
Campus Universitario s/n
E-36200 Vigo

TESIS DOCTORAL

DISEÑO Y VERIFICACIÓN DE SISTEMAS DISTRIBUIDOS MEDIANTE LA APLICACIÓN COMBINADA DE MÉTODOS FORMALES

Autor: Alberto Gil Solla
Ingeniero de Telecomunicación

Director: José J. Pazos Arias
Doctor Ingeniero de Telecomunicación

1999

Tesis Doctoral: DISEÑO Y VERIFICACIÓN DE SISTEMAS DISTRIBUIDOS MEDIANTE LA APLICACIÓN COMBINADA DE MÉTODOS FORMALES

Autor: Alberto Gil Solla

Director: Dr. José J. Pazos Arias

TRIBUNAL CALIFICADOR

Presidente:

Vocales:

Secretario:

CALIFICACIÓN:

Vigo, a 18 de Enero de 2000

Presidente: Vocales: Secretario:

A mis padres

Agradecimientos

Difílcilmente podré, en tan limitado espacio, agradecer adecuadamente todo y a todos los que algo debo en relación al trabajo que aquí presento. Si algo tiene de meritorio, sin duda es fruto de muchos más de los que menciono en estas líneas. Los defectos que contiene son reflejo de mis limitaciones y han sobrevivido muy a pesar de todos los que, de una u otra forma, trataron de ayudarme.

Primero, y a gran distancia del resto, mis padres. Sin su sacrificio, la ayuda de los que siguen habría sido estéril.

A mis hermanos y demás familia. Su contribución es especial. Y continúa.

Gracias, por supuesto, a Jose, mi director de tesis. Sus consejos, correcciones y confianza me señalaron el camino cuando lo necesité.

A Ana, cuya incansable labor de jardinería ha hecho posible que este trabajo pueda ser mostrado y no sólo leído.

A Raúl, por todos estos años, ¡que ya empiezan a ser muchos!

A mis compañeros Andrés, Ardao, Cándido, Jorge, Manolo, Manuel y Rebeca, por el buen ejemplo y la grata compañía.

Al resto de mis compañeros del Área de Ingeniería Telemática, por toda la ayuda recibida a lo largo de este tiempo, globalmente muy positivo.

A los alumnos, hoy ingenieros, que han colaborado conmigo a lo largo de todos estos años, especialmente a los directamente involucrados en este trabajo: Íñigo, Joaquín, Javier, Fernando, Elena, Chos y David.

A todos mis amigos y amigas , ¡gracias por los buenos momentos!. Especialmente a los de Domayo, mi pueblo.

A Claudia, por tanto cariño, aunque sin su ayuda nunca hubiera conseguido demorar tanto este momento.

Termino por Miguel Strogoff, precisamente porque, al menos en mi memoria, el fue el primero. ¡Gracias, Marita!

Quisiera también agradecer a la Xunta de Galicia el apoyo que nos ha prestado. La financiación recibida a través del proyecto XUGA 32206A97, “LIRA: Contorno software de desenvolvemento de aplicacións con técnicas de descripción formal”, nos ha permitido realizar este trabajo con los medios materiales apropiados.

Resumen

Esta tesis se ubica en el campo de la Ingeniería del *Software*, disciplina cuyo objeto es facilitar el desarrollo de productos *software* de calidad, que satisfagan las expectativas de los usuarios.

Para lograrlo, muchos autores defienden que, como ocurre en otras ingenierías, es necesario formalizar matemáticamente las etapas de desarrollo del sistema, lo que disminuiría la probabilidad de que la implementación final contenga funcionalidades erróneas. Esta aplicación de métodos formales al desarrollo del *software* pasa, inevitablemente, por una especificación formal del sistema, es decir, por una descripción del mismo mediante una o varias notaciones de carácter matemático.

Dada la carga computacional que conlleva la verificación matemática de los sistemas de gran tamaño, un procedimiento muy común cuando se trabaja con métodos formales consiste en el desarrollo del sistema a través de una serie de refinamientos formalmente verificados. El proceso de refinamiento concluye cuando se dispone de una especificación con una estructura apropiada y el suficiente nivel de detalle para realizar una implementación directa, en muchos casos semiautomática.

Entre los partidarios de los métodos formales, cada día es más generalizada la opinión de que el método, notación o técnica formal que conviene emplear depende de la fase de desarrollo del sistema [Bro96]. En las fases iniciales del proyecto es preferible utilizar técnicas orientadas a propiedades, que permitan capturar los requisitos del sistema. Sin embargo, al avanzar en el desarrollo se impone la necesidad de emplear técnicas constructivas, que permitan estructurar el sistema e identificar subcomponentes de cara a la implementación.

Por tanto, las herramientas de apoyo al desarrollo con métodos formales deben combinar varias técnicas para obtener lo mejor de cada una en cada fase de la evolución del sistema.

LIRA [PA95] es un entorno transformacional orientado al desarrollo de sistemas *software* con métodos formales. Es una herramienta que sigue el principio de refinamientos sucesivos: se parte de una arquitectura inicial descrita en LOTOS y se refina mediante transformaciones formalmente verificadas.

El proceso de refinamientos, por tanto, se articula en torno a una técnica constructiva como es LOTOS [ISO89b]. Ello hace que la herramienta sea más

apropiada a partir del momento en que disponemos de una arquitectura inicial.

El propósito de esta tesis es ampliar la funcionalidad del entorno transformacional LIRA para dar soporte al proceso de captura de una arquitectura inicial. Esta descripción preliminar debe servir como punto de partida para el proceso de refinamientos.

Para ello, en esta tesis se define y formaliza un procedimiento metodológico para obtener la arquitectura inicial a partir de los requisitos de usuario. Este procedimiento, por estar dirigido a la fase inicial del desarrollo, se basa en técnicas formales orientadas a propiedades (concretamente en lógica temporal), más útiles a la hora de identificar y verificar las propiedades del sistema.

A lo largo de las etapas del procedimiento desarrollado en esta tesis se han realizado varias aportaciones, que se pueden resumir en:

- Se ofrece un marco para la especificación de comportamientos conocidos del sistema (y del intercambio de información que se produzca con el entorno) y su solapamiento en una estructura que los combine y unifique.
- Se define un formalismo que permite la especificación, mediante lógica temporal, de las propiedades interesantes del sistema y su clasificación siguiendo criterios de homogeneidad.
- Hemos diseñado un algoritmo, basado en *model checking*, para verificar automáticamente las propiedades citadas sobre la representación matemática del sistema.
- Se establecen mecanismos para ayudar al diseñador en la toma de decisiones y aportarle sugerencias sobre las modificaciones que es preciso llevar a cabo si el sistema no cumple una propiedad.
- A partir de la descripción final del sistema, generamos una especificación LOTOS equivalente (en el nuevo estándar ELOTOS [ISO98]) que constituirá la arquitectura inicial. Esta descripción del sistema es el nexo de unión con los mecanismos de LIRA, previos a los trabajos de esta tesis, que dan soporte al proceso de refinamientos.
- Se han implementado todas las notaciones y algoritmos desarrollados en los apartados anteriores en el entorno transformacional LIRA.

Abstract

This PhD. thesis belongs to the Software Engineering research field, a discipline concerned with theories and tools which are needed to help in the development of quality software products that meet the user's expectactions.

To achieve this goal, many authors argue that, as it happens in other engineering areas, it is necessary to increase the mathematical modelling of the system development stages, which would reduce the probability of wrong functionalities in the final implementation. The application of formal methods in software development requires a formal specification of the system, i.e., a system description by means of one or several mathematical notations.

Mathematical verification of large systems often needs a lot of computational resources. Thus, a very usual procedure entails arranging the system development in several successive refinements. Each refinement is formally verified and the process ends when a proper specification is achieved. The final specification must include an appropriate structure and be detailed enough to carry out a direct implementation, frequently semiautomatic.

There is a wide agreement, among those in favor of formal methods, to consider that the more appropriated technique for a particular system depends on the development phase [Bro96]. In the initial stages, it is better to use property oriented techniques, because they adapt to system requirement capture. However, as development progresses, it is more convenient to employ constructive techniques, which allow to obtain the system structure and to identify subsystems with a view to implementation.

So, formal development support tools must combine several techniques to take the most of each one in every development phase.

LIRA [PA95] is a transformational environment oriented to system development with formal methods. The tool supports the procedure of correctness-preserving successive refinements: it starts with an initial architecture described in LOTOS and refines it through formally verified transformations.

As we can see, the refinement process is built around a constructive technique, as LOTOS is [ISO89b]. For that reason, the tool is more appropriate once the initial architecture has been achieved.

The main goal of this PhD. thesis is to extend the functionality of the transformational environment LIRA to support the capture of the initial ar-

chitecture. This preliminary description must play the role of the start point in the refinement process.

In order to achieve it, a methodological procedure to obtain the initial architecture starting from the user requirements is defined and formalized in this PhD. thesis. This procedure is oriented to the initial phases of the development and, therefore, based on property oriented formal techniques (to be precise, in temporal logic). These kind of techniques are more useful to identify and verify system properties.

Throughout the stages of the procedure developed in this PhD. thesis, we have made several contributions. They can be summarize in:

- We offer a framework for the specification of well-known behaviours of the system and the information interchange between the system and its environment. Procedures for their unification and folding in a single structure have been designed.
- We define a formalism for the specification, using temporal logic, of the system properties and their classification following homogeneity criteria.
- We have designed an algorithm, based in model checking, to automatically verify the aforementioned properties in the mathematical representation of the system.
- Some mechanisms are established to help the designer to take decisions and to give him suggestions about the necessary modifications if the system does not fulfil a requirement.
- From the final system description, we create an equivalent LOTOS specification (in the new standard ELOTOS [ISO98]) which must serve as the initial architecture. This system description is the link with the LIRA mechanisms, previous to this work, which support the refinement process.
- All the notations and algorithms developed in this PhD. thesis have been implemented in the transformational environment LIRA.

Índice general

I Introducciones 1

1. La Ingeniería del Software 3

1.1. Introducción . . . 3

1.1.1. Los sistemas distribuidos . . . 4

1.2. Los procesos de desarrollo de software . . . 5

1.2.1. Procesos de desarrollo de software . . . 7

1.3. Las Técnicas de Descripción Formal . . . 11

1.3.1. Clasificación de las técnicas formales . . . 15

1.4. Fases de diseño con técnicas formales . . . 18

1.4.1. Captura de requisitos . . . 19

1.4.2. Refinamientos sucesivos . . . 20

1.4.3. Implementación final . . . 21

1.5. Objetivos de la tesis . . . 21

1.6. Organización de la memoria . . . 22

2. Estado del arte 25

2.1. Introducción . . . 25

2.2. Especificación . . . 27

2.2.1. Técnicas orientadas a sistemas secuenciales . . . 28

2.2.2. Técnicas orientadas a sistemas concurrentes . . . 31

2.2.3. LOTOS . . . 34

2.3. Simulación y transformación . . . 37

2.3.1. Transformaciones . . . 38

2.3.2. Sistemas de Transiciones Etiquetadas 39
2.4. Verificación 41
2.4.1. Demostradores de teoremas 41
2.4.2. Model checking 44
2.4.3. Líneas de trabajo 47

3. La lógica temporal 49
3.1. Introducción 49
3.2. La lógica temporal 50
3.2.1. Semántica temporal 52
3.2.2. Aplicación a sistemas reales 53
3.3. Verificación con lógica 55
3.3.1. Equivalencia de juegos 57
3.3.2. Tableau 59
3.3.3. Equivalencia de sistemas 60
3.3.4. Verificación de sistemas con paso de valores 61
3.4. Propiedades de interés 62
3.4.1. Clasificaciones 63

4. Objetivos de la Tesis 67
4.1. Captura de la arquitectura inicial 67
4.2. Etapas del procedimiento 68
4.2.1. Especificación de los eventos del sistema 69
4.2.2. Especificación y solapamiento de las trazas iniciales . . . 70
4.2.3. Formulación y verificación de propiedades 72
4.2.4. Modificación del sistema 73
4.2.5. Síntesis de la arquitectura inicial 74
4.3. Formalización de las etapas 75

II Desarrollo 77

5. Sistemas de Transiciones Simbólicos 79

5.1. Introducción 79
5.2. Sistemas de transiciones simbólicos 80
5.2.1. Definición formal 81
5.3. Preservación de la coherencia 83
5.3.1. Eventos fijos 84
5.3.2. Eventos prohibidos 85

6. La lógica LTCS 87
6.1. Descripción de la lógica 87
6.1.1. Eventos 88
6.1.2. Estructura de un requisito 91
6.1.3. Las fórmulas LTCS 92
6.2. Especificación de propiedades con LTCS 96
6.2.1. Finalidades 97
6.2.2. Invarianzas 100
6.2.3. Precedencias 102

7. Solapamiento de las trazas iniciales 105
7.1. Introducción 105
7.2. Especificación de trazas mediante LTCS 106
7.3. Solapamiento de las trazas 108
7.3.1. Unicidad de las variables 108
7.3.2. Solapamiento de eventos 109
7.3.3. Solapamiento de guardas 111
7.3.4. Procedimiento general de solapamiento 114
7.4. Ejemplo 115

8. Verificación de propiedades temporales 119
8.1. Introducción 119
8.2. Relación de satisfacción 120
8.2.1. Simplificaciones 122
8.3. Una lógica de tres estados 123

8.3.1. Los operadores lógicos 124
8.4. Verificación de eventos simbólicos 125
8.4.1. Condición de verificación 126
8.4.2. Condición de rechazo 128
8.5. Estados de aplicabilidad 129
8.5.1. La estructura CEA 130
8.6. Recursividad 132
8.6.1. Implementación 133
8.6.2. Ámbito de aplicación 135
8.7. Algoritmo de verificación 136
8.8. Análisis de los resultados 144

9. Modificaciones funcionales 147

9.1. Introducción 147
9.2. Modelo desarrollado 149
9.2.1. Fases del procedimiento 150
9.3. Generación de la información sobre el incumplimiento 151
9.3.1. Algoritmo de generación de la IVR 153
9.4. Generación de las sugerencias 162
9.4.1. Tipos de sugerencias 162

III Conclusiones 175

10. Conclusiones y trabajo futuro 177

10.1. Contribuciones 179
10.2. Conclusiones 181
10.3. Trabajo futuro 183

IV Apéndices 185

A. La herramienta LIRA 187

A.1. Introducción 187
A.2. El entorno transformacional LIRA 188
A.2.1. Funcionalidades previas de LIRA 188
A.2.2. Implementación 189
A.3. Funcionalidades añadidas a LIRA 190
A.4. Arquitectura de LIRA 192
A.5. Implementación del núcleo 195
A.5.1. Evolución de la arquitectura inicial 197
A.6. Interfaz de LIRA 198

B. Ejemplo de desarrollo **201**
B.1. Identificación de los eventos 202
B.2. Solapamiento de las trazas iniciales 203
B.3. Verificación de propiedades 204
B.3.1. P1: Ausencia de bloqueo 204
B.3.2. P2: Todas las conexiones deben ser liberadas 206
B.3.3. P3: Todos los recursos están disponibles 207
B.3.4. P4: Las solicitudes de recursos pueden duplicarse 208
B.3.5. P5: Hay que esperar el asentimiento 209
B.4. Obtención de sugerencias 210
B.4.1. S1: Las solicitudes de recursos pueden duplicarse 211
B.4.2. S2: Los asentimientos no pueden perderse 212
B.4.3. S3: Las solicitudes de conexión pueden duplicarse 213
B.4.4. S4: Puede aparecer otra solicitud de conexión 214
B.4.5. S5: Avisar cuando venza una temporización 216
B.4.6. S6: Los asentimientos negativos se registran 217
B.5. Traducción a ELOTOS 218

Bibliografía **221**

Índice de figuras

1.1. Modelo de cascada . . . 8
1.2. Modelo en espiral . . . 10

2.1. Representación gráfica de un STE . . . 40

4.1. Captura de la arquitectura inicial . . . 69

5.1. Sistema de transiciones simbólico . . . 83

7.1. Traza recursiva . . . 108
7.2. Solapamiento simple . . . 110
7.3. Solapamiento de guardas . . . 111
7.4. Patrones no equivalentes . . . 112
7.5. Guardas equivalentes . . . 112
7.6. Guardas no equivalentes . . . 113
7.7. Solapamiento de T1 y T2 . . . 116
7.8. Solapamiento de T3 . . . 118
7.9. Solapamiento de T4 . . . 118

A.1. Antigua organización modular de LIRA . . . 190
A.2. Arquitectura de LIRA . . . 194
A.3. Núcleo de LIRA . . . 197
A.4. Etapas del desarrollo . . . 198
A.5. Interfaz de LIRA . . . 199

B.1. Resultado del solapamiento 205
B.2. Primera modificación 212
B.3. Segunda modificación 214
B.4. Tercera modificación 216
B.5. Especificación ELOTOS del sistema 219

Parte I

Introducciones

Capítulo 1

La Ingeniería del Software

1.1. Introducción

En las últimas décadas hemos asistido a la incesante penetración de los ordenadores en nuestra sociedad. Hoy en día, directa o indirectamente, influyen de manera notable en la rutina diaria de los habitantes de cualquier país desarrollado, siendo responsables de muchos de los elementos que determinan su calidad de vida.

Cada vez más, los ordenadores forman parte de la actividad cotidiana de múltiples sectores de la sociedad: el almacenamiento, control y distribución de grandes volúmenes de información; la tutela de incontables procesos productivos en el sector industrial; el control de situaciones de alto riesgo; la oferta de actividades lúdicas y educativas, etc; son sólo algunos de los ejemplos en los que, total o parcialmente, el ordenador juega un papel determinante, en muchos casos imprescindible.

Y, evidentemente, es el *software* que se ejecuta en esos ordenadores el que, en última instancia, se encarga de gobernar de forma adecuada todos los procesos citados. Es el *software* el que, en la mayor parte de los casos, conforma la identidad de un sistema informático y proporciona a su funcionalidad las características exclusivas que posee.

Por la misma razón, suelen ser los problemas relacionados con el *software* los más preocupantes en el desarrollo y evolución de un sistema informático y los que pueden causar más perjuicios al entorno que le rodea.

Este hecho se acentúa cada vez más, con el constante incremento de tamaño y complejidad de los grandes sistemas informáticos (control del tráfico aéreo, gobierno de centrales nucleares, automatización de los mercados de capitales, etc.). Cada día más, el *software* aparece como elemento de primer orden en el desarrollo de tareas críticas, como puedan ser las ya descritas o todas aquellas relacionadas con el sector de la sanidad. Evidentemente, la necesidad de una gran fiabilidad en este tipo de aplicaciones está fuera de toda duda, habida cuenta de los efectos que un fallo puede tener en la vida de las personas o en la economía de un país.

Es por ello que, ya desde muy pronto (mediados de los cincuenta, [BS93]), el coste de un sistema informático depende principalmente del coste de los programas. En cuanto los avances en la cantidad y calidad de los recursos *hardware* permitieron abordar la construcción de sistemas de información de tamaño medio-alto (mediados de los sesenta), comenzaron a aflorar los problemas del *software*: los proyectos eran entregados tarde, desbordaban el presupuesto inicial, eran difíciles de mantener, poco eficientes o faltos de fiabilidad. Muy a menudo, presentaban varios de esos defectos y, no pocas veces, todos [Som95].

Todo ello dio origen al nacimiento de una nueva disciplina: la ingeniería del *software* [NR69]. Su objetivo es, en líneas generales, posibilitar el desarrollo de productos *software* de calidad que satisfagan las expectativas de los usuarios. El camino hacia ello pasa, inevitablemente, por organizar la producción de los programas a través de un proceso de desarrollo del *software* que identifique etapas, objetivos y productos intermedios y finales.

1.1.1. Los sistemas distribuidos

Los problemas mencionados suelen ser proporcionales al tamaño de los sistemas informáticos y éste, ciertamente, ha aumentado constantemente en los últimos años.

El constante descenso de los precios de los ordenadores, el continuo incremento de su potencia de cálculo y la aparición de nuevas tecnologías en el campo de las comunicaciones han supuesto la rápida expansión de lo que se han denominado sistemas distribuidos. Un sistema distribuido es aquel que está formado por varios sistemas autónomos que trabajan de forma coordinada para la realización de una determinada tarea. Estos sistemas suelen estar compuestos de un conjunto de procesos concurrentes que interactúan entre sí.

Generalmente, su complejidad es muy superior a la suma de las complejidades de los sistemas secuenciales que lo integran, pues, a la complejidad de los subsistemas hay que añadir las dificultades derivadas de la comunicación y sincronización de los diversos procesos cooperantes que evolucionan en paralelo.

Además, este tipo de sistemas suelen presentar lo que se denominan propiedades emergentes. Son propiedades que no están presentes en los subsistemas estudiados de forma aislada, sino que se desprenden de la integración de varios de ellos. Este tipo de propiedades no pueden ser analizadas en cada uno de ellos sino que, en muchos casos, requieren la integración final para su estudio. Ejemplos de estas características son la fiabilidad, la seguridad, el rendimiento, etc.

Una característica común de muchos sistemas distribuidos (sistemas operativos, protocolos de comunicación, etc.) es que su comportamiento no tiene una terminación previsible. Se caracterizan más por reaccionar a estímulos del entorno que por terminar produciendo algún tipo de resultado final. Pnueli denominó esta característica como **reactividad** [Pnu85].

Los sistemas reactivos, por tanto, se describen mejor a través de su interacción con el entorno que mediante una relación entre el conjunto de entradas y salidas. Esto dificulta todavía más su desarrollo y verificación.

1.2. Los procesos de desarrollo de software

La ingeniería del *software* es una disciplina que se ocupa de la búsqueda de métodos, técnicas y herramientas para mejorar la calidad del *software* de un sistema informático. Como comentábamos antes, el enfoque que adopta para mejorar la calidad del producto final es el estudio de las etapas que conducen a él. Un buen producto pasa por un buen proceso de desarrollo de *software*.

Un proceso de desarrollo de *software* se define [Som95] como el conjunto de actividades organizadas y resultados asociados que tienen como objetivo el desarrollo de un producto *software*.

Para ello, generalmente, se define el *ciclo de vida* de un desarrollo *software*, estableciendo una serie de etapas que deben llevarse a cabo para conseguir el producto final. Cada una de las etapas producirá unos resultados que consti-

tuirán la información necesaria para comenzar la siguiente fase. Las funciones del proceso de desarrollo son, precisamente, establecer la necesidad y ordenación de las etapas, los criterios de transición entre las mismas, las actividades que se deben llevar a cabo en cada etapa y las salidas que debe producir cada una (ya sean documentos o productos *software*) [Boe88].

Existen en la literatura múltiples modelos de desarrollo *software* [Som95, LK95]. Ello es debido, fundamentalmente, a que es muy difícil homogeneizar un proceso de creación en el que, como es el caso del *software*, hay muchas entradas diferentes y múltiples salidas válidas para cada entrada. Por ello, el grado de sistematización es muy variable en los diferentes modelos, influyendo características tales como el grado de madurez de la tecnología empleada, la cultura organizativa de la empresa o el dominio de aplicación del *software* [KS97]. Ello implica que el secuenciamiento de las actividades y los resultados varíen de un modelo a otro.

En cualquier caso, existen cuatro actividades principales que, en mayor o menor medida, están presentes en todos los procesos de desarrollo *software*:

- **Especificación.** Define la funcionalidad del sistema y las restricciones impuestas. En esta fase se establecen los objetivos del proyecto, las necesidades de los usuarios o clientes y el dominio de aplicación.

- **Diseño y desarrollo.** Se construye un sistema *software* que satisfaga la especificación anterior. Para ello se define la arquitectura del sistema, identificando el modelo de datos, los mecanismos de comunicación entre los diversos componentes y los algoritmos apropiados. Posteriormente se implementan en la tecnología y el *hardware* escogidos.

- **Validación.** Se comprueba que el sistema hace lo esperado y con los parámetros de funcionamiento requeridos (fiabilidad, eficiencia, etc.).

- **Evolución.** Una vez que el *software* ha sido instalado y puesto en funcionamiento, comienza una fase de mantenimiento en la que se subsanan los errores y se realizan modificaciones del sistema a medida que van surgiendo nuevos requisitos.

Algunas de las características deseables de un proceso de desarrollo de *software* son [Som95]:

- **Claridad.** Debe entenderse con facilidad.
- **Visibilidad.** Un proceso se dice visible cuando sus actividades producen resultados identificables externamente.
- **Facilidad de soporte.** Disponibilidad de herramientas que den soporte a todas o alguna de las actividades del proceso.
- **Fiabilidad.** Debe ser capaz de evitar o detectar posibles errores.
- **Facilidad de mantenimiento.** Requiere capacidad para incorporar nuevos requisitos o modificar alguno de los existentes.
- **Rapidez.** Debe ser capaz de obtener una implementación del sistema en un tiempo reducido.

En cualquier caso, algunas de las características descritas persiguen objetivos enfrentados (por ejemplo, fiabilidad y rapidez). Resulta generalmente imposible optimizar todos los parámetros del proceso, precisando la búsqueda de la casi siempre obligada solución de compromiso.

No existe un consenso sobre cuál es el mejor modelo de proceso de desarrollo de *software*. Se pueden emplear distintos modelos para resolver el mismo problema. Sin embargo, sí es cierto que cada modelo es más adecuado para un determinado tipo de problema, en función de los riesgos o incógnitas que plantea el sistema a diseñar. Identificar las peculiaridades del sistema y elegir el modelo adecuado puede condicionar de forma significativa la calidad del sistema final.

1.2.1. Procesos de desarrollo de software

Los procesos de desarrollo de *software* son en la actualidad un campo importante de investigación dentro de la ingeniería del *software*. En la literatura podemos encontrar una serie de modelos que, por su relevancia y difusión, comentaremos brevemente.

Modelo en cascada. Fue el primer modelo de gran difusión, sobre todo en el sector industrial por su facilidad de gestión y visibilidad.

Como se puede ver en la figura 1.1, trata las etapas identificadas como procesos estancos, claramente separados en cuanto a sus actividades y funcionalidad. Tras completar una etapa se pasa a la siguiente, en una especie de cascada. Si se detecta algún error, se vuelve atrás en la cadena.

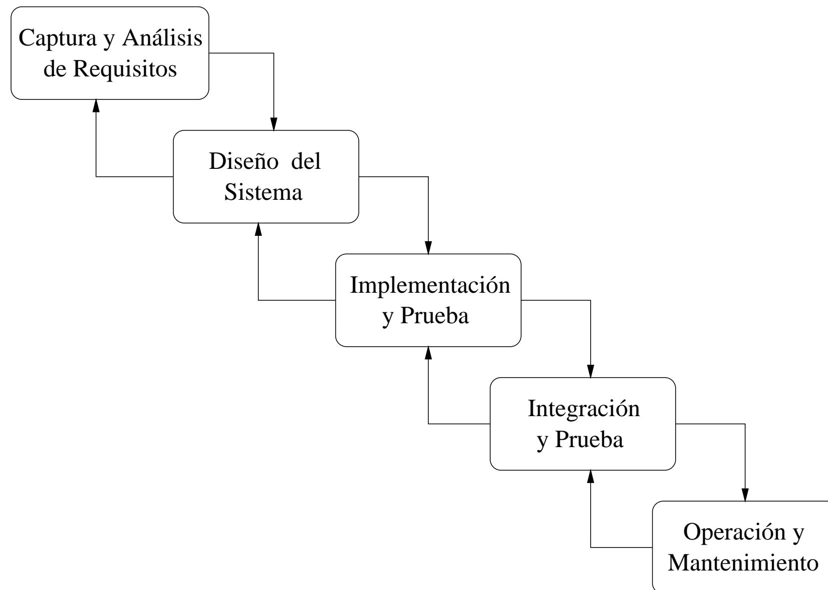

Figura 1.1: Modelo de cascada

La realidad, sin embargo, no suele ser tan secuencial. Las diversas etapas no tienen fronteras tan definidas; es frecuente que se solapen y compartan información. Los documentos de salida de cada etapa no suelen ser definitivos al comenzar la siguiente y los requisitos acostumbran a cambiar durante el desarrollo.

Todo ello resalta la falta de flexibilidad de este modelo, que proviene del rígido dimensionamiento de las diferentes etapas. En [LK95] puede encontrarse una descripción más extensa de los defectos que se le achacan, incluyendo la dificultad para acomodar prototipos, reutilizar *software* o realizar pruebas antes de la etapa de implementación.

Modelo evolutivo. En este modelo se entrelazan las etapas de especificación, desarrollo y validación. Inicialmente, a partir de aquellos requisitos de usuario que se entienden bien, se desarrolla un sistema inicial que los

satisface. Después, con la ayuda del cliente, se va refinando paulatinamente el producto obtenido, incluyendo cada vez más requisitos a medida que se consolida su descripción.

Existen dos variantes muy comunes:

- **Exploratorio.** El sistema construido de tal forma es finalizado y entregado.
- **De prototipo desechable.** El objetivo es entender plenamente los requisitos del usuario. Una vez conseguido, se desecha el prototipo y se empieza desde el principio para conseguir un diseño más robusto y estructurado.

Es un modelo muy apropiado cuando es difícil o imposible establecer desde el principio una descripción completa y detallada de las necesidades del cliente.

Sin embargo, por la velocidad del proceso de refinamientos, es difícil conseguir una adecuada visibilidad del proceso. Además, los desarrollos intermedios suelen carecer de una estructuración adecuada, lo que hace poco razonable suponer que se pueden extender para recoger nuevas necesidades del cliente. En [Boe88, Som95] se pueden encontrar críticas más extensas de este modelo.

Por todo lo comentado, este modelo suele emplearse en proyectos de pequeña-mediana escala y vida media corta (la falta de una estructura planificada dificulta las actualizaciones propias de un sistema perdurable), especialmente en sistemas de inteligencia artificial e interfaces de usuario.

Modelo transformacional. En este modelo se parte de una especificación formal inicial del comportamiento conocido del sistema. A partir de aquí, y mediante métodos matemáticos, se va transformando la especificación hasta conseguir una con el suficiente nivel de detalle como para ser implementada de forma semiautomática.

Si las transformaciones que se aplican son correctas, se puede asegurar que el sistema construido satisface la especificación. Es decir, es posible afirmar que el programa es correcto por construcción. Además, se puede proceder al mantenimiento del sistema sobre la especificación, aplicando posteriormente las mismas transformaciones que en el proceso original.

En el lado negativo, hay que destacar la necesidad de disponer de una especificación inicial apropiada, la exigencia de diseñadores altamente cualificados y la falta de experiencia en la aplicación de este método a proyectos de grandes dimensiones.

Modelo en espiral. El modelo en espiral [Boe88] nace con el objetivo de integrar las ventajas de los anteriores, amortiguando sus defectos. Su principal característica es la consideración del riesgo como elemento determinante en la toma de decisiones sobre los pasos a seguir. Los riesgos subyacentes, fruto generalmente de la falta de información, son el elemento conductor del proceso de desarrollo.

El modelo recibe su nombre por la forma que toma la evolución del proceso de desarrollo que propone: una espiral (figura 1.2). Cada ciclo de la espiral simula una fase del desarrollo, mientras que los diversos sectores establecen las actividades a realizar en cada momento.

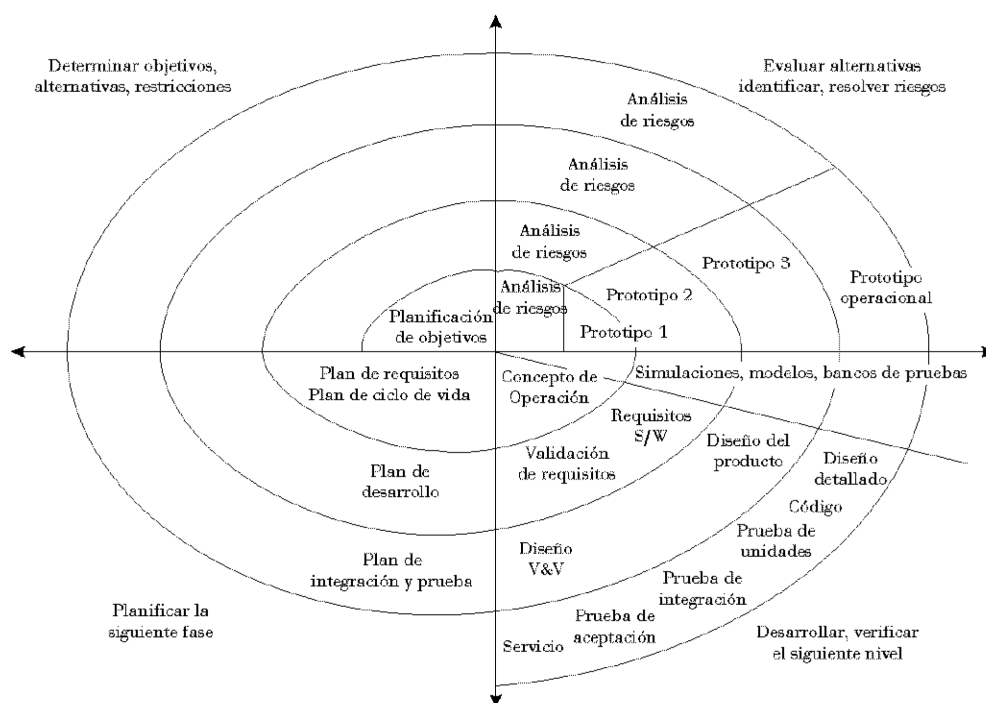

Figura 1.2: Modelo en espiral

En general no hay fases fijas. Las decisiones las debe tomar el diseñador

en función de los riesgos existentes en cada fase del proyecto.

Cada fase se compone de cuatro partes:

1. *Determinación de objetivos.* Se especifican los objetivos de la fase, se identifican posibles alternativas en función de los riesgos y se delimitan las restricciones existentes.
2. *Análisis de riesgos.* Se evalúan los riesgos identificados y se procede a su reducción en función de su naturaleza (por medio de prototipado, simulación, paso de pruebas, etc.).
3. *Desarrollo.* En función de los riesgos todavía existentes, se decide un modelo de desarrollo para esta fase. Por ejemplo, si los riesgos están relacionados con el interfaz de usuario, convendrá el modelo evolutivo. Si preocupa especialmente el presupuesto de gastos y fechas de entrega, será más conveniente el modelo de cascada.
4. *Planificación.* Se decide si se inicia otra fase de la espiral y se planifica ésta.

Como vemos, el modelo en espiral puede acomodar en su interior a los modelos anteriores y utilizarlos en función de la estrategia para resolver los riesgos subyacentes en cada fase o para cada parte del proyecto.

En general, la principal ventaja del modelo en espiral es que su flexibilidad acomoda las ventajas de los otros métodos, mientras que la consideración del riesgo atenúa sus inconvenientes. Además, la evaluación constante de alternativas favorece la reutilización de *software*. Su principal inconveniente tiene que ver con la necesidad de cierta experiencia en la identificación de los riesgos potenciales.

1.3. Las Técnicas de Descripción Formal

Independientemente del modelo empleado, existe un amplio consenso entre los expertos en el desarrollo de *software* en que, para aumentar la calidad de los sistemas, es necesario dar al ordenador un papel más importante en el proceso de desarrollo. Para ello, es fundamental poder modelar y capturar en la máquina toda la información importante acerca del problema y su solución.

En los últimos años, cobra cada vez más fuerza la idea de que una formalización matemática del *software* es el enfoque más apropiado para conseguir mejorar su calidad [1]. Ello ha dado un fuerte impulso a la definición y empleo de lo que se han denominado *métodos formales* en el proceso de desarrollo del *software*.

Los métodos formales tienen como denominador común el empleo de las matemáticas (fundamentalmente la teoría de conjuntos, el álgebra y la lógica) para la modelación del sistema que se quiere construir. Sus partidarios defienden que su empleo, a lo largo de todo el ciclo de vida, facilita el desarrollo de especificaciones claras, concisas y no ambiguas, permite el análisis funcional de la especificación y posibilita el desarrollo de implementaciones correctas con respecto a la especificación.

El objetivo es permitir la verificación y prueba del sistema a lo largo de todo su ciclo de vida. La fuerte base matemática de la mayoría de los métodos formales nos permite razonar sobre el sistema desde el primer momento, sin tener que esperar a la fase de codificación, donde los defectos de diseño están tan asentados que su corrección implica un gran coste económico.

Los beneficios potenciales que supone el poder razonar de forma temprana y constante serían:

- Una mejor comprensión del sistema.
- Una mejor comunicación con el cliente, al disponer de una descripción no ambigua de los requisitos de usuario.
- Una descripción más precisa del sistema (en cuanto a consistencia y compleción) y de las propiedades que debe cumplir.
- Una seguridad de carácter matemático de que el sistema implementado es correcto con respecto a su especificación.
- Una mayor calidad del *software* (entendiendo ésta como el grado de cumplimiento de las expectativas de los usuarios).
- Mayor productividad.

[1] De hecho, múltiples autores resaltan la aparente contradicción de una disciplina que, denominándose ingeniería, presta tan poca atención a los métodos y modelos matemáticos para lograr su objetivo.

En este ámbito, suele denominarse técnicas de especificación formal a los lenguajes cuyo vocabulario, sintaxis y semántica están formalmente definidos. Por tanto, una técnica de especificación formal es un método formal especialmente adecuado para escribir especificaciones de sistemas. Dos propiedades fundamentales de un lenguaje de este tipo son:

Expresividad. Debe ser lo suficientemente expresivo para poder describir, de forma clara, el comportamiento de los sistemas que pretende modelar. Ello depende, fundamentalmente, del conjunto de constructores y operadores matemáticos de que se compone su sintaxis.

Abstracción. Debe ser lo suficientemente abstracto para poder describir qué hace un sistema sin poner restricciones a cómo lo hace[2].

Estas dos propiedades permiten lograr una característica deseable en el proceso de desarrollo: hacer independiente la especificación de un sistema de su posterior implementación. Esta característica suele hacer muy apropiadas a las técnicas de descripción formal para la especificación de normas y estándares [BBRTL90].

El gran auge que a nivel académico han tenido los métodos formales en los últimos años ha supuesto la aparición de muchos y muy distintos métodos, técnicas, notaciones, etc. En general, no hay ningún tipo de consenso en qué método o técnica es mejor, aunque sí en la importancia de la adecuación de las características del método a la fase de desarrollo en la que se aplica [Bro96]. Por ejemplo:

- En la fase de captura de requisitos es más importante la especificación flexible de un sistema en función de sus propiedades más importantes. La descripción resultante de esta fase constituye la arquitectura inicial del sistema.

- En la fase de diseño de la arquitectura es necesario poder describir los componentes del sistema, su interfaz y cómo interactúan entre ellos. Es, por tanto, necesario poder expresar la estructura.

[2]Esto, a menudo, no es posible en la práctica, ya que los requisitos de usuario pueden imponer ciertas restricciones a la implementación (p.e. conformidad con estándares, compatibilidad con otro *software* o *hardware*, exigencias de rendimiento, etc.)

- En la fase de implementación, los componentes tienen que poder describirse de forma que su comportamiento pueda ser generado por máquinas. El proceso de traducción de un lenguaje de especificación a un lenguaje de implementación se puede automatizar en gran medida. Por ello, esta fase se suele decir que es semiautomática. Por la misma razón, resulta preferible realizar las tareas de mantenimiento a nivel de lenguaje de especificación.

Sin embargo, pese a todas las ventajas que sus defensores afirman, los métodos formales han tenido hasta el momento poca penetración en la industria, limitándose su uso a proyectos de pequeña-mediana escala [GCR94], generalmente cofinanciados por organismos de apoyo a la investigación u obligados por directivas gubernamentales. Los motivos, justificados o no, son muy variados y han provocado la publicación de interesantes artículos a medio camino entre la aclaración, la desmitificación y la autorreflexión [Hal90, BH95a, BH95b].

Alguna de las razones esgrimidas con mayor frecuencia para no utilizar métodos formales son:

- Es difícil demostrar a priori la rentabilidad del esfuerzo inicial. Los gestores de proyectos suelen optar por posturas conservativas y esperar a que sus beneficios sean claros.
- Falta de formación matemática adecuada del personal de las empresas.
- El empleo de métodos formales suele derivar en un aumento de los costes y tiempo de desarrollo.
- Desconocimiento general de este tipo de técnicas por parte de los clientes.
- No existen procedimientos documentados para su integración en los procesos de desarrollo tradicionales. Los métodos formales deberían ser técnicas complementarias a los mecanismos existentes y no exigir la transformación completa de estos.
- La mayor parte del esfuerzo investigador ha sido de carácter teórico. Son necesarias más herramientas y métodos.
- No está clara la viabilidad de su aplicación a grandes proyectos. El análisis y verificación de una especificación extensa consume mucho tiempo y

> recursos. Una aproximación más razonable consistiría en realizar verificaciones parciales, aplicando este tipo de técnicas sólo a algunas partes del sistema, las más críticas.

En los citados artículos, al tiempo que se detallan las críticas más habituales a los métodos formales, se rebaten razonadamente sus argumentos punto por punto. Evidentemente, su imparcialidad es siempre cuestionable, por tratarse de expertos en el campo de los métodos formales.

En todo caso, no deja de ser cierto que muchas de estas críticas están basadas en opiniones personales más que en hechos probados (por ejemplo, la dificultad en el aprendizaje de las notaciones). Incluso las críticas con una base documentada, como puedan ser la falta de herramientas y cursos o el incremento de costes, olvidan que alguno de estos defectos son inherentes a cualquier tecnología emergente con poca experiencia práctica. La falta de hábito implica una dificultad manifiesta en la estimación de la duración y coste de un desarrollo con métodos formales. Además, resulta difícil la comparación entre los resultados con y sin métodos formales, ya que ni los proyectos ni sus circunstancias son fácilmente equiparables.

1.3.1. Clasificación de las técnicas formales

Como comentábamos anteriormente, el panorama actual en el campo de los métodos formales dista mucho de la homogeneidad. En la literatura podemos encontrar multitud de métodos y técnicas de especificación formal, de muy diversas características, ámbito de aplicación y experiencia publicada.

En general, lo que más predomina en este campo son las técnicas de especificación, notaciones formales que favorecen la descripción precisa de sistemas y propiedades. Más escasos resultan los métodos, vistos éstos como procedimientos sistemáticos que sirven de guía al diseñador, en todo o parte del proceso de desarrollo. Esta falta de sistematización implica generalmente la necesidad o conveniencia de cierta experiencia en el desarrollo de abstracciones y la modelación de sistemas.

Los criterios de clasificación también son muy variados y de distinta relevancia en función del enfoque adoptado. Desde el punto de vista de los aspectos que modelan el sistema podríamos diferenciar entre:

- Métodos que modelan la funcionalidad del sistema. Son la mayor parte.
- Métodos orientados a la descripción de propiedades de concurrencia. Muy apropiados para la descripción de protocolos de comunicación y sistemas distribuidos.
- Métodos que permiten modelar el tiempo. Aunque menos desarrollados que los anteriores, son de gran importancia en la aplicación a sistemas de tiempo real.

La fase de desarrollo del sistema también es importante para valorar la aplicabilidad de un método formal. Podríamos dividirlos en:

- **Técnicas constructivas.** Describen directamente la funcionalidad del sistema y suelen tener como resultado una especificación formal que puede ser fácilmente ejecutable. Son, por tanto, muy apropiadas para realizar un prototipado rápido del sistema, en fases intermedias o finales.

  Su principal inconveniente suele ser la imposibilidad para especificar de forma explícita las propiedades del sistema, lo que dificulta la verificación de propiedades.

- **Técnicas no constructivas.** Describen el sistema a través de sus propiedades, por lo que también suelen llamarse técnicas orientadas a propiedades. Facilitan la verificación de propiedades, su comprobación de consistencia, etc.

  Por contra, sus inconvenientes radican en la imposibilidad de describir la estructura del sistema y la dificultad para decidir si una especificación es o no completa.

  Por ello, son más recomendables para la fase inicial de captura y análisis de requisitos del sistema.

Sin embargo, la clasificación más recurrente en la literatura está relacionada con la naturaleza del sustrato matemático que emplea cada método. Atendiendo a este criterio, podemos dividirlas en los siguientes grupos [SS97]:

Métodos algebraicos. Los métodos algebraicos modelan un sistema mediante álgebras *multisort*, como un conjunto de tipos y operaciones sobre

esos tipos. Para cada tipo se define un conjunto de valores y de operaciones sobre dichos valores. Las operaciones de un tipo se definen a través de un conjunto de axiomas o ecuaciones que especifican las restricciones que deben satisfacer las operaciones.

Estos métodos son especialmente útiles para especificar interfaces entre los diversos componentes del sistema. De este modo, una vez definidos con claridad los interfaces, el desarrollo de los componentes se puede realizar en paralelo. A este nivel, el sistema se modela como un conjunto de tipos abstractos de datos.

Un inconveniente frecuente es la dificultad para demostrar que la especificación es completa [Vli93].

Algunos de los métodos de especificación algebraicos más conocidos son Larch [GH93] y OBJ [GT79].

Métodos basados en modelos matemáticos. En estos métodos se suele describir el sistema mediante la modelación de su espacio de estados. Esta descripción se realiza, principalmente, a través de teoría de conjuntos y lógica de primer orden. Las operaciones del sistema se definen mediante la especificación de su efecto sobre el espacio de estados y la declaración de predicados que relacionan diferentes estados.

Una característica común a muchos de los métodos de este tipo es que permiten definir condiciones invariantes sobre el espacio de estados, mediante predicados sobre sus componentes.

Los métodos de este tipo más conocidos son Z [ISO95], VDM [ISO93] y B [LH95].

Métodos basados en álgebra de procesos. Las álgebras de procesos permiten modelar de forma sencilla la interacción entre procesos concurrentes. Esto ha hecho que hayan tenido una gran difusión en la especificación de sistemas de comunicación (protocolos y servicios de telecomunicaciones) y de sistemas distribuidos y concurrentes.

Este tipo de métodos facilita la especificación y verificación de propiedades de seguridad (*cierto hecho negativo nunca tendrá lugar*) y de viveza (*cierto hecho positivo puede tener o tendrá lugar*) [SS97].

Los ejemplos más conocidos de este tipo de métodos son CCS [Mil80], CSP [Hoa85] y LOTOS [ISO89b].

Métodos basados en lógica. Diferentes versiones de la lógica temporal han sido usadas satisfactoriamente para especificar sistemas, especialmente sistemas de tiempo real.

Estos métodos nos permiten especificar de forma explícita las propiedades de seguridad y viveza de un sistema. Un sistema se especifica a través de un conjunto de fórmulas lógicas que definen relaciones y sucesos que ocurren en el tiempo. Además, permiten trabajar con especificaciones parciales del sistema.

Su principal inconveniente radica en su incapacidad para expresar la estructura de un sistema. Por ello, su utilización suele limitarse a las primeras fases del diseño.

Siguiendo una tendencia común a muchas disciplinas, algunos de los métodos anteriores han evolucionado incorporando operadores y conceptos de utilidad demostrada en otros métodos similares.

1.4. Fases de diseño con técnicas formales

Tal como se comentó en un principio, la fuerte base matemática que subyace en la mayoría de los métodos formales nos permite verificar la corrección de la implementación final respecto a la especificación del sistema.

Sin embargo, éste es un objetivo difícil de alcanzar cuando hablamos de sistemas de tamaño y complejidad medio-alto. La tarea de verificar (de forma completa) la corrección de una implementación respecto a una especificación formal suele precisar demasiados recursos para que pueda ser llevada a cabo en grandes sistemas, con las máquinas y algoritmos actuales.

Un procedimiento más simple y de uso frecuente en el diseño con métodos formales consiste en la construcción del sistema mediante refinamientos sucesivos. El proceso comienza con una descripción muy abstracta (una arquitectura inicial, generalmente carente de detalles que condicionen su implementación) y se construye de forma incremental, como una secuencia de refinamientos hasta alcanzar el producto final, que satisface todos los requisitos del usuario.

En cada refinamiento, se toma como entrada el diseño resultante de la etapa anterior y un objetivo (un nuevo requisito de usuario, el aumento del nivel de

detalle de la descripción, etc.). Se toma una decisión de diseño, se realiza una transformación y se obtiene el nuevo sistema que cumple el objetivo.

Veamos con más detalle cada una de las etapas.

1.4.1. Captura de requisitos

En primer lugar, es necesario capturar los requisitos del usuario, analizarlos y generar una primera descripción (generalmente bastante abstracta) del sistema, que denominaremos arquitectura inicial.

El término "requisitos de usuario" adolece de cierta ambigüedad, ya que es fuertemente dependiente del sistema, la organización que hace uso de él, el ámbito en el que se desarrolla, etc. En general, engloba descripciones de cómo se debe comportar el sistema, información sobre el dominio de aplicación, restricciones o especificaciones de propiedades, atributos del sistema, etc.

Hasta hace no mucho tiempo, era frecuente entender los requisitos de usuario como una descripción de lo que debía hacer el sistema, sin entrar en cómo lo debería hacer. Sin embargo, hoy en día, cada vez se hace más mayoritaria la opinión de que es muy difícil separar ambos conceptos [LK95, KS97], incluso poco conveniente en algunos casos. Muchas veces, la especificación formal se ve contaminada con restricciones de cara a la arquitectura por muy diferentes motivos: necesidad de aprobación por algún organismo regulador, compatibilidad con los diferentes subsistemas en los que se va a integrar, necesidad de identificar subcomponentes para poder trabajar en paralelo, reutilización de ciertos subsistemas ya desarrollados, etc.

Tradicionalmente, la captura de la arquitectura inicial se ha llevado a cabo de una manera informal, a través de una interacción entre el diseñador y los usuarios, la elaboración de documentos en un lenguaje natural, el empleo de procedimientos de prueba y error, etc. Sin embargo, en los últimos años, muchos autores mantienen que el tratamiento informal de los requisitos de usuario es el origen de muchos problemas ocurridos en el diseño de grandes sistemas [Som95]. Esto es así porque, muchas veces:

- No reflejan las necesidades reales de los usuarios que, a menudo, no saben con precisión lo que quieren.
- Son inconsistentes, incompletos o contradictorios entre sí.

- Existen diferencias entre las visiones que tienen del sistema el usuario y el diseñador.
- Suelen cambiar en el transcurso del proyecto.

Esto ha dado origen a la consideración de una nueva disciplina, la ingeniería de requisitos, que, dentro de la ingeniería del *software*, se encarga de la formalización del procedimiento para pasar de unos requisitos de usuario establecidos de manera informal a una arquitectura inicial consistente, carente de los problemas descritos.

1.4.2. Refinamientos sucesivos

Una vez que disponemos de la descripción inicial del sistema, se la somete a un procedimiento de refinamientos sucesivos hasta alcanzar una descripción fácilmente implementable, generalmente con el nivel de detalle apropiado para proceder a la implementación.

En cada refinamiento, se transforma la especificación para lograr un objetivo: introducir un nuevo requisito de usuario, especificar algún componente con más detalle, cambiar el modelo de descripción para acercarla a la máquina, etc.

Generalmente, se somete la especificación a una serie de transformaciones arquitectónicas: descomposición funcional, extensiones, reducciones, cambios de estructura, etc. Estas transformaciones suelen acompañarse de una verificación de que las descripciones de entrada y salida mantienen una cierta relación de equivalencia o de orden. La equivalencia a comprobar puede ser muy variada y de diversa justificación, siendo las más habituales la equivalencia observacional [Mil80], la equivalencia de pruebas [NH84] o la relación de implementación [BSS86].

La gran ventaja de este enfoque es que la verificación de la corrección de cada transformación es un proceso mucho más simple y abordable que la verificación completa de la implementación final respecto a la especificación inicial.

Además, una forma alternativa de comprobar esa corrección consistiría no en constatar que hacen lo mismo, sino en verificar que ambas descripciones cumplen el mismo conjunto de propiedades [PA95].

1.4.3. Implementación final

Cuando, tras los oportunos refinamientos, se disponga de una especificación del sistema que recoja satisfactoriamente todos los requisitos del usuario y cuente con el suficiente nivel de detalle, hay que proceder a su implementación en la tecnología elegida.

Este paso, por tratarse en el fondo de una traducción entre dos lenguajes conocidos, puede automatizarse en gran medida, existiendo en la actualidad herramientas que realizan la implementación de forma semiautomática entre ciertas técnicas de descripción formal y algunos lenguajes de programación [TG98, Flo95, MdM88].

1.5. Objetivos de la tesis

Los objetivos de esta tesis se centran en el campo de trabajo de la ingeniería de requisitos. Nuestra intención es la formalización de la captura de la arquitectura inicial de un sistema a partir de los requisitos de usuario.

Esta formalización debería ayudar a mitigar las deficiencias comentadas anteriormente, especialmente en lo referente al mantenimiento de la coherencia entre los requisitos de usuario y a la flexibilización del desarrollo para acomodar nuevos requisitos.

Para ello, definiremos un procedimiento metodológico que nos lleve de la especificación de un sistema a nivel informal (basada en una descripción en lenguaje natural de los requisitos de usuario) a una especificación formal que sirva de arquitectura inicial para un proceso de refinamientos sucesivos.

Este procedimiento contemplará las siguientes funcionalidades:

- Especificación y solapamiento de aquellos comportamientos parciales conocidos por el usuario.
- Definición de una biblioteca de propiedades funcionales de uso frecuente. Estos arquetipos de propiedades estarán parametrizados en ciertas variables que identificarán su funcionalidad.
- Especificación amigable de las propiedades funcionales que se deriven de los requisitos de usuario. El usuario sólo tendría que elegir la propiedad

mediante su funcionalidad y particularizar los parámetros para su sistema.

- Verificación y comprobación de coherencia de las propiedades anteriores. Se ofrecerá un algoritmo para la verificación de las propiedades anteriores en el sistema en desarrollo.
- Generación automática de sugerencias sobre modificaciones funcionales. Definiremos un algoritmo para generar sugerencias sobre modificaciones del sistema para que cumpla una determinada propiedad. El procedimiento debe garantizar la integridad del desarrollo, comprobando que las modificaciones del sistema preserven el cumplimiento de las propiedades ya verificadas.
- Traducción del resultado final a un lenguaje de especificación formal. La arquitectura inicial será entregada en forma de especificación E-LOTOS [ISO98].

Al final de la parte introductoria de esta memoria (en el capítulo 4) se realiza una descripción y justificación más detallada de los objetivos de este procedimiento.

Los resultados obtenidos serán implementados en el entorno transformacional LIRA (*Lotos Interactive Reasoning Aid*) [PA95] (descrito en el apéndice A).

1.6. Organización de la memoria

La presente memoria se estructura en cuatro partes:

Parte 1 Se compone de cuatro capítulos de introducción:

- El actual, dedicado a presentar una perspectiva general del problema y los objetivos de la tesis.
- Un capítulo dedicado a describir el estado del arte en el campo que nos ocupa.
- Un tercer capítulo que realiza una introducción a la lógica temporal como técnica de especificación formal.

- El último capítulo describe las líneas generales del procedimiento propuesto para la captura de la arquitectura inicial.

Parte 2 Se compone de cinco capítulos que describen la implementación del procedimiento expuesto en el capítulo 4 y las aportaciones realizadas en esta tesis.

- El primer capítulo introduce los sistemas de transiciones simbólicos como elemento de representación del comportamiento de un sistema. Una vez descritos de forma general se presentan algunas características particulares del modelo empleado.
- El siguiente capítulo del desarrollo describe la lógica temporal empleada y su aplicación en la formulación de una biblioteca de arquetipos o propiedades temporales que nos permite verificar las cualidades de un sistema.
- Un tercer capítulo introduce el empleo de la lógica temporal para especificar trazas de comportamiento y desarrolla el procedimiento de solapamiento de esas trazas iniciales para generar el grafo simbólico de partida y sus posteriores modificaciones.
- A continuación se describe el algoritmo propuesto para verificar las propiedades temporales de interés sobre el modelo del sistema.
- Un último capítulo describe un algoritmo para calcular, de forma automática, sugerencias sobre posibles modificaciones del sistema para lograr que verifique una propiedad.

Parte 3 Establece las conclusiones y líneas de trabajo futuro.

Parte 4 La última parte de esta memoria está compuesta por dos apéndices:

A. Una descripción de la herramienta que implementa los resultados obtenidos.
B. Un ejemplo de aplicación donde se muestra el interés de las aportaciones realizadas en esta tesis.

El último apartado de esta memoria recoge la bibliografía empleada para la elaboración de este trabajo.

Capítulo 2

Estado del arte

2.1. Introducción

A pesar del cuantioso trabajo realizado en el campo de los métodos formales en el entorno académico, la transferencia tecnológica realizada al sector industrial puede catalogarse de reducida. Como comentábamos en el capítulo anterior, algunas explicaciones pueden encontrarse en [Hal90, BH95a, BH95b].

Sin embargo, en los últimos años se ha notado un incremento significativo en el número de sistemas reales desarrollados con la ayuda de métodos formales. Este auge, que algunos autores pronostican de despegue definitivo [CW96], se debe a varias causas:

- Las técnicas de especificación, sin dejar de ser rigurosas, son más accesibles al diseñador de sistemas. Esto significa no sólo que son más sencillas de aprender, sino que están arropadas por metodologías que guían su aplicación de modo eficiente.
- Nuevos algoritmos y optimizaciones en las técnicas de verificación tradicionales han permitido un notable incremento en el tamaño de los sistemas que pueden ser objeto de verificación.
- Empieza a existir una biblioteca notable de ejemplos documentados que describen aplicaciones con éxito de métodos formales a sistemas reales, cada vez de mayor tamaño y variedad.

Especialmente importante ha sido el esfuerzo dedicado a clarificar el ámbito de aplicación de cada método. Un motivo de reticencia importante por parte de las empresas es el no saber con claridad qué método usar para su campo de trabajo y cómo emplearlo correctamente.

Sin embargo, muchos son los retos que todavía restan por resolver para lograr una aceptación de los métodos formales en el sector industrial. Por ello, muchas son también las líneas de investigación en curso, tanto en el ámbito académico como en el empresarial. Entre ellas, cabe destacar:

- El estudio de metodologías de composición de distintos formalismos, técnicas, modelos, teorías, etc.
- La descomposición de problemas computacionalmente complejos (sobre todo relacionados con la verificación) en un conjunto de subproblemas más fácilmente abordables.
- La creación de esquemas de parametrización de modelos y teorías que faciliten la reutilización del trabajo realizado.
- La integración de formalismos para tratar sistemas discretos y continuos. Muchos sistemas tienen componentes de ambas clases, que exigen razonar con matemática discreta y continua.
- La optimización de estructuras de datos y algoritmos. Uno de los problemas más importantes en el desarrollo de grandes sistemas es el gran número de estados y datos que hay que procesar. Por ello, es necesario encontrar nuevas estructuras de datos y algoritmos que optimicen el uso de los recursos.

Tan importante como los resultados teóricos que se extraigan de esas investigaciones (formalismos, teorías, métodos, etc.) son las herramientas que se desarrollen para ayudar en su aplicación. La transferencia de esa metodología al sector industrial pasa por la creación de herramientas que se adecúen a las necesidades que allí imperan [CW96]. Por ejemplo:

- Deben proporcionar un beneficio inmediato y proporcional al esfuerzo.
- Deben ser útiles en varias fases del desarrollo.

- Deben ser fácilmente integrables con las herramientas que existen actualmente.
- Deben de ser fáciles de usar y requerir poco entrenamiento.
- Por supuesto, deben ser eficientes.
- Deben estar enfocadas al análisis y a la detección de errores, más que a la demostración de la corrección.

En lo que resta de capítulo realizaremos un estudio de las técnicas y métodos que cuentan con mayor difusión en la actualidad, agrupándolas según la fase de desarrollo en la que se emplean: especificación, transformación o verificación.

2.2. Especificación

La especificación es el proceso mediante el que se describe un sistema y sus propiedades deseadas. Como comentamos en el capítulo inicial, la especificación formal utiliza un lenguaje con sintaxis y semántica formalmente definidas para este propósito.

El principal beneficio de la especificación es intangible: se adquiere un mejor conocimiento y comprensión del sistema a desarrollar. A través del proceso de especificación se descubren errores, inconsistencias, ambigüedades e incompleciones. Con el empleo de técnicas de especificación formal se consigue, además, que la especificación pueda ser analizada formalmente, es decir, puede ser probada su consistencia así como ciertas propiedades del sistema especificado.

La dificultad para definir lenguajes con la suficiente expresividad para especificar todos los aspectos del sistema ha restringido a éstos, en gran medida, a la especificación de los requisitos funcionales de un sistema (aquellos relacionados con la ordenación temporal de los eventos que constituyen su comportamiento). Sin embargo, cada vez con mayor frecuencia, se empiezan a especificar aspectos no funcionales como el rendimiento, las restricciones temporales, la fiabilidad, etc.

Las últimas tendencias persiguen la integración de diferentes lenguajes en un mismo formalismo, cada uno capaz de modelar con mayor fidelidad un aspecto distinto del sistema.

Desde el punto de vista de la adecuación de un lenguaje a la descripción de un determinado tipo de sistemas, podemos clasificar a las técnicas de descripción formal en aquellas orientadas a sistemas secuenciales y las orientadas a sistemas concurrentes.

2.2.1. Técnicas orientadas a sistemas secuenciales

Los lenguajes especializados en la especificación del comportamiento de sistemas secuenciales suelen expresar los estados del sistema mediante formalismos matemáticos tales como conjuntos, relaciones y funciones, y las transiciones entre estados mediante pre y postcondiciones [CW96].

Los ejemplos más conocidos de este tipo de lenguajes son:

- **Z** [ISO95, Spi88]. Desarrollada en la Universidad de Oxford a partir de 1980, Z es una notación formal basada en lógica de predicados de primer orden y en teoría de conjuntos. Actualmente, está en proceso de normalización por parte de ISO.

  Está orientada a la construcción de modelos de forma incremental. Emplea unas entidades denominadas *schemas* para la descripción de los estados del sistema, sus variables y las operaciones posibles en cada estado, utilizando las pre y postcondiciones (de forma implícita) para especificar las restricciones a las que se ven sometidas las citadas operaciones (invariantes).

  La utilización de la lógica de predicados permite la descripción abstracta del efecto de cada operación en un sistema y de los invariantes que deben cumplirse, así como el razonamiento sobre el comportamiento del sistema. Por otra parte, la teoría de conjuntos permite modelar el sistema mediante entidades matemáticas conocidas, tales como conjuntos, relaciones, funciones y secuencias.

  Z es una notación muy empleada (especialmente en el Reino Unido) y existen múltiples herramientas que la emplean, entre ellas **CADiZ** [JMT90], **Cogito** [ABT95] y **ZANS** [Xia95].

Z ha sido empleada, en mayor o menor medida, en el desarrollo de múltiples sistemas reales de gran tamaño. Por ejemplo:

- **CICS** [HK91]. Proyecto conjunto entre la Universidad de Oxford y los laboratorios Hursley de IBM para desarrollar un sistema de transacciones en tiempo real.
- **CIDS** [HP95]. Parte del *software* del Airbus A330/340 fue verificado mediante métodos formales, especialmente Z.

- **VDM** [ISO93, Jon90]. **VDM** (**V**ienna **D**evelopment **M**ethod) es una colección de técnicas formales para la descripción y el desarrollo de sistemas *software*. Tiene su origen en trabajos llevados a cabo a mediados de los setenta por Cliff Jones y Dines Bjørner [JB82, JB78] en un laboratorio de IBM en Viena. Utiliza una notación matemática precisa para determinar la funcionalidad que debe proporcionar el sistema.

  VDM consiste en un lenguaje de especificación denominado **VDM-SL**, (**VDM S**pecification **L**anguage), unas reglas para el refinamiento de datos, operaciones que permiten establecer vínculos entre requisitos abstractos y especificaciones detalladas, y una teoría de pruebas para verificar propiedades y decisiones de diseño.

  Las especificaciones VDM se construyen en base a modelos de un estado subyacente del sistema, el cual puede ser modificado mediante operaciones definidas a través de pre y postcondiciones.

  VDM-SL se basa en la utilización de una lógica de funciones parciales [1] y, al igual que Z, en teoría de conjuntos con secuencias y funciones. A diferencia de Z, en VDM-SL se describen de forma explícita las precondiciones y postcondiciones asociadas a una operación.

  VDM dispone de muchas herramientas para formalizar el proceso de desarrollo, entre ellas **IFAD VDML-SL Toolbox** [ELL94], **Mural** [BR91] y **SpecBox** [BFM89].

  Entre los proyectos más importantes en los que se ha utilizado VDM se encuentran:

[1] LPF. En esta lógica, el resultado de la evaluación de una expresión puede tener tres valores: Verdadero, Falso e Indefinido

  - **CDIS** [Hal96]. En 1992, la empresa Praxis empleó VDM para verificar parte del mecanismo de representación de información del sistema de control de tráfico aéreo de Londres.
  - **ACS** [MS93]. En este proyecto se especificaron mediante VDM los requisitos de seguridad de un sistema de almacenamiento de explosivos del ejército británico.

- **Larch** [GH93]. **Larch** es un proyecto de investigación desarrollado principalmente por el MIT y DEC[2]. Su objetivo es explorar métodos, lenguajes y herramientas para facilitar el uso de las especificaciones formales. Su principal característica es que describe un sistema mediante dos estilos de especificación, soportados por dos lenguajes diferentes:

  - *Lenguaje de interfaz Larch.* Diseñado para un lenguaje de programación específico. Se utiliza para especificar los interfaces entre los componentes del sistema.
  - *Lenguaje compartido Larch* (**LSL**). Similar a un lenguaje de especificación algebraico e independiente de los lenguajes de programación.

  Varias herramientas han sido desarrolladas en el proyecto, entre las que se encuentran **Larch Prover** [GG91] (un conocido demostrador de teoremas sobre especificaciones escritas en Larch) y **LCLint** [EGHT94].

- **Método B** [LH95]. Se trata de un conjunto de técnicas matemáticas apropiadas para el diseño e implementación de componentes *software*. Los sistemas se modelan como una colección de máquinas abstractas interdependientes que se describen con un enfoque orientado a objetos mediante la notación AMN (*Abstract Machine Notation*).

  El **Método B** establece guías para la estructuración de grandes diseños y la comprobación de la consistencia y corrección de estos.

  Algunas herramientas que lo emplean son **B-Toolkit** [HK91] y **Atelier-B** [Abr99].

[2] Digital Equipment Corporation

2.2.2. Técnicas orientadas a sistemas concurrentes

Los lenguajes especializados en la descripción de sistemas concurrentes especifican el comportamiento del sistema a través de secuencias de estados, árboles u órdenes parciales de eventos.

Los ejemplos más representativos de este tipo de lenguajes son:

- **CCS** [Mil80]. **CCS** (**C**alculus of **C**ommunicating **S**ystems) fue definido por Milner en 1980. CCS fue el primer álgebra para describir concurrencia y comunicación entre procesos.

  La semántica de CCS se define a través de sistemas de transiciones etiquetados y estos sistemas son los modelos naturales de las lógicas temporales y modales.

  Entre las herramientas más importantes que emplean CCS destacan el **Concurrence Workbench** [CPS93] de la Universidad de Edimburgo, **JACK** [BGL94] del IEI-CNR y **ATG FC-TOOLS** [BRRdS96] del INRIA/CMA-ENSMP

- **CSP** [Hoa85] (**C**ommunicating **S**equential **P**rocesses). Es otro álgebra para describir concurrencia y comunicación entre procesos, desarrollada en 1978 por C.A.R. Hoare. Una versión más flexible fue definida en 1985.

  CSP pemite describir un sistema a través de un conjunto de componentes (procesos) que operan independientemente y que se comunican entre ellos a través de canales definidos para tal propósito (la restricción de que los procesos componentes fueran secuenciales se eliminó entre 1978 y 1985, si bien se mantuvo el nombre del lenguaje).

  El modelo de comunicación de CSP, en el que la información se transmite solamente a través de canales nombrados explícitamente, es uno de los principios básicos del lenguaje de programación **OCCAM** [PM87].

  Las herramientas más conocidas que implementan CSP son **FDR** [Ros95], **ProBE** [Ros97] y **Casper** [Low98].

- **SDL** [ITU93, Hau96]. **SDL** (**S**pecification and **D**escription **L**anguage) es un lenguaje de descripción formal, de propósito general, normalizado por **ITU** (**I**nternational **T**elecommunication **U**nion), recomendación Z.100. Incluye elementos para definir la estructura, el comportamiento y los

datos del sistema, permitiendo especificar características temporales e indeterminismo.

Es un lenguaje de amplio espectro, tanto en su utilidad (desde la fase de especificación de requisitos hasta la de implementación) como en su ámbito de aplicación (sistemas de telecomunicación, control ferroviario, aplicaciones médicas, etc.).

Incorpora dos sistemas de representacion semánticamente equivalentes: uno gráfico (**SDL/GR**) y otro textual (**SDL/PR**). El modelo de datos está basado en Act-One [FEH83].

En SDL, los sistemas se representan estructuralmente mediante una jerarquía de bloques y procesos que intercambian señales entre ellos y con el entorno. Se trata de un modelo orientado a objetos que permite la especificación de subsistemas dentro de los bloques, lo que favorece el desarrollo por refinamientos sucesivos.

La descripción del comportamiento de los sistemas está basada en máquinas de estados finitas extendidas (que representan los procesos) que se comunican entre sí y con el entorno. Ello lo convierte en un lenguaje muy apropiado para representar sistemas basados en estímulo-respuesta.

SDL es, probablemente, la técnica formal más empleada en la industria, existiendo múltiples herramientas, tanto comerciales como de dominio público. Entre ellas, destacan:

- **QUEST** [HDMC96]. Un entorno para la descripción e integración de subsistemas, el análisis de requisitos, la visualización de comportamiento y el estudio de rendimientos.
- **ProcGen** [Flo95]. Una herramienta comercial de desarrollo de sistemas por refinamientos sucesivos que permite generar código de forma automática.

Alguno de los proyectos realizados con la ayuda de SDL son:

- **INSYDE** [PJW+95]. Proyecto Sprit para el estudio de diseños híbridos *hardware-software*. En este proyecto, Alcatel utilizó SDL para la especificación del *software* de un sistema de vídeo bajo demanda.

    - **NewCore** [Hol94]. Sistema de telecomunicaciones formalmente verificado con SDL.

- **ESTELLE** [ISO89a]. Fue desarrollado para la especificación de servicios y protocolos del modelo de referencia OSI de ISO. No obstante, es útil para la especificación de sistemas distribuidos en general.

  Está basado en máquinas de estados finitos y su sintaxis es muy similar a la del lenguaje PASCAL. En ESTELLE, un sistema se especifica mediante un conjunto de máquinas de estados finitos que se comunican entre sí mediante colas y variables globales.

  La ventaja de este tipo de lenguajes es que las especificaciones de los sistemas son muy naturales, ya que las descripciones se basan en conceptos muy conocidos y, además, proporcionan al equipo de implementación del sistema una guía de trabajo.

  Existen herramientas que traducen automáticamente ESTELLE a C++ (**PetDingo** [JL95] y **XEC** [TG98]) y a C (**EDT** [Bud92]) y que incluyen simuladores y ayudas a la depuración.

  Mediante ESTELLE se han especificado y verificado varios protocolos ISO, por ejemplo, **ROSE** [JL95].

- **LOTOS** [ISO89b, BB89]. **LOTOS** (**L**anguage **O**f **T**emporal **O**rdering **S**pecification) es un lenguaje de especificación formal normalizado por ISO con el objetivo de expresar formalmente normas de servicios y protocolos OSI de una manera clara, no ambigua, precisa, completa, e independiente de la implementación.

  En la actualidad, **LOTOS** se encuentra en proceso de revisión. Los trabajos, que se encuentran en una fase avanzada, darán lugar a una nueva versión de **LOTOS** denominada **E-LOTOS** [ISO98] (**E**nhanced-**LOTOS**).

  En la sección 2.2.3 se describe esta técnica de especificación formal con más detalle.

- **Lógica Temporal** [Got92, Lam83, MP89, Sti96].

  La introducción de la lógica modal en la especificación y verificación de sistemas *software* ha sido de gran utilidad en el proceso de razonamiento sobre programas, sobre todo por ser un vehículo muy adecuado para la expresión de las propiedades de un sistema [Pnu77].

La lógica temporal permite realizar afirmaciones sobre el comportamiento progresivo de los programas y expresar conceptos tales como ausencia de bloqueo (*deadlock*), viveza o vivacidad de sistemas (*liveness*) y exclusión mutua.

En el capítulo 3 se realiza una introducción más extensa al papel de la lógica temporal como método descriptivo y de verificación.

2.2.3. LOTOS

LOTOS (**L**anguage **O**f **T**emporal **O**rdering **S**pecification) es una técnica de especificación formal normalizada por ISO con el objetivo de expresar formalmente normas de servicios y protocolos OSI.

El mecanismo básico que emplea el lenguaje LOTOS para la especificación de un sistema consiste en la descripción de la relación temporal entre las posibles interacciones que constituyen el comportamiento externamente observable de dicho sistema. En esta definición se resalta el carácter de descripción extensional de las especificaciones LOTOS; uno de los objetivos fundamentales del lenguaje es permitir que el especificador defina cómo tiene el sistema que relacionarse con su entorno, tratando de obviar, en lo posible, la estructuración interna del sistema que permite ese comportamiento.

Como técnica formal que es, la semántica formal de LOTOS permite realizar descripciones completas, claras, consistentes y no ambiguas del sistema a especificar. El condicionar lo menos posible la arquitectura de futuras implementaciones es también una característica importante que deben perseguir las descripciones LOTOS. Una especificación debe contener las mínimas restricciones posibles a la estructura que el implementador quiera dar a la realización del sistema, y los constructores del lenguaje ciertamente favorecen la consecución de ese objetivo.

El lenguaje LOTOS está formado por dos sublenguajes que se complementan para obtener la descripción más adecuada de un sistema. La parte que trata el comportamiento y el control de las interacciones entre procesos está principalmente basada en el álgebra de procesos *Calculus of Communicating Systems (CCS)* [Mil80, Mil89] de Milner, aunque con fuertes influencias de álgebras de procesos similares como CIRCAL [Mil85] y, fundamentalmente, el *Communicating Sequential Processes (CSP)* [Hoa85] de Hoare.

Por otro lado, LOTOS tiene un segundo componente que se ocupa de la descripción de las estructuras de datos y expresiones de valor. Este sublenguaje de tratamiento de datos está basado en la teoría formal de tipos abstractos de datos y sigue, con bastante fidelidad, al lenguaje algebraico de especificación de datos *Act-One* [FEH83]. Este segundo componente es independiente del que trata de la descripción de los comportamientos y, por tanto, fácilmente sustituible por otro de similar funcionalidad. La elección de un modelo de tipos de datos abstractos frente a un conjunto de tipos de datos concretos viene a reforzar la ya mencionada búsqueda de independencia entre la especificación y sus posibles implementaciones. La no decantación por un modelo de datos concreto (sino la simple enumeración de sus características) ofrece el mínimo posible de restricciones a los implementadores.

La fuerte base matemática que sustenta la semántica formal de LOTOS hace posible el desarrollo y aplicación de métodos de validación y verificación de propiedades y equivalencias. Ello es de capital importancia para lograr el objetivo prioritario de obtener sistemas altamente fiables. A través de estos mecanismos, no sólo podemos garantizar que nuestra especificación cumple una serie de requisitos o propiedades de interés, sino que se puede llegar a determinar la corrección o conformidad de una implementación respecto a una especificación. En cualquier caso, el empleo de esta clase de métodos nos ayudará a descubrir, más rápidamente, la existencia de posibles errores o disfuncionalidades en la especificación del sistema.

2.2.3.1. E-LOTOS

Actualmente, el lenguaje de especificación formal LOTOS se encuentra en proceso de revisión. Tras cinco años de vida del estándar, en 1993 **ISO** decidió crear un nuevo grupo de trabajo para refinar y enriquecer el lenguaje, conociéndose al futuro estándar como E-LOTOS (Enhanced LOTOS) [ISO98].

Las modificaciones que se pretenden introducir en el lenguaje son, generalmente, fruto de las sugerencias de los usuarios de LOTOS. Los objetivos generales perseguidos son:

- Mejorar, en lo posible, aquellos aspectos que conforman la facilidad de uso del lenguaje. Por ejemplo, facilitando la construcción de estructuras de datos modulares y coherentes, estableciendo canales para la integración de descripciones realizadas en otros formalismos, etc.

- Acercar su sintaxis a la de los lenguajes de programación más usuales (introduciendo bucles, excepciones, etc.).
- Eliminar algunas deficiencias o limitaciones que se han observado en su capacidad expresiva o de razonamiento simbólico, facilitando las sincronizaciones complejas entre múltiples procesos, la evaluación de expresiones de valor, etc.

Las propuestas de modificación del lenguaje [ISO98] afectan tanto a la parte de datos como a la del comportamiento, existiendo un acuerdo general para acercar ambos componentes, ampliando su simetría en lo que respecta a los constructores del lenguaje. En cuanto al modelo de datos, se pretende sustituir el actual por otro más amigable y familiar para los usuarios, que garantice la consistencia de los tipos de datos empleados y facilite la realización práctica de herramientas de apoyo al proceso de desarrollo de una especificación. Este nuevo lenguaje estaría compuesto de dos sublenguajes:

- Uno de carácter algebraico, para la especificación de ecuaciones, al estilo de ACT-ONE, pero incluyendo nuevos elementos que tenderían a simplificar la especificación de tipos de datos abstractos (por ejemplo, una organización modular con interfaces de acceso). Este sublenguaje cubriría los objetivos del actual, en cuanto a la abstracción de sus descripciones y la capacidad de razonamiento simbólico sobre los tipos de datos así construidos.
- Otro de carácter funcional, más orientado a la implementación directa de la especificación, a la ejecución simbólica, simulación y, en general, a la manipulación práctica de la especificación mediante herramientas de apoyo. Este sublenguaje, basado en lenguajes funcionales existentes como SML [CMP93] o MIRANDA [CMP95], proporcionaría unos tipos básicos, permitiendo al especificador nuevas definiciones, extensiones de los ya existentes o incluso importar tipos de datos definidos en otros lenguajes.

En lo que respecta a la parte de comportamiento, se pretende introducir varios constructores y mecanismos que mejoren la potencia expresiva de LOTOS y ayuden a construir especificaciones más claras y modulares. Entre ellos, destacan el levantamiento y captura de excepciones, el reconocimiento de patrones

(*pattern matching*), la composición en paralelo generalizada, contructores que permitan la iteración, la composición secuencial de comportamientos y mecanismos para especificar características no funcionales como una extensión de tiempo real para describir el paso del tiempo o la existencia de prioridades. Todo ello, tratando de mantener en lo posible la compatibilidad con el estándar actual.

2.3. Simulación y transformación

Como ya comentamos en el capítulo de introducción, la base matemática que subyace en las técnicas de especificación formal favorece el desarrollo de sistemas por un procedimiento transformacional.

Por metodología transformacional se entiende la construcción del sistema a través de diversas transformaciones, mediante las cuales se va refinando la descripción inicial y añadiendo información hasta llegar a una etapa con el suficiente nivel de detalle para ser directamente traducida a un lenguaje de implementación convencional. En muchos casos, este último paso es semiautomático, ya que existen herramientas que, con mayor o menor ayuda del diseñador, son capaces de implementar una especificación en alguno de los lenguajes de programación más comunes [TG98, Flo95, MdM88].

A lo largo del desarrollo transformacional descrito es necesario someter continuamente la especificación a un conjunto de pruebas y verificaciones que nos garanticen que el proceso es correcto y que las propiedades deseadas se mantienen. Entre otras técnicas, es típico hacer uso de la simulación, la ejecución simbólica, el prototipado o la verificación matemática.

La simulación o ejecución de especificaciones es un procedimiento bastante habitual, que permite animar y observar el comportamiento del sistema en su fase actual. Existen múltiples variantes, desde la ejecución paso a paso guiada por el usuario, hasta la animación con elección automática del próximo evento (basada en criterios probabilísticos, equitativos, estáticos, dinámicos, etc.).

Uno de los principales problemas de esta aproximación es la resolución de expresiones de datos con variables libres. En muchos lenguajes, la evaluación de estas expresiones es fundamental para decidir si un evento es posible o no. Los enfoques actuales para afrontar este problema se basan en reescritura [Klo92] y *narrowing* [Klo87].

Existe una gran variedad de herramientas que permiten simular la evolución simbólica de una especificación, entre las que se encuentran **LOLA** [QPF89], **ISLA** [GL93], **SMILE** [BvLV94], **HIPPO** [vE88] y **CADP** [FGM+94] para LOTOS; **JACK** [BGL94] para CCS; **PetDingo** [JL95] para ESTELLE y **ObjectGEODE** [Leb97] para SDL.

2.3.1. Transformaciones

Durante el proceso de desarrollo puede ser interesante someter a la especificación a múltiples transformaciones, con objetivos muy distintos:

- Para verificar sus propiedades mediante demostradores de teoremas o *model checking* (sección 2.4). En este caso, conviene someter a la especificación a procesos de expansión, previo a conseguir su sistema de transiciones equivalente (sección 2.3.2), y de parametrización, para reducir el número de estados de ese sistema.

- Para identificar componentes con vistas a estructurar o crear prototipos. Para ello, se somete a la especificación a transformaciones estructurales (de caja blanca a caja blanca) y a reducciones y extensiones funcionales.

- Para refinar la especificación. Se suelen realizar descomposiciones funcionales (de caja negra a caja blanca), que sustituyen un componente por otro equivalente descrito con mayor nivel de detalle.

- Para realizar una implementación. Hay que transformar la especificación para que pueda ser implementada en una máquina. Hay que eliminar todos los constructores que no tengan correspondencia directa en el lenguaje de especificación elegido.

Al realizar estas transformaciones, es necesario asegurarse de que la especificación inicial y la final guardan una cierta relación. Estas relaciones suelen expresarse en términos de equivalencias y de órdenes, que se verifican automáticamente en cada paso o transformación. Las más usuales son las bisimulaciones [Mil80], la equivalencia observacional [NH84], la equivalencia de pruebas [Mil80] y la relación de implementación [BSS86].

Las herramientas que se encuadran dentro de este segmento suelen conformar verdaderos entornos transformacionales, implementando múltiples funcionalidades de transformación y verificación. Entre ellas destacan **ALDEBARAN/CAESAR** [FGM+94], **JACK** [BGL94], **LOLA/TOPO** [QPF89, MdM88], **AUTO/FC2TOOLS** [BRRdS96], el **Concurrence Workbench** [CPS93], etc.

2.3.2. Sistemas de Transiciones Etiquetadas

Como mencionamos en la sección anterior, una transformación muy habitual consiste en la generación de una estructura matemática que nos permita representar de forma apropiada los estados del sistema.

Un Sistema de Transiciones Etiquetadas (STE) se define como una cuádrupla ($\mathcal{S}$, s_0, $\mathcal{L}$, $\mathcal{T}$) con el siguiente significado:

- $\mathcal{S}$ es el conjunto de estados o diferentes situaciones en los que puede encontrarse el sistema.

- s_0 es el estado inicial del sistema. Por tanto, $s_0 \in \mathcal{S}$.

- $\mathcal{L}$=$(a, b, c, ...)$ es el conjunto de eventos que pueden provocar que el sistema cambie de estado. Pueden ser eventos producidos por el propio sistema, eventos generados por el entorno, el simple paso del tiempo, etc.

- $\mathcal{T}$ es el conjunto de transiciones del sistema. Cada transición estará compuesta por el estado origen, el estado destino y el evento que etiqueta o provoca esa transición. Gráficamente, suele representarse por $s_i \xrightarrow{e} s_j$.

Un Sistema de Transiciones Etiquetadas, por tanto, nos permite representar de forma compacta la evolución de un sistema a lo largo del tiempo. En él quedan reflejadas, de forma precisa, las distintas etapas que puede atravesar el sistema y los posibles eventos que pueden provocar, en cada momento, las transiciones entre esas etapas.

Los Sistemas de Transiciones Etiquetadas son la forma clásica de expresar la semántica temporal de los lenguajes de procesos y una representación del

sistema muy apropiada para proceder al estudio de equivalencias entre sistemas y a la verificación de propiedades mediante *model checking*.

Generalmente, un *STE* suele representarse gráficamente por un árbol, donde los nodos son los estados y las transiciones entre nodos van etiquetadas con el evento que las produce. Por ejemplo, en la figura 2.1 se representa un árbol equivalente a un *STE* con:

- $\mathcal{S}=\{s_0, s_1, s_2, s_3\}$
- $\mathcal{L}=\{tx, ack, nack, tout, query\}$
- $\mathcal{T} = \{\mathcal{T}_{tx}, \mathcal{T}_{ack}, \mathcal{T}_{nack}, \mathcal{T}_{tout}, \mathcal{T}_{query}\}$
    - $\mathcal{T}_{tx}=\{s_0 \xrightarrow{tx} s_1,\ s_2 \xrightarrow{tx} s_1\}$
    - $\mathcal{T}_{ack}=\{s_1 \xrightarrow{ack} s_0\}$
    - $\mathcal{T}_{nack}=\{s_1 \xrightarrow{nack} s_2\}$
    - $\mathcal{T}_{tout}=\{s_1 \xrightarrow{tout} s_3\}$
    - $\mathcal{T}_{query}=\{s_3 \xrightarrow{query} s_1\}$

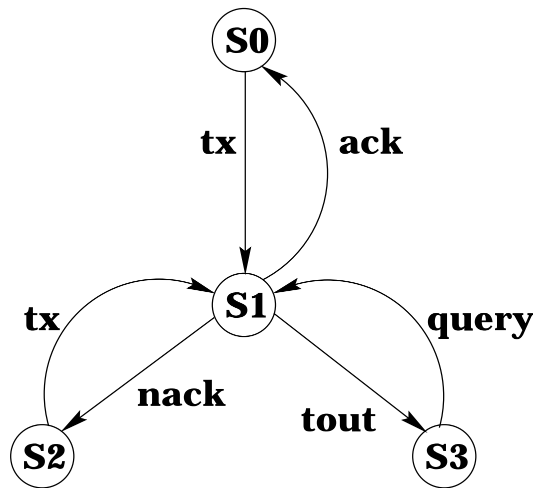

Figura 2.1: Representación gráfica de un STE

2.4. Verificación

La verificación formal de un sistema es una de las mayores ventajas que proporciona el desarrollo con métodos formales. A través de la verificación podemos demostrar matemáticamente que un sistema cumple las propiedades deseadas, que los refinamientos de la especificación inicial son correctos o que la implementación final alcanza los objetivos propuestos.

El proceso de verificación implica la comparación de dos objetos formales: dos especificaciones del sistema, una especificación y una implementación, una especificación y una propiedad, etc. En cualquier caso, el paso inicial consiste en formalizar los objetos a comparar, es decir, crear modelos matemáticos que describan las propiedades de los objetos reales a los cuales representan.

Son esos modelos (y no el sistema real) los que van a ser objeto del proceso de verificación. Por tanto, la calidad de los resultados está condicionada por el grado de fidelidad con que esos modelos reemplazan a los sistemas bajo estudio.

En la actualidad, dos son los mecanismos más difundidos para la verificación formal de sistemas: los demostradores de teoremas y el *model checking*.

2.4.1. Demostradores de teoremas

Es el enfoque más antiguo en la verificación formal de sistemas. Consiste en emplear el razonamiento deductivo para extraer conclusiones a partir de axiomas o premisas cuya veracidad conocemos desde un principio o ha sido probada anteriormente.

La técnica que emplean los demostradores de teoremas se basa en expresar tanto el sistema como sus propiedades deseadas en alguna lógica matemática. Esta lógica forma parte de un sistema formal donde se define un conjunto de axiomas y de reglas de inferencia.

Una propiedad se define como un teorema, cuya veracidad es el objetivo a demostrar. Para ello se empleará un razonamiento deductivo a partir de los axiomas y la aplicación de las reglas de inferencia, las definiciones, lemas o teoremas intermedios derivados a lo largo del proceso de demostración.

La rigurosidad matemática que subyace en este mecanismo garantiza la cor-

rección de los resultados (otro problema distinto es la dificultad en la obtención de resultados en sistemas de tamaño real). Por ello, los demostradores de teoremas se están usando cada vez más en la verificación de sistemas con fuertes requisitos de seguridad, así como en diseños *hardware* y *software* [CW96].

El grado de automatización es un criterio importante a la hora de realizar clasificaciones en este campo. Los demostradores de teoremas actuales varían en un amplio rango de automatización, desde los muy automatizados (generalmente de propósito general) hasta los sistemas muy interactivos (generalmente con características específicas para un determinado tipo de sistemas). Estos últimos son más adecuados para el desarrollo sistemático con métodos formales, aunque la necesidad de intervención del usuario los hace más lentos y propensos a errores.

En general, las clasificaciones existentes en la literatura son ciertamente difusas, atendiendo a criterios como el citado acerca del grado de automatización o el uso previsto para la herramienta. Incluso, dependiendo de la ponderación de los criterios realizada, algunas herramientas son encuadradas en diferentes categorías [Sha94]. Según el grado de automatización [CW96] podemos encontrar:

Herramientas de deducción automática guiadas por el usuario Este tipo de herramientas tiene como característica común el que su operación se guía por una secuencia de lemas y definiciones, si bien cada teorema se demuestra automáticamente mediante heurísticos sobre inducción, lemas, reescritura y simplificación.

Ejemplos de este tipo de sistemas son:

- **Nqthm** [BM79]. Es el demostrador de teoremas de la lógica de Boyer-Moore. Está basado en inducción y reescritura. La lógica subyacente es muy parecida a la que forma la base del lenguaje LISP: de primer orden, sin tipos y sin cuantificadores.

  Fue utilizado en la solución de algunos problemas de verificación de grandes sistemas *hardware* y *software* [BY96], así como en la verificación de teoremas matemáticos [Sha94].

- **ACL2** [KM95]. ACL2 es un demostrador de teoremas formulados en la lógica ACL2. Su nombre deriva de *A Computational Logic for Applicative Common Lisp*. ACL2 es el sucesor del demostrador de

teoremas **Nqthm** y, como indica su nombre, la lógica que emplea es un subconjunto de la de LISP, lenguaje en el que está programada la herramienta casi en su totalidad.

ACL2 ha sido empleado con éxito en numerosos desarrollos de sistemas *hardware* [KM95, BKM96].

- Otros ejemplos de este tipo de herramientas son: **LP** [GG88], **Eves** [CKM+88], **REVE** [Les83] y **RRL** [KM87].

Proof checkers. Han sido usados para formalizar y verificar problemas complejos de matemáticas, así como para la verificación tanto de *hardware* como de *software*. Los ejemplos más ilustrativos de este tipo de demostradores de teoremas son:

- **LCF** [GMW79]. Uno de los primeros demostradores de teoremas. Los axiomas son teoremas simples y las reglas funciones que transforman unos teoremas en otros. Ha sido empleado para la verificación de propiedades y la comprobación de corrección de algún algoritmo de unificación [Pau84].

  De él derivan **HOL**, **Nuprl** e **Isabelle** entre otros.

- **HOL** [GM93]. Es un sistema demostrador de teoremas en Higher Order Logic, diseñado para construir especificaciones y verificaciones formales de sistemas. Ha sido aplicado en entornos académicos e industriales para el desarrollo de *hardware*, sistemas de tiempo real, verificación de compiladores, etc.

  Existe un numeroso grupo de herramientas basadas en **HOL**, entre ellas **ProofPower** [KAW96] y **HOL-Z** [KTW96].

- **Isabelle** [Pau94]. **Isabelle** es un demostrador de teoremas genérico, basado en el λ-*calculus* [Tho91]. **Isabelle** posibilita la introducción de nuevas lógicas especificando su sintaxis y reglas de inferencia. Proporciona un alto grado de automatización.

  Isabelle ha sido empleado con éxito en el estudio de múltiples problemas matemáticos, entre ellos la verificación de protocolos criptográficos [Pau98].

- Otros ejemplos de este tipo de demostradores son: **Coq** [CCF+85], **LEGO** [LP92], y **Nuprl** [Cea86].

Sistemas híbridos. Su característica principal es que combinan varias técnicas de verificación. Entre los más representativos cabe citar:

- **Analytica** [CZ93]. Combina la demostración de teoremas con el sistema de álgebra simbólica *Mathematica* y ha sido empleado con éxito para resolver problemas complejos de teoría de números.
- **PVS** [ORS92] y **STeP** [Bea96] combinan potentes procedimientos de decisión con *model checking* (sección 2.4.2) para realizar demostraciones interactivas. **PVS** ha sido usado con éxito para verificar diseños *hardware* [MS95], así como sistemas reactivos, de tiempo real y tolerantes a fallos.

2.4.2. Model checking

Los demostradores de teoremas son una herramienta muy potente que nos permite verificar un sistema con un alto grado de fiabilidad. Sin embargo, su aplicación a sistemas reales presenta serios problemas:

- El tamaño de los sistemas susceptibles de ser verificados de tal forma es bastante limitado.
- Requiere usuarios con gran capacidad y preparación.
- Requiere la inversión de mucho tiempo, lo que retrasa el proceso de desarrollo.
- A veces se pierde tiempo en demostrar teoremas cuya veracidad es obvia o resulta intrascendente.
- No ofrecen contraejemplos en caso de fallar la verificación. Esto dificulta la identificación de los errores.

Por todo ello, a mediados de los setenta se comenzó a trabajar en otro tipo de técnicas que Clarke bautizó como *model checking* [CE81].

El *model checking* es un procedimiento que se basa en la construcción de un modelo finito del sistema y la comprobación de que una propiedad dada se satisface en dicho modelo.

Básicamente, la demostración se ejecuta mediante una búsqueda exhaustiva en el espacio de estados del modelo del sistema, por lo que para que termine, este espacio de estados deberá ser necesariamente finito. Desde un punto de vista técnico, los trabajos de investigación en *model checking* se dirigen a la búsqueda de algoritmos y estructuras de datos más eficientes, que permitan el manejo de espacios de estados muy grandes, propios de los sistemas reales.

Inicialmente, el *model checking* se usó en la verificación de sistemas *hardware* y de protocolos de comunicación. Actualmente, comienza a aplicarse esta técnica al análisis y verificación de especificaciones de sistemas *software*.

En la actualidad, existen dos enfoques principales para realizar *model checking*:

- El *model checking* temporal es una técnica desarrollada en la década de los ochenta de forma independiente por Clarke y Emerson [CE81] y por Queille y Sifakis [QS82]. Esta técnica utiliza la lógica temporal para expresar las propiedades que queremos verificar en los sistemas. Estos, por su parte, se modelan como sistemas de transición de estados finitos (sección 2.3.2).

  El trabajo de Clarke y Emerson condujo a la herramienta **EMC** [CES86] (y posteriormente a **SMV** [McM93]), mientras que Queille y Sifakis desarrollaron **CÆSAR** [QS82]. Ambas herramientas utilizaban la lógica CTL [CE81].

  En la seccion 3.3 profundizaremos en el empleo de esta técnica.

- El segundo enfoque para abordar el *model checking* consiste en modelar ambos objetos, sistema y propiedad, mediante autómatas. Luego se comparan, sometiéndolos a pruebas, para comprobar si el comportamiento del sistema es o no el definido por la propiedad.

  En la literatura se pueden encontrar varios tipos de pruebas: Inclusión de lenguajes [HK90, Kur94b], relaciones de orden [CPS93, Ros94] y equivalencia observacional [CPS93, FGK$^+$96, RS90].

  Entre las herramientas que siguen este planteamiento destacan **SPIN** [Hol91] y **Mur**ϕ [Dil96].

Las principales ventajas del *model checking* son:

- La posibilidad de automatizar totalmente las demostraciones.
- Su rapidez, que lo hace capaz de dar un resultado en poco tiempo.
- Su capacidad para trabajar con especificaciones parciales y proporcionar información útil, aun cuando el sistema no se encuentre totalmente especificado.
- La posibilidad de producir contraejemplos, que ayudan a localizar los errores.

Por contra, su principal desventaja es el problema de la explosión de estados. El tamaño de los sistemas reales suele ser tal que su espacio de estados se vuelve inmanejable y, por tanto, la búsqueda exhaustiva en él es inviable.

Por ello, una de las líneas de trabajo más importantes en este campo es la búsqueda de técnicas para reducir el espacio de estados del sistema conservando sus propiedades (sección 2.4.3).

En la actualidad, el *model checking* es una técnica de demostración muy potente, de ahí que empiece a usarse en la industria para la verificación de nuevos diseños. Las herramientas disponibles manejan entre 100 y 200 variables de estado y se han realizado demostraciones con sistemas de 100^{120} estados alcanzables [BCL+94], y mediante el empleo de técnicas de abstracción apropiadas se pueden realizar demostraciones sobre sistemas de prácticamente un número ilimitado de estados [CGL92].

En la literatura comienzan a ser frecuentes las aplicaciones del *model checking* al desarrollo y verificación de proyectos de diversa índole. Por ejemplo:

- Protocolos de coherencia de caché como el **IEEE Futurebus+** [CGH+93] (verificado con SMV) o el **IEEE SCI** [DDHY92] (con Murϕ).
- Arquitecturas multiprocesador como **PowerScale** [CGM+96] (verificado con CÆSAR).
- Sistemas de control estructural para proteger edificios frente a terremotos [ECB94] (verificado con el Concurrency Workbench).

2.4.3. Líneas de trabajo

Las principales líneas de investigación que se están desarrollando para solucionar o reducir el problema de la explosión de estados se encaminan en las siguientes direcciones:

- Representar los estados del sistema de forma simbólica (*model checking simbólico* [McM93]) mediante fórmulas que representan un conjunto de estados, aquellos en los que la fórmula es cierta. En esta línea cabe citar los trabajos de McMillan y Bryant [Bry86] con los Diagramas de Decisión Binaria Ordenados (OBDD), con los que consiguieron aumentar de forma considerable el tamaño del espacio de estados que se podía verificar.
- Otra línea de trabajo se dirige a utilizar la información de orden parcial disponible [Pel96] y la minimización semántica [ECB96] para eliminar estados innecesarios del modelo del sistema.
- Una línea de trabajo, denominada *reducción homomórfica*, combina la abstracción de partes no esenciales del sistema con la explotación de la estructura y las simetrías de los sistemas [Kur94a, Kur94b].

Una de las líneas de investigación más interesantes y que, a priori, parece más prometedora es la de buscar métodos para la integración de *model checking* y demostradores de teoremas [KL93]. En este campo existen varias posibilidades:

- Emplear *model checking* como un procedimiento de decisión en un entorno de demostración basado en la deducción. Este camino se ha tomado como base en el desarrollo de herramientas como PVS y STeP.
- Usar la deducción para obtener una abstracción de estado finito de una implementación, que luego será verificada mediante *model checking*.
- Puede usarse la deducción para verificar las premisas generadas mediante la composición de componentes que han sido verificados por separado por medio de *model checking*.

Capítulo 3

La lógica temporal

3.1. Introducción

En el capítulo introductorio vimos que el desarrollo de un sistema por refinamientos sucesivos es un procedimiento muy adecuado cuando se trabaja con métodos formales. La principal ventaja en la que descansaba nuestra argumentación es que resulta menos costoso (en recursos materiales, tiempo y esfuerzo conceptual) la verificación de las sucesivas pequeñas transformaciones a las que sometemos la especificación inicial que la verificación del sistema final respecto a la susodicha especificación.

Ahora bien, la utilidad del procedimiento exige, ineludiblemente, la disponibilidad de mecanismos de validación y verificación que nos garanticen que cada una de las transformaciones es correcta. Como relatábamos en el capítulo anterior, existen varios formalismos diseñados a tal efecto y, en alguno de ellos, la lógica temporal juega un papel muy importante.

En general, dos son las preguntas que podemos plantearnos a la hora de juzgar el resultado de una transformación:

- La nueva especificación, ¿cumple la propiedad que perseguíamos con la transformación?. Se trata de un problema clásico de verificación de las propiedades de un sistema.
- El sistema refinado, ¿es equivalente al inicial?. En este caso, la respuesta pasa, en primer lugar, por definir el tipo de equivalencia que queremos

mantener en la transformación.

Como veremos en este capítulo, si enfocamos el trabajo de verificación con la ayuda de la lógica temporal, las respuestas a estas dos preguntas están bastante relacionadas entre sí.

3.2. La lógica temporal

La lógica temporal es una disciplina matemática que nos permite razonar sobre el transcurso del tiempo, ya sea de forma cuantitativa o cualitativa. En este último aspecto, nos permite establecer restricciones sobre la ordenación que ciertos eventos pueden o deben tener a lo largo de un periodo de tiempo. Mediante la lógica temporal podemos relacionar sucesos a través de conectivas tales como *antes, después, a la vez, a continuación*, etc.

Para ello, las distintas lógicas, además de las conectivas *booleanas* tradicionales, suelen ofrecer operadores temporales que relacionan predicados lógicos. Por ejemplo:

- $\Box p$: La fórmula lógica p es cierta en todos los estados futuros.
- $\Diamond p$: La fórmula p será cierta en el futuro, en alguna de las posibles evoluciones del sistema.
- $p\mathcal{U}q$: La fórmula p es cierta en todos los estados futuros hasta que la fórmula q sea cierta.

Se pueden encontrar muchas lógicas en la literatura, entre las que cabe destacar la lógica temporal de Pnueli (TL) [Pnu81], la lógica proposicional dinámica (PDL) [FL79], la lógica modal de Hennessy-Milner [HM80] y su extensión (para el tratamiento de recursividad) el *μ-calculus* [Koz83], la lógica CTL [CE81] y su extension (para expresar equidad) CTL* [ES89], etc.

En la sección 1.1.1 definíamos un sistema reactivo como aquel cuyo comportamiento no consiste en una serie de pasos ordenados que finalizan produciendo un resultado, sino que se perpetúa en una constante interacción con su entorno,

sin un fin previsible [Pnu85]. La importancia de este tipo de sistemas es creciente, al gobernar procesos cuyo control debe ser continuo: red de semáforos, procesos industriales, tráfico aéreo, etc.

En este tipo de sistemas, por tanto, resulta difícil e inadecuado la especificación de su comportamiento mediante la relación entre entradas y salidas, siendo preferible la descripción de su interacción con el entorno. El resultado es el conjunto de trazas que detallan el comportamiento que el sistema puede llevar a cabo. Cada una de esas trazas recogerá una secuencia de ejecución válida, es decir, una posible ordenación de los eventos que pueden tener lugar al interactuar con su entorno.

Por tal motivo, la lógica temporal en general es un formalismo muy adecuado para el estudio de este tipo de sistemas [Got92, Pnu79]. Las distintas versiones de la lógica temporal nos permiten (por supuesto, con distinto grado de adecuación):

- Describir la evolución de un sistema a través de la ordenación temporal de los eventos que determinan su interacción con el entorno. Mediante lógica temporal se realiza la especificación del comportamiento de forma implícita, al restringir las ordenaciones temporales posibles de sus eventos[1].

- Describir ciertas propiedades cuya verificación en un sistema consideramos interesante. Esto se suele realizar, también, mediante restricciones sobre los ordenamientos posibles.

- Razonar sobre el cumplimiento de una propiedad en un sistema.

Además, una de las ventajas más importantes de la lógica temporal es que nos permite trabajar de forma directa con especificaciones parciales [Pnu85]. La relación de satisfacción que se define en el campo de la lógica nos permite proceder a la verificación de propiedades en una parte del sistema, sin necesidad de tratar con la especificación completa.

[1] Esta descripción se realiza de forma explícita en otras técnicas de descripción formal de uso frecuente: CCS [Mil80], CSP [Hoa85], LOTOS [ISO89b], etc.

3.2.1. Semántica temporal

El estudio de la lógica temporal en lo referente a sus aplicaciones a la descripción y verificación de sistemas *software* data de los años sesenta, cuando primero Floyd [Flo67] y luego Hoare [Hoa69] procedieron a definir la semántica de un programa $\mathbf{p}$ asociada a una lógica $\mathcal{L}$ como el conjunto de fórmulas de dicha lógica que satisface, $\Phi_{\mathcal{L}}(\mathbf{p})$.

$$\Phi_{\mathcal{L}}(\mathbf{p}) = \{\phi \in \mathcal{L} \ / \ \mathbf{p} \models \phi\}$$

Lo primero que se desprende de esta definición es la necesidad de formalizar una *relación de satisfacción* ($\models$) entre un programa o especificación $\mathbf{p}$ y una fórmula lógica ϕ. Trataremos este punto en la sección 3.3.

En segundo lugar, podemos comprobar que de la relación de satisfacción definida se desprende directamente una relación de equivalencia entre programas ($\equiv_{\mathcal{L}}$): dos programas serán equivalentes si tienen las mismas propiedades.

$$\mathbf{p} \equiv_{\mathcal{L}} \mathbf{q} \Longleftrightarrow \Phi_{\mathcal{L}}(\mathbf{p}) = \Phi_{\mathcal{L}}(\mathbf{q})$$

La validez de esta equivalencia es completamente dependiente del lenguaje de procesos $\mathcal{P}$ al que pertenece $\mathbf{p}$ y de la lógica $\mathcal{L}$. Para afirmar que esta equivalencia temporal es suficiente para validar los refinamientos de un sistema, es necesario estudiar su relación con las equivalencias clásicas derivadas a partir de la comparación de comportamientos (bisimulación, equivalencia observacional, equivalencia de pruebas, etc.).

Otro enfoque muy interesante y más reciente consiste en trasladar todos los elementos involucrados en el proceso de verificación al dominio de la lógica temporal [PA95].

En este caso, el primer paso consistiría en definir una semántica temporal para el lenguaje de procesos $\mathcal{P}$ en el que está expresado el sistema. De esta forma, cada sistema $\mathbf{p}$ sería traducido a una fórmula lógica, $\mathcal{L}(\mathbf{p})$, que recogería su comportamiento. En este marco, un programa verificaría una propiedad si la fórmula lógica del programa implica la propiedad.

$$\forall \mathbf{p} \in \mathcal{P}, \forall \phi \in \mathcal{L}, \qquad \mathbf{p} \models \phi \iff \mathcal{L}(\mathbf{p}) \Rightarrow \phi$$

Para que este formalismo resulte útil, es necesario que la lógica temporal elegida satisfaga ciertas propiedades en relación con el lenguaje de procesos $\mathcal{P}$ en el que se describe inicialmente el sistema: adecuación, expresividad, total abstracción y composicionalidad [PA95, Pnu85].

3.2.2. Aplicación a sistemas reales

Aunque cada vez sean más y de mayor tamaño los sistemas verificados formalmente con este tipo de procedimientos [BBRTL90, BS93, GCR94], todavía son muy minoritarios, estando limitada su aplicación real a sistemas de tamaño medio-pequeño. El gran coste en recursos que implica este tipo de técnicas hace que generalmente se utilicen para verificar partes muy concretas del sistema en desarrollo, generalmente aquellas que tienen unas restricciones más fuertes de seguridad y fiabilidad.

Conforme el estado de la técnica nos permite abordar el estudio de sistemas cada vez más variados, también se incrementa el rango de características reales que deseamos modelar. Al igual que ocurre en otros ámbitos donde se utilizan las matemáticas para modelar problemas reales, el diferente nivel de abstracción con el que queremos estudiar problemas distintos hace que ningún modelo sea óptimo para su aplicación a cualquier tipo de sistema.

Por contra, parece que una jerarquía de modelos es una solución más apropiada [MP93]. Cada modelo de la jerarquía estudiaría un sistema con un nivel de abstracción determinado, adecuado para nuestros propósitos. Para ello nos proporcionaría los constructores necesarios para trabajar con el nivel de detalle buscado, sin entorpecer nuestro trabajo con sintaxis, semántica y procedimientos innecesarios en ese nivel de abstracción.

Manna y Pnueli [MP93] proponen una jerarquía de esas características, compuesta por los modelos:

Reactivo. Se trata del esquema clásico, que modela los aspectos cualitativos de la ordenación temporal del comportamiento de un sistema [MP92a].

De tiempo real. Trata de profundizar en el modelo anterior, definiendo métricas para el tiempo que permitan abordar aspectos cuantitativos [MP92b].

Híbrido. Su objetivo es incrementar el nivel de detalle mediante la inclusión de componentes continuos dentro del modelo [MP92c]. Se trata de mode-

los adecuados para lograr un nivel de abstracción mínimo, muy cercano al sistema real que queremos estudiar. Los trabajos en este tipo de modelos están todavía en su fase inicial.

Cada uno de los anteriores modelos implica diferentes formalismos para describir los sistemas, lógicas temporales con distinta expresividad, conjuntos de propiedades temporales cada vez más complejas, etc.

En la literatura existen múltiples lógicas con muy distinta sintaxis y expresividad. Aunque no hay un claro consenso en cuanto a qué lógica es mejor, sí lo hay en establecer ciertos criterios de selección, sobre todo relacionados con las características del sistema bajo estudio. Uno de esos criterios tiene que ver con la semántica que se atribuye al programa sobre el que se quieren realizar tareas de verificación.

Por una parte se le puede asignar una semántica lineal, donde cada programa se estudia mediante su conjunto de trazas o secuencias de ejecución posibles. En este caso, la concurrencia se modela mediante el intercalado de las acciones de los sistemas envueltos en la concurrencia. Mediante este tipo de descripciones no quedan reflejados los puntos de decisión en la evolución del sistema

Por otra parte, se le puede asignar una semántica ramificada, modelando el sistema como un árbol de estados unidos por transiciones provocadas por los eventos del sistema. En este caso, sí quedan reflejados los instantes en que el sistema toma decisiones.

De la anterior caracterización se desprende otra aplicable a las distintas lógicas, en función de su adecuación para modelar uno u otro grupo:

Lógicas de tiempo lineal. Llevan a cabo una interpretación lineal del tiempo y del futuro [EH83]. El tiempo es una secuencia ordenada de instantes discretos. En un instante determinado, hay un sólo futuro posible.

Lógicas de tiempo ramificado. Interpretan el tiempo como una estructura en árbol [ES88]. En cada momento hay varios futuros alternativos. Un proceso se modela como una estructura formada por múltiples caminos.

No hay acuerdo en qué tipo de lógica es mejor. En general, dependerá del tipo de problema que se quiera tratar. Lamport [Lam80] afirma que estos dos

tipos de lógica no se pueden comparar, aunque hay una coincidencia en que la lógica de tiempo ramificado es más expresiva [Lam80, Pnu85].

3.3. Verificación con lógica

En la sección anterior vimos que el eje central sobre el que pivota el mecanismo de verificación de los refinamientos de un sistema es la relación de satisfacción ($\models$), que nos permite decidir si un programa o la especificación de un sistema cumple una propiedad.

Esta relación de satisfacción depende completamente de cual sea el procedimiento elegido para la verificación. En el capítulo anterior vimos que las dos aproximaciones más habituales a la verificación formal son los demostradores de teoremas y el *model checking*.

El *model checking temporal* es un mecanismo ampliamente utilizado, tanto en la industria como en el entorno académico, y en el que la lógica temporal se emplea para especificar la propiedad a verificar y para conducir el proceso de verificación. En esta metodología participan varios elementos [MP89]:

- Un modelo computacional, que provee un marco de representación común para los distintos lenguajes de procesos o técnicas de especificación formal que utilizamos para describir un sistema. Uno de los modelos más empleados en el campo de la lógica temporal son los sistemas de transiciones equitativos 2.3.2, que modelan el comportamiento del sistema mediante la especificación de los distintos estados por los que puede pasar y las transiciones (provocadas por ciertos eventos) que conducen de estado a estado.

- Un lenguaje de especificación que emplearemos para la representación de las propiedades que deseamos comprobar en el sistema. En este caso no es más que la lógica temporal con la que trabajamos, que debe ser lo suficientemente expresiva para reflejar todas las cualidades del sistema que queremos modelar.

- Una clasificación de propiedades que agrupe en categorías homogéneas las distintas características que deseamos estudiar. Esto no sólo nos ayudará a verificar la compleción de nuestra especificación, sino que nos

servirá para orientar nuestros procedimientos de verificación en función de los objetivos.

- Un procedimiento de verificación de la fórmula lógica sobre el modelo computacional que representa al sistema. Este procedimiento suele estar compuesto por un conjunto de reglas que puede dividirse en dos categorías: una general, para reglas universalmente válidas, y una particular, para reglas válidas solamente en el sistema bajo estudio (generalmente relacionadas con el dominio de los datos que maneja).

Como ya comentamos, el *model checking* consiste en comprobar si el modelo computacional del sistema estudiado es realmente un modelo para la propiedad, es decir, si la propiedad está presente en el conjunto de posibles ejecuciones del sistema.

Los primeros trabajos al respecto [EL86] tenían por objetivo averiguar si una fórmula se satisfacía en el modelo de estados de un sistema. Para ello, procedían a la búsqueda del conjunto de estados del sistema que satisfacían la propiedad, para luego comprobar si el estado en el que queríamos verificar la propiedad estaba entre ellos. Aunque la tarea se llevase a cabo mediante una serie de aproximaciones de carácter finito [Sti96], el procedimiento era poco escalable para sistemas de gran tamaño y, por supuesto, no aplicable a sistemas con un número no finito de estados.

Posteriores refinamientos condujeron a lo que Stirling y Walker denominaron *local model checking* [Lar90, SW91, Win91]. En este caso, el objetivo es la comprobación directa de si la fórmula se verifica o no en un estado, sin tener que calcular el conjunto de estados del sistema.

El mecanismo general se basa en aplicar un conjunto de reglas de inducción estructural sobre la fórmula, de tal forma que el problema de verificar la fórmula original se transforme en la verificación de una o más subfórmulas de menor complejidad. A través de la aplicación reiterada del conjunto de reglas se llegará a fórmulas simples, cuya veracidad o falsedad puede decidirse directamente, o a situaciones de recursividad, que implicarán el cumplimiento o no de la fórmula original según el significado de ésta.

Un enfoque muy ilustrativo de este procedimiento es el basado en la bisimulación y la equivalencia de juegos que se desprende directamente de ella [Sti96].

3.3.1. Equivalencia de juegos

La equivalencia de juegos tiene su origen en la relación de bisimulación [Mil89]. Esta relación establece que dos procesos **p** y **q** son bisimilares ($p \sim q$) cuando:

1. Si $p \xrightarrow{a} p_1$, $\exists q_1 \ / \ q \xrightarrow{a} q_1 \ \ y \ \ p_1 \sim q_1$
2. Si $q \xrightarrow{a} q_1$, $\exists p_1 \ / \ p \xrightarrow{a} p_1 \ \ y \ \ p_1 \sim q_1$

Lo que expresa esta relación es que, haga lo que haga un proceso, el otro es capaz de imitarlo. Bajo este enfoque, se puede plantear la verificación de una fórmula Φ en el estado E de un sistema como un juego de bisimulación entre dos oponentes, uno de ellos tratando de demostrar que Φ no se verifica en el modelo (el jugador I) y el otro tratando de demostrar que sí lo hace (el jugador II).

El juego se desarrolla mediante la generación de una serie de pares estado-fórmula:

$$(E_0, \Phi_0), (E_1, \Phi_1), \ldots, (E_i, \Phi_i)$$

donde E_0 es E y Φ_0 es Φ.

Cada par (E_{i+1}, Φ_{i+1}) se genera mediante inducción estructural, en función de la conectiva principal de Φ_i:

- Si la conectiva principal expresa la conjunción de varias subfórmulas, será el jugador I el que elija una de ellas como Φ_{i+1}, siendo E_{i+1} igual a E.
- Si la conectiva principal expresa la necesidad de verificación de una fórmula Ψ en varios estados, (por ejemplo un operador de futuro **G** en CTL o un operador modal "`[-]`" en el *μ-calculus*) también será el jugador I el que elija uno de esos estados, que será E_{i+1}, mientras que Φ_{i+1} será Ψ.
- Si la conectiva principal expresa la disyunción de varias subfórmulas, será el jugador II el que elija una de ellas como Φ_{i+1}, siendo E_{i+1} igual a E.

- Si la conectiva principal expresa la necesidad de verificación de una fórmula Ψ en uno de varios estados (por ejemplo un operador de futuro **F** en CTL o un operador modal "<->

  " en el *μ-calculus*), también será el jugador II el que elija uno de esos estados, que será E_{i+1}, mientras que Φ_{i+1} será Ψ.

El jugador II (jugador I) gana el juego cuando la fórmula de un par es lo suficientemente simple para que su veracidad (falsedad) en el estado asociado pueda ser decidida directamente. También ganará si es el turno del jugador I (jugador II) y éste no puede mover (por no disponer de transiciones en el estado actual).

Al llegar a situaciones de recursividad, se da la victoria a uno u otro jugador en función de la semántica de la fórmula, según sea una finalidad o una invarianza (ver sección 3.4.1).

Como podemos ver, las reglas de turno y elección de movimiento son consistentes con los intereses de los jugadores. Cuando se deben verificar todas las posibilidades existentes, se da el turno al jugador I, para que elija la que más le conviene para demostrar que la fórmula no se verifica. Cuando se debe verificar una entre varias, se le da el turno al jugador II, para que elija la más apropiada para sus intereses.

Se dice que un jugador tiene una *estrategia ganadora* si es capaz de ganar el juego siempre, independientemente de las jugadas del oponente (cosa que ocurre siempre para alguno de los dos jugadores). Si es el jugador II quien tiene una estrategia ganadora, entonces la fórmula se verifica en el estado inicial, ya que termina ganando el juego, no importa lo que elija el jugador I.

El conjunto completo de reglas que gobiernan el desarrollo del juego y las condiciones de finalización dependen de la lógica con la que se trabaje. En [Sti96] puede encontrarse un ejemplo para el *μ-calculus*.

Evidentemente, el problema de este tipo de enfoque es encontrar la estrategia ganadora. Es un problema equivalente, por tanto, al de la construcción de un *tableau*, el método clásico de verificación de propiedades por *model checking*.

3.3.2. Tableau

Un *tableau* [SW91] es un árbol de prueba que tratamos de construir con éxito a partir de la fórmula y un estado del sistema. Si lo conseguimos, la fórmula se verifica en ese estado.

El procedimiento se basa en un conjunto de reglas de inducción estructural, a través de las cuales se descompone la fórmula padre en una o varias más simples cuyo cumplimiento garantiza la veracidad de la fórmula padre.

El nodo inicial del árbol está etiquetado con la fórmula y el estado originales. A partir de las reglas de inducción, se crean nuevos nodos hijos del árbol, etiquetados con subfórmulas en el mismo u otros estados. Si la fórmula implica la necesidad de que se verifiquen varias subfórmulas, se crea un nodo hijo por cada una de ellas. Si implica la necesidad de que se verifique una subfórmula de un conjunto, se elije una de ellas[2].

El proceso se desarrolla hasta que todos los nodos sean terminales. Un nodo se dice terminal cuando se puede decidir directamente la veracidad o falsedad de su fórmula en el estado que lleva asociado. Esto puede ocurrir por varios motivos: porque la fórmula es una afirmación básica directamente decidible, porque no se puede avanzar más, porque se llegó a una situación de recursividad, etc.

Si todas las fórmulas de los nodos terminales son ciertas, decimos que el *tableau* ha sido construido con éxito y, en ese caso, la propiedad se cumple en el sistema.

En este caso, la dificultad del procedimiento estriba en tomar las decisiones oportunas en el transcurso de la construcción del árbol para lograr que todos los nodos terminales contengan fórmulas ciertas.

Un *tableau* construido con éxito se puede ver como una representación de la estrategia ganadora (para el jugador II) que buscábamos en el caso de la equivalencia de juegos. Por tanto, ambos enfoques son equivalentes.

Igual que antes, las reglas exactas que gobiernan la construcción del *tableau* y las condiciones de finalización dependen de la lógica con la que se trabaje. En [Sti96] también pueden encontrarse varios ejemplos de construcción de *tableaux* para el μ-*calculus*.

[2]Esto implica directamente la existencia de varios árboles posibles. Con que uno de ellos sea construido con éxito será suficiente.

3.3.3. Equivalencia de sistemas

En la sección 3.2.1 observábamos que de la relación de satisfacción ($\models$) se derivaba directamente una relación de equivalencia ($\equiv_{\mathcal{L}}$) entre programas o especificaciones.

$$\mathbf{p} \equiv_{\mathcal{L}} \mathbf{q} \Longleftrightarrow \Phi_{\mathcal{L}}(\mathbf{p}) = \Phi_{\mathcal{L}}(\mathbf{q})$$

Ahora bien, esta equivalencia no tiene por qué ser suficiente para nuestros propósitos. Nuestra intención es validar un refinamiento del sistema, demostrando que presenta el mismo comportamiento que la fase anterior. Para ello, lo habitual es recurrir a las equivalencias tradicionales sobre un lenguaje de procesos.

Por tanto, es importante establecer las posibles relaciones existentes entre esta equivalencia temporal y las que se deriven de la semántica operacional del lenguaje de procesos empleado. Este estudio no es generalizable y debe ser llevado a cabo para cada lógica y cada equivalencia definida en un lenguaje de procesos.

Como ejemplo de tal estudio, Hennessy y Milner demuestran en [HM80, HM85] que la bisimulación entre procesos definida en CCS ($\sim$), implica el mismo conjunto de propiedades en la lógica modal de Hennessy-Milner. Es decir:

$$\mathbf{p} \sim \mathbf{q} \Longrightarrow \Phi(\mathbf{p}) = \Phi(\mathbf{q})$$

Por contra, lo inverso no es completamente cierto. Para poder afirmarlo, debemos restringirnos a un tipo especial de sistemas llamados *de imágenes finitas* [3]. Un proceso es *de imágenes finitas* si sus estados tienen un número finito de transiciones. Pues bien, entre procesos de imágenes finitas, la igualdad de propiedades implica la equivalencia de bisimulación.

$$[\mathbf{p}, \mathbf{q} \text{ de imágenes finitas}] \qquad \Phi(\mathbf{p}) = \Phi(\mathbf{q}) \Longrightarrow \mathbf{p} \sim \mathbf{q}$$

Por tanto, para sistemas de imágenes finitas, la equivalencia temporal es

[3] Traducción literal del término *image-finite*

equivalente a la bisimulación.

$$[\mathbf{p},\ \mathbf{q} \text{ de imágenes finitas}] \qquad \mathbf{p} \equiv_{\mathcal{L}} \mathbf{q} \Longleftrightarrow \mathbf{p} \sim \mathbf{q}$$

Posteriormente, Stirling [Sti96] generalizó este resultado para el *μ-calculus*, imponiendo una condición adicional: las fórmulas deben ser cerradas, es decir, no deben contener variables libres.

3.3.4. Verificación de sistemas con paso de valores

Una de las líneas de trabajo más interesantes en la actualidad se centra en el modelado de la información que los sistemas intercambian con su entorno. Esto, además de permitir una descripción más fiel del sistema y su dinámica, posibilita una representación compacta de muchos de los sistemas cuyo comportamiento tiene un carácter no finito.

Para ello, los modelos más habituales enriquecen los eventos que se emplean para describir el comportamiento del sistema, dotándolos de la capacidad de ofrecer información al exterior o aceptarla y asignarla a variables. Este hecho confiere un carácter simbólico a los distintos elementos involucrados en el formalismo [HL95a], al no representar un hecho concreto, sino el conjunto de posibilidades derivadas de los distintos valores que pueden ser asignados a las variables.

La adición de variables en las lógicas y lenguajes de procesos empleados suele ir acompañada de su enriquecimiento con estructuras de especificación condicional, cuantificadores sobre las variables, parametrización de los comportamientos, etc. Asimismo, en los procesos de verificación por inducción estructural se generan construcciones no cerradas, cuya semántida depende de los valores asignados en el pasado. Estos términos abiertos exigen la extensión de los sistemas de prueba existentes para introducir expresiones de datos que reflejen esta dependencia [HL95b].

El resultado son formalismos de verificación inevitablemente más complejos, que, en último lugar, suelen depender de otros sistemas de prueba auxiliares (especializados en el tratamiento de expresiones de datos) para la toma de decisiones. La mayor parte de los conceptos tradicionales se amplían para reflejar esta especificación simbólica del comportamiento del sistema: los sistemas

de transiciones etiquetados son sustituidos por los sistemas de transiciones simbólicos (sección 5.1), la bisimulación tradicional se extiende a la bisimulación simbólica [HL95a] (con sus variantes *early* y *late*), las equivalencias entre procesos se parametrizan con expresiones de datos, etc.

En el estado actual de la técnica, los formalismos de verificación de este tipo no son completos. Cada uno de ellos suele establecer un conjunto de restricciones a su ámbito de aplicación [HL95b, HR97]: los modelos de sistemas susceptibles de análisis, el tipo de propiedades verificables, los tipos de recursiones cubiertas, etc.

3.4. Propiedades de interés

El estudio de un sistema reactivo a través del conjunto de propiedades que satisface suscita inmediatamente un interrogante: ¿Cuál es el conjunto de propiedades que queremos verificar?.

La respuesta, evidentemente, depende del sistema en cuestión. Sin embargo, es bien cierto que, en la mayor parte de los sistemas reactivos, el conjunto de propiedades de interés responde a unos patrones que se repiten con gran frecuencia. A partir de este hecho, se han producido múltiples intentos de establecer clasificaciones de propiedades en función de características comunes [Lam89, DW89, MP90, CMP92].

Un objetivo importante que se persigue con la caracterización de un conjunto de clases que reúnen a propiedades de perfil similar es verificar la compleción de la especificación [MP89, MP90]. La mejor forma de asegurarnos de que nuestra especificación es completa y no hemos olvidado ningún comportamiento importante es un enfoque sistemático. Si disponemos de una biblioteca de propiedades, clasificadas por características comunes, podemos chequear el sistema contra todas y cada una de ellas (descartando las que consideremos irrelevantes para ese sistema).

Una característica importante de una lógica es su capacidad para recoger o expresar todos aquellos aspectos del sistema cuya modelación es conveniente. Es lo que se denomina *expresividad* de la lógica. Pnueli [Pnu85] formalizó este concepto un tanto abstracto diciendo que una lógica $\mathcal{L}$ es expresiva para un lenguaje de procesos $\mathcal{P}$ si se satisface:

$$\forall p \in \mathcal{P},\ \exists\ \mathcal{L}(p) \in \mathcal{L} \qquad \text{tal que}$$

1. $\forall q \in \mathcal{P} \quad \longrightarrow \quad p \models \mathcal{L}(q) \Longleftrightarrow p \equiv q$
2. $\forall \phi \in \mathcal{L} \quad \longrightarrow \quad p \models \phi \Longleftrightarrow \mathcal{L}(p) \Rightarrow \phi$

3.4.1. Clasificaciones

Una partición que podemos establecer de forma natural es la que divide las propiedades entre las de carácter funcional y no funcional.

Propiedades funcionales. Son aquellas que caracterizan el comportamiento del sistema en sus aspectos cualitativos. Es decir, se limitan a especificar el ordenamiento de los eventos que describen la evolución del sistema.

Propiedades no funcionales. Se ocupan de ciertas restricciones a las que se ve sometido el sistema: velocidad de ejecución, tiempo de respuesta, recursos disponibles, etc. Abarcan, por tanto, aspectos cuantitativos que afectan a la implementación del sistema.

Sin embargo, la clasificación más extendida en los formalismos que emplean lógica temporal es la debida a Lamport [Lam89], donde se establecen dos grandes grupos de propiedades:

Propiedades de seguridad. Su objetivo es especificar que *algo malo* no va a ocurrir en el sistema. Son propiedades que, por tanto, excluyen situaciones indeseadas del comportamiento del sistema. En términos del conjunto de trazas de comportamiento del sistema, en ninguna traza está presente el efecto indeseado.

Ejemplos bien conocidos de este tipo de propiedades son: ausencia de un estado de bloqueo, acceso correcto a regiones críticas, etc.

Propiedades de viveza. Estas propiedades aseguran que *algo bueno* va a ocurrir en el sistema. Por tanto, nos garantizan que ese sistema, tarde o temprano, llegará a un determinado estado. En este caso, todas las trazas de comportamiento del sistema contienen el suceso deseado.

Por ejemplo, una propiedad de viveza puede servir para garantizar el acceso a un recurso compartido o para establecer la equidad (*fairness*) del sistema (propiedad muy importante con la que se designa el hecho de que ningún componente de un sistema concurrente sea retardado indefinidamente).

Estableciendo una analogía con sistemas secuenciales tradicionales [MP90], una propiedad de seguridad sería la corrección parcial: no se garantiza la finalización pero sí que, de producirse, el resultado nunca será incorrecto. Por contra, una propiedad de viveza sería la corrección total: se garantiza la finalización del comportamiento del sistema.

La anterior caracterización obliga a que la propiedad esté presente en todas las trazas posibles. Stirling [Sti96], por su parte, definió las propiedades de seguridad o de viveza débiles (*weak safety* y *weak liveness*) como las que sólo exigen la observación de la propiedad en alguna de las trazas (algo malo nunca ocurre en alguna de las secuencias de ejecución o algo bueno ocurre en alguna de esas secuencias).

En todo caso, la clasificación de una propiedad en uno u otro grupo no siempre es sencilla, dependiendo muchas veces de la interpretación realizada. Por ejemplo, la ausencia de bloqueo puede verse como una propiedad de seguridad (nunca se alcanza un estado de bloqueo) o de viveza (el sistema siempre puede avanzar).

En [MP89], Manna y Pnueli, basándose en formalizaciones de las propiedades de seguridad y viveza realizadas en [Lam83] y [AS85], prueban que:

- Las clases de propiedades de seguridad y viveza puras son disjuntas.
- Cualquier propiedad puede ser representada en términos de una propiedad de seguridad y otra de viveza.

A partir de esta simple clasificación en dos grandes grupos, muchos autores han procedido a realizar refinamientos de distinto nivel de detalle. Por ejemplo, en [MP89] se caracterizan una serie de propiedades en base a criterios de persistencia, ciclicidad, repeticiones infinitas, etc.

En todo caso, la mayor parte de las distintas clasificaciones suelen ser variaciones, de mayor o menor alcance, sobre tres arquetipos muy comunes:

Invarianzas. Son propiedades que se cumplen en todos los estados del sistema estudiado.

Finalidades. Se trata de propiedades que, no importa cual sea la evolución del sistema, acabarán por ser ciertas en algún estado en el futuro.

Precedencias. Estas propiedades establecen un cierto orden o precedencia en la sucesión temporal de varios eventos del sistema.

Capítulo 4

Objetivos de la Tesis

4.1. Captura de la arquitectura inicial

En el primer capítulo de esta memoria realizamos una introducción a las Técnicas de Descripción Formal y su utilización en el diseño y construcción de sistemas distribuidos complejos. Resaltábamos allí que un procedimiento empleado con gran frecuencia consiste en la construcción de una arquitectura inicial seguida de un conjunto de refinamientos que completan paulatinamente la especificación. El proceso de refinamientos se prolonga hasta alcanzar una descripción con el suficiente nivel de detalle como para realizar una implementación directa, frecuentemente semiautomática.

También mencionábamos que la captura de la arquitectura inicial suele llevarse a cabo por procedimientos tradicionales y que ése era el origen de frecuentes incompleciones e inconsistencias en la especificación de la funcionalidad del sistema [Som95]. Hay un consenso generalizado en que el empleo de métodos más formales en esta primera fase del proceso evitaría y descubriría errores en un momento en donde su reparación es aún poco costosa.

El objetivo de la presente tesis es la formalización de esta etapa inicial. Esta intención general debería concretarse en un procedimiento metodológico para la construcción de la arquitectura inicial de un sistema distribuido complejo.

Este procedimiento debería establecer una sucesión de etapas en el proceso de construcción de la arquitectura inicial, de tal forma que, con la correspondiente particularización en cada caso, la captura de la arquitectura inicial pudiera

llevarse a cabo de una forma sistemática y previsible.

Entre otras, algunas características básicas que deberíamos pedir a este procedimiento serían:

- Debe ofrecer soporte para la captura y representación de los requisitos funcionales del sistema, es decir, las ordenaciones temporales correctas de los sucesos que pueden acaecer en el sistema y que conforman su comportamiento.
- Debe ofrecer mecanismos para especificar y comprobar ciertas características que deseamos que tenga el sistema, así como apoyo en la identificación de los motivos de fallo y en las modificaciones pertinentes.
- Debería contemplar la tipificación de la información que el sistema intercambia con el entorno (o entre distintas partes del sistema).
- Las etapas del procedimiento deberían ser lo suficientemente flexibles para reparar omisiones o decisiones erróneas a lo largo del proceso. Es conveniente la posibilidad de una realimentación natural entre diversas etapas que facilite cualquier corrección que se estime oportuna.
- Sería bueno contemplar, desde el principio, las peculiaridades de un sistema distribuido y orientar el procedimiento a tal efecto. En concreto, debería facilitar el diseño por separado de los diversos subsistemas y su integración y sincronización posterior.

4.2. Etapas del procedimiento

Con los objetivos y condicionantes descritos hasta el momento, proponemos un procedimiento para la captura de la arquitectura inicial que articula el proceso en las siguientes etapas:

1. Identificación de los eventos que intervienen en la descripción del comportamiento del sistema.
2. Especificación de las trazas inicialmente conocidas y solapamiento de las mismas para generar un sistema de transiciones etiquetadas (sección 2.3.2).

3. Formulación y verificación de las propiedades que deseamos que cumpla el sistema.

4. Modificación del sistema para lograr el cumplimiento de aquellas propiedades que no se verifican.

5. Síntesis automática de la arquitectura inicial del sistema en una técnica de especificación formal constructiva.

En la figura 4.1 se puede ver una representación gráfica del procedimiento descrito.

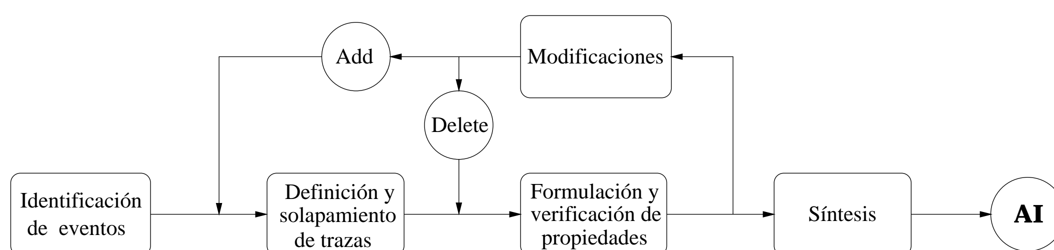

Figura 4.1: Captura de la arquitectura inicial

En caso de tratarse de un sistema compuesto de varios subsistemas que cooperan entre sí (como ocurre con los sistemas distribuidos), podríamos aplicar el procedimiento para obtener la arquitectura inicial de cada uno de los subsistemas. En ese caso, habría que añadir una etapa posterior de integración y sincronización de las arquitecturas iniciales diseñadas para obtener la del sistema final.

4.2.1. Especificación de los eventos del sistema

El primer paso en el diseño de todo sistema dinámico consiste en la identificación del conjunto de eventos que caracterizan su comportamiento y hacen observable su evolución. Estos eventos, ya ocurran por iniciativa del sistema o del entorno, describen la dinámica del sistema y sus cambios de estado, a un nivel de abstracción determinado.

La elección de los eventos observables supone la primera decisión de diseño que debemos tomar. Normalmente, en una especificación formal, los eventos son sucesos atómicos, es decir, indivisibles. Por tanto, no es posible profundizar a un nivel de detalle mayor que el que marca la ocurrencia discreta de un evento. La realidad, en cambio, es siempre continua. En general, cualquier suceso podría ser descompuesto en varias etapas secuenciales, resultando una descripción más detallada del comportamiento. Nuestra necesidad de ese detalle será la que determine la granularidad de los eventos. Por tanto, la elección del conjunto de eventos determina, indirectamente, el nivel de refinamiento con el que vamos a estudiar el sistema.

Un procedimiento muy habitual en la especificación de sistemas dinámicos consiste en modelar la información que se intercambia con el entorno a través de los eventos que provocan las transiciones entre los estados del sistema. Siguiendo esa línea, un objetivo de nuestro formalismo es ofrecer un marco de especificación que permita una descripción de los eventos del sistema y del intercambio de información con el entorno que puede ocurrir a través de cada evento. El procedimiento debería ser abierto e incluir tanto a los tipos de datos tradicionales (enteros, *booleanos*, reales, tipos compuestos, etc.) como a los de propósito específico definidos por los usuarios.

4.2.2. Especificación y solapamiento de las trazas iniciales

Tras establecer el conjunto de eventos que provocan o reflejan la evolución del sistema, empezaremos a especificar las ordenaciones temporales correctas de esos eventos.

El procedimiento que estamos definiendo está destinado a ser aplicado en las fases iniciales del proceso de desarrollo, en las cuales no hay todavía un conocimiento ni claro ni completo de cómo debe comportarse el sistema. Nos encontramos en una fase en donde tratamos de experimentar con diversas alternativas que nos parecen viables y ver con qué grado de fidelidad responden al objetivo buscado.

Lo habitual en estos casos es disponer solamente de una cierta idea, más o menos precisa, de algunas de las trazas del sistema. Es decir, conocemos algunas de las secuencias de eventos que pensamos (inicial y provisionalmente) que

deben ser posibles en la evolución del comportamiento del sistema. El carácter provisional que hemos atribuido a esas trazas iniciales pretende resaltar que ni son las únicas (seguramente ese conjunto inicial se verá incrementado al aumentar el conocimiento que tengamos del sistema) ni son imprescindibles (cabe la posibilidad de que alguna de ellas sea incompatible con alguna de las propiedades que vamos a exigir al sistema).

Por tanto, uno de los objetivos del procedimiento será ayudar en la confección de las trazas conocidas del sistema, compuestas por los eventos que gobiernan su evolución desde su inicio hasta su finalización. En el caso de sistemas reactivos, esas trazas deberán ser obligatoriamente recursivas, al tratarse de sistemas cuyo comportamiento no tiene fin (al menos, no un fin previsto).

Tras definir este conjunto de trazas iniciales del sistema, procederemos a crear una estructura única que recoja de una forma más compacta todas las ordenaciones identificadas. Se trata de solapar las trazas entre sí para eliminar redundancias, unificando parcialmente varias trazas en una sola hasta el momento en que diverjan los comportamientos que describen. La implementación de un procedimiento automático para lograr este solapamiento será nuestro objetivo más importante en esta etapa.

El resultado de este proceso será una estructura muy bien conocida y frecuentemente empleada en el terreno de la especificación formal: un sistema de transiciones etiquetadas (sección 2.3.2). Esta estructura, que denominaremos coloquialmente *árbol del sistema*, describirá el conjunto de estados en los que podrá encontrarse este sistema inicial que hemos derivado de las trazas conocidas hasta el momento. En cada uno de los estados estarán definidas una serie de transiciones a otros estados del árbol, cada una provocada por alguno de los eventos que intervienen en la descripción del comportamiento del sistema.

En realidad, debido a que los eventos pueden llevar asociadas variables para representar la información que intercambian con el entorno, la estructura resultante será un grafo simbólico (sección 5.2):

- En cada estado estarán declaradas una serie de variables libres que almacenan información asignada en estados previos y que pueden ser empleadas para ofrecer información al exterior.
- Las transiciones pueden ir condicionadas por expresiones en las que intervienen las citadas variables. De esta forma, las posibles evoluciones del sistema dependen de los valores de esas variables.

4.2.3. Formulación y verificación de propiedades

Una vez que disponemos del prototipo de partida, es el momento de razonar sobre las propiedades del sistema. Este es, en definitiva, el principal objetivo de un procedimiento de especificación formal: verificar matemáticamente que el sistema cumple ciertas propiedades.

En esta fase, por tanto, el diseñador debe formular las propiedades o requisitos que cree que el sistema debe cumplir (que el conjunto de posibles comportamientos del sistema debe cumplir). En nuestro procedimiento vamos a contemplar solamente los requisitos funcionales del sistema, aquellos relacionados con las posibles secuencias de eventos que el sistema pueda llevar a cabo. Vamos a estudiar, por tanto, propiedades que caractericen a los ordenamientos temporales correctos de los eventos del sistema, dejando para etapas posteriores los aspectos no funcionales (rendimiento, recursos, etc.).

Como ya hemos mencionado en el capítulo 3, este tipo de propiedades se puede clasificar en dos grandes apartados: las finalidades (o propiedades de viveza) y las invarianzas (o propiedades de seguridad). El objetivo genérico de las finalidades es asegurar que algo va a suceder en el sistema a lo largo de su evolución temporal. Las invarianzas, por contra, describen generalmente la exigencia de que algo no pueda ocurrir (o dejar de ocurrir) en el sistema.

Para describir estas propiedades, y dado que modelan relaciones de orden temporal, lo más indicado será utilizar una lógica temporal. Por tanto, nuestro primer objetivo en esta etapa será elegir una lógica para tal tarea. La lógica escogida debería ser suficientemente expresiva para describir los distintos escenarios sobre los que pretendemos razonar. En concreto, debería proporcionar constructores para establecer no sólo las propiedades, sino también ciertas relaciones de causalidad entre varios requisitos. Para ello, de ser necesario, procederíamos a un enriquecimiento de los constructores que definen su expresividad.

A la hora de definir las propiedades del sistema, nos parece interesante eximir al diseñador de la necesidad de conocer los detalles de la lógica empleada para tal propósito. Con tal motivo, junto con el procedimiento se debería proporcionar una biblioteca de propiedades fundamentales parametrizadas, con su funcionalidad descrita en base a los parámetros modificables por el diseñador. Éste sólo tendrá que elegir la propiedad observando su funcionalidad (descrita en un lenguaje natural) y rellenar los parámetros correspondientes. De esta

forma, además, se ayudaría a garantizar la corrección de la propiedad, objeto siempre de incertidumbre ante resultados confusos o inesperados en el proceso de verificación.

Una vez que el diseñador ha establecido el conjunto de finalidades e invarianzas que su conocimiento actual del sistema le permite especificar, el siguiente paso a seguir será la comprobación de que el prototipo inicial cumple esas propiedades. Para ello, el último objetivo de esta fase consiste en el desarrollo de un procedimiento para determinar el cumplimiento o no de cada propiedad en el sistema.

4.2.4. Modificación del sistema

Evidentemente, si el sistema en su estado actual cumple una determinada propiedad, no existe ningún problema y podemos dar por concluido este apartado en lo referente a esa propiedad en concreto. Si no es así, lo que procede es estudiar el motivo del incumplimiento y, si es posible, modificar adecuadamente el árbol del sistema para lograr el cumplimiento de la propiedad.

Generalmente, el tipo de propiedad nos aporta información sobre las características del incumplimiento.

- En el caso de una finalidad, los incumplimientos son debidos a que nunca se alcanza un estado donde se cumple una cierta condición (por ejemplo, la posibilidad o ausencia de un evento). La solución, pues, pasa por añadir esa condición a algún estado.
- Por contra, el incumplimiento de una invarianza suele ser debido a que cierta condición no se cumple en algún estado, cuando debería hacerlo en todos. En este caso, la solución es más concreta, al conocer con precisión los estados sobre los que hay que actuar.

En general, aun cuando las modificaciones que haya que realizar para lograr el cumplimiento de una propiedad sean conceptualmente sencillas, su determinación puede complicarse notablemente por varios factores:

- No suele haber una única alternativa de modificación del sistema para

cumplir la propiedad. Probablemente existan varias elecciones de diversa dimensión y distinta repercusión en el futuro del desarrollo.

- Los cambios a realizar en cada modificación pueden afectar a múltiples eventos en distintos estados, resultando difícil su identificación de manera conjunta.

También puede ocurrir que sea del todo punto imposible modificar el sistema para que cumpla esa propiedad, por ejemplo, porque dejaría de cumplir alguna otra. Estamos, entonces, ante una incoherencia de las propiedades que atribuimos al sistema, que describen requisitos contradictorios. En este caso, por tanto, debemos revisar nuestros planteamientos acerca del sistema y decidir los cambios necesarios en los requisitos para eliminar la incoherencia detectada.

Tal como comentamos en la sección 2.4.2, una de las ventajas del *model checking* es la posibilidad de producir contraejemplos para las propiedades que no se cumplen. Es, por tanto, un objetivo de nuestro formalismo el proporcionar información acerca del motivo del incumplimiento de las propiedades que se verifiquen. Adicionalmente, en función del motivo del incumplimiento, trataremos de aportar sugerencias sobre lo que se debería cambiar en el sistema para lograr que la propiedad se verifique.

En todo caso, es necesario someter al desarrollo a un continuo chequeo de integridad, para comprobar que los cambios realizados para lograr el cumplimiento de una propiedad no afecten al cumplimiento de las propiedades verificadas con anterioridad.

4.2.5. Síntesis de la arquitectura inicial

Las etapas anteriores se sucederán, posiblemente con realimentaciones, hasta alcanzar un sistema de transiciones simbólico que represente un sistema que contenga las trazas de actuación del que pretendemos diseñar y que cumpla las propiedades funcionales que hemos establecido en su momento.

Estamos, por tanto, ante la arquitectura inicial que buscábamos. Esta descripción inicial del sistema es el primer paso sólido en el proceso de desarrollo. A partir de aquí, tal como introducíamos en la sección 1.4, se procede a refinar paulatinamente esta arquitectura inicial, añadiendo información y trans-

formándola hasta alcanzar una descripción que sea lo suficientemente completa para llevar a cabo una implementación semiautomática.

Sobre la especificación que hemos conseguido resta, todavía, un más o menos largo proceso de razonamiento y estructuración, donde se satisfacen diversas necesidades:

- Comprobar la validez de los objetivos mediante la creación de prototipos.
- Definir la estructura del sistema.
- Transformación de la especificación a diferentes estilos, con los propósitos de identificación de subsistemas, definición de máquinas de estados para facilitar la implementación, etc.
- Verificar formalmente las transformaciones realizadas.

Parece apropiado, por tanto, expresar esa arquitectura inicial mediante una técnica de especificación formal constructiva, que nos permita enriquecer la descripción del sistema en los términos descritos, utilizando, en lo posible, las múltiples herramientas de apoyo que existen en cada caso.

En el procedimiento aquí definido, sintetizaremos de forma automática una especificación E-LOTOS [ISO98] a partir de la arquitectura inicial conseguida. Esta especificación describirá simplemente las trazas de comportamiento del sistema, sin que exista ningún tipo de estructura.

4.3. Formalización de las etapas

El procedimiento descrito en la sección anterior define una serie de etapas con unos objetivos a alcanzar en cada una. En él se exponen las ideas generales, sin descender a los detalles de implementación ni a la descripción de los instrumentos creados o utilizados a tal efecto.

La materialización de esas etapas en un procedimiento formal plantea una serie de problemas de distinta índole, algunos de ellos completamente abiertos en lo que se refiere al estado actual de la técnica. En esta tesis se concretan las etapas de ese procedimiento en un formalismo que integra una serie de

notaciones y algoritmos para alcanzar el objetivo descrito: una herramienta de apoyo a la captura formal de la arquitectura inicial de un sistema complejo.

En los siguientes capítulos se detallan los elementos envueltos en el procedimiento:

Capítulo 5 Se define formalmente un sistema de transiciones simbólico (STS) y se detallan las particularidades del que vamos a emplear en nuestro procedimiento para modelar el sistema en desarrollo.

Capítulo 6 Se introduce la lógica temporal definida para articular el mecanismo de verificación de propiedades. Se propone una biblioteca de arquetipos o propiedades temporales genéricas a partir de las cuales se pueden formular aquellas características que deseamos comprobar en el modelo del sistema.

Capítulo 7 Se describe la utilización de la lógica temporal LTCS para especificar las trazas iniciales del sistema. Se expone el mecanismo de solapamiento de varias trazas de comportamiento para conseguir un STS que nos sirva como modelo del sistema.

Capítulo 8 Se presenta un algoritmo de verificación de propiedades temporales sobre un sistema de transiciones simbólico y un procedimiento de análisis de los resultados.

Capítulo 9 Se define un algoritmo para la generación de sugerencias sobre la modificación de un sistema que no cumpla una propiedad.

Parte II

Desarrollo

Capítulo 5

Sistemas de Transiciones Simbólicos

5.1. Introducción

En la sección 2.3.2 introducíamos los sistemas de transiciones etiquetados como una opción muy apropiada para representar la evolución temporal de un sistema. La dinámica del sistema quedaba completamente especificada mediante la identificación de todos los estados en los que podía encontrarse el sistema y los eventos, internos o externos, que provocaban las transiciones de un estado a otro.

Como mencionamos en la sección 2.4.2, la utilidad de este tipo de representación en cuanto a la verificación mediante *model checking* se refiere, estaba restringida a sistemas de carácter finito, con un número limitado de estados y transiciones.

Sin embargo, el hecho de que el formalismo que estamos definiendo contemple el modelado de la información que el sistema intercambia con su entorno, convierte a los sistemas que pretendemos desarrollar en intrínsecamente infinitos. Ello es debido, fundamentalmente, a la capacidad con la que dotamos a los eventos para reflejar ese movimiento de información al producirse las transiciones entre estados.

Esta capacidad de ofrecimiento o aceptación de valores en el transcurso de las transiciones, unido a que los tipos de esos valores puedan no ser finitos (por

ejemplo, si trabajamos con números enteros), puede conducir a la existencia de infinitas transiciones distintas, cada una de ellas derivada de un valor concreto. Ello nos conduciría a un sistema de transiciones etiquetado con infinitas transiciones e infinitos estados, no apto para una exploración de estados exhaustiva mediante el *model checking* temporal.

Para tratar con esta dificultad, en lugar de utilizar sistemas de transiciones etiquetados, trabajaremos con sistemas de transiciones simbólicos (STS) [HL95a].

En este capítulo introduciremos de forma general a los sistemas de transiciones simbólicos. Esta será la estructura que, siguiendo la pauta trazada en el capítulo 4, utilizaremos para representar el comportamiento del sistema en desarrollo.

Posteriormente, describiremos alguna de las particularidades que presentan los sistemas simbólicos de nuestro formalismo. Las extensiones han sido realizadas, principalmente, con el propósito de ayudar a garantizar la coherencia del proceso de desarrollo.

5.2. Sistemas de transiciones simbólicos

La principal característica de los sistemas de transiciones simbólicos es la introducción de variables para almacenar y representar la información que el sistema intercambia con su entorno. Ello convierte a los eventos que provocan las transiciones en eventos simbólicos, al reflejar en una variable a todos los posibles valores que pueden verse involucrados en un intercambio de información.

Del mismo modo, las transiciones también podrán ser simbólicas puesto que, si contienen alguna de esas variables, representarán a todas las transiciones distintas que se derivarían de cada uno de los valores potenciales de las variables.

Cada una de esas transiciones simbólicas conducirá a un estado simbólico, que estará caracterizado por las variables que se hayan definido en los eventos que etiquetan las transiciones que conducen a él. Estos estados simbólicos representarán a todos los estados reales que se derivan de asignar un valor distinto a las variables que los caracterizan.

5.2.1. Definición formal

La definición formal de un sistema de transiciones simbólico es similar a la de un sistema de transiciones etiquetado (sección 2.3.2). Matemáticamente, es una estructura que puede ser representada por una cuádrupla ($\mathcal{S}_s$, s_0, $\mathcal{L}_s$, $\mathcal{T}_s$), aunque la definición de cada elemento sea un poco más compleja.

- $\mathcal{L}_s$ representa al conjunto de eventos simbólicos que etiquetan las transiciones entre los distintos estados del sistema.

  Como hemos mencionado, estos eventos simbólicos podrán llevar asociado un intercambio de información entre el sistema y su entorno, intercambio que podrá ser modelado mediante variables.

  Cada evento simbólico (e_s) estará formado por su identificador i_s y un patrón p_s, opcional, que describe el tipo de datos que se intercambia y el sentido en el que fluye.

  Siguiendo la sintaxis de alguno de los lenguajes de procesos más difundidos [ISO89b, Mil80], emplearemos el operador **"?"** para especificar la entrada de información y **"!"** para indicar la salida. Por tanto:

  - $\mathbf{i_s}$ **?x:T** representará al evento $\mathbf{i_s}$, que acepta un valor del tipo **T** procedente del entorno y lo almacena en la variable **x**. Al ocurrir el evento, diremos que se liga la variable **x**, que pasa a ser una variable libre de los estados siguientes.
  - $\mathbf{i_s}$ **!e** representará al evento $\mathbf{i_s}$, que ofrece al exterior la expresión **e**. Esta expresión estará compuesta por las constantes y operaciones de los tipos de datos definidos en el formalismo y podrá incluir a las variables libres del estado en el que es posible el evento.

  El patrón p_s de un evento e_s puede estar compuesto de varios intercambios de información. El conjunto de variables que acepten datos en el patrón formarán el conjunto de variables ligadas en el evento, $bv(e_s)$.

- $\mathcal{S}_s$ representa al conjunto de estados simbólicos en los que se puede encontrar el sistema en un momento dado. El estado inicial, s_0, formará parte de ese conjunto.

  Cada estado del sistema (s_i) se caracterizará por una serie de variables libres, $fv(s_i)$, que almacenarán la información aceptada en transiciones

previas. Es decir, el conjunto de variables libres de un estado s_i será el conjunto de variables ligadas en todos los eventos que etiquetan las transiciones que conducen a s_i.

Si del estado s_i puede pasarse al s_j a través de una transición etiquetada con el evento e_s:

- Las variables libres de s_j serán las de s_i más las variables ligadas en e_s.

$$fv(s_j) = fv(s_i) \cup bv(e_s)$$

- Si a través de e_s se ofrece una expresión al exterior (e_x), todas las variables contenidas en esa expresión deben ser variables libres del sistema en ese estado.

$$\forall\ i_s\ !e_x \in s_i \longrightarrow fv(e_x) \subseteq fv(s_i)$$

- $\mathcal{T}_s$ es el conjunto de transiciones simbólicas. Cada una de estas transiciones estará formada por el estado origen, el estado destino y el evento simbólico que etiqueta o provoca la transición.

  Además, cada transición puede estar condicionada por una guarda o expresión (g_s) que debe ser cierta para que la transición sea posible. Esta relación suele denotarse gráficamente por:

$$s_i \xrightarrow{e_s[g_s]} s_j$$

  Si el evento simbólico e_s está condicionado por una guarda g_s, cada una de las variables que aparezcan en esa expresión debe formar parte del conjunto de variables libres del estado origen o del conjunto de variables ligadas en e_s.

$$fv(g_s) \subseteq fv(s_i) \cup bv(e_s)$$

En la figura 5.1 puede observarse un ejemplo de un sistema de transiciones simbólico sencillo donde:

- El conjunto de variables ligadas en el evento *read* es $\{h,i\}$.
- El conjunto de variables libres en el estado s_2 es $\{h,i,t\}$.
- La transición de s_1 a s_4 está condicionada por la expresión "$h > 0$".

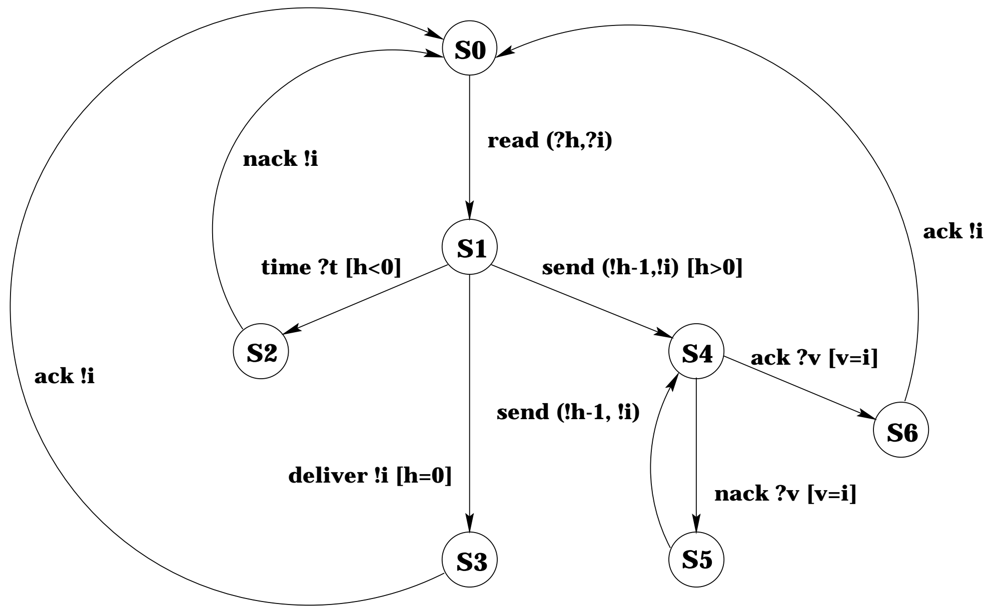

Figura 5.1: Sistema de transiciones simbólico

5.3. Preservación de la coherencia

Un objetivo fundamental del formalismo que estamos definiendo es ayudar a preservar la coherencia del proceso de desarrollo.

Tal como se introdujo en la sección 4.2 y desarrollaremos en posteriores capítulos, cuando dispongamos de una representación inicial del sistema en forma de STS, procederemos a comprobar si ésta cumple las propiedades temporales que consideremos de interés.

En caso de que una propiedad no se cumpla, trataremos de modificar la representación del sistema para que lo haga. Para ello contaremos con las sugerencias que serán calculadas automáticamente a partir del algoritmo descrito en el capítulo 9.

Sin embargo, las modificaciones que llevemos a cabo en el árbol simbólico para lograr el cumplimiento de la nueva propiedad deben mantener la integridad del desarrollo. Es decir, hay que garantizar que las modificaciones realizadas no deriven en el incumplimiento de alguna propiedad verificada anteriormente.

Para ello, es necesario realizar anotaciones acerca de las condiciones que exigen las distintas propiedades que vamos verificando. Cada una de ellas establecerá una serie de invariantes que no deben ser modificados para garantizar que se siga cumpliendo la propiedad. Esos invariantes deberán ser anotados en cada estado, junto a la propiedad que los impuso.

Para reflejar esta información, vamos a enriquecer nuestro modelo de sistema de transiciones simbólico con varios elementos que nos permitirán no sólo mantener esa coherencia sino también establecer distintos grados de incumplimiento de una propiedad.

5.3.1. Eventos fijos

El cumplimiento de una invarianza puede exigir que un evento sea posible en un estado determinado. En ese caso, debemos descartar futuras modificaciones del sistema de transiciones simbólico que eliminen ese evento del estado en cuestión.

Si la propiedad es una finalidad que hace referencia a la posibilidad del evento en algún estado, es posible que podamos eliminarlo de ése si es necesario, pero deberemos comprobar que todavía se cumple la finalidad o, en otro caso, añadir el evento en otro estado para conseguirlo.

Son dos ejemplos que nos muestran el distinto grado de necesidad de un evento en un estado:

- Ninguno. El evento puede ser eliminado sin contradecir ninguna propiedad verificada con anterioridad.

- Parcial. El evento puede ser eliminado bajo determinadas circunstancias y comprobaciones.
- Total. El evento no puede ser eliminado. Hablaremos, en ese caso, de un evento fijo.

Para determinar correctamente el grado de incumplimiento de una propiedad y mantener la integridad a la hora de tomar decisiones de modificación, vamos a anotar en los estados del árbol simbólico del sistema el grado de necesidad de cada evento posible.

Estas anotaciones nos permitirán decidir no sólo si un requisito se verifica en un sistema de transiciones simbólico, sino también obtener información acerca de la viabilidad de una modificación del sistema para remediar un hipotético incumplimiento.

El grado de necesidad de un evento en un estado estará asociado a la propiedad que lo impone. Por tanto, lo que vamos a almacenar con cada evento es la lista de propiedades verificadas que han realizado alguna afirmación sobre él, cada una junto con el grado de necesidad que le atribuye. El grado de necesidad del evento, por tanto, será el más restrictivo de todos los grados de necesidad especificados.

Para indicar el grado de necesidad de un evento en una representación gráfica del árbol simbólico, le añadiremos un modificador en su sintaxis: $\hat{e_s}$ (evento totalmente necesario) y $\check{e_s}$ (evento parcialmente necesario).

5.3.2. Eventos prohibidos

Un razonamiento muy similar al anterior puede llevarse a cabo para las afirmaciones que realicen las propiedades acerca de la prohibición de que un evento sea posible en un cierto estado.

En función del tipo de propiedad, su verificación puede derivar en la catalogación de determinados eventos como total o parcialmente prohibidos en un determinado estado.

Esta información también deberá ser almacenada en todos los estados afectados, para avisar ante un intento de adición de uno de esos eventos prohibidos por propiedades anteriores.

La sintaxis escogida en este caso para reflejar tal caracterización es: $\bar{e}_s$ (evento totalmente prohibido) y $\breve{e}_s$ (evento parcialmente prohibido).

Capítulo 6

La lógica LTCS

6.1. Descripción de la lógica

La **Lógica Temporal Causal Simple (LTCS)** [PAGDGS^{+}98] es una lógica temporal especialmente creada para la captura y especificación de los requisitos de un sistema, en especial aquellos relacionados con las restricciones de las ordenaciones temporales de los eventos que componen su comportamiento y determinan sus propiedades persistentes.

Su principal objetivo es ofrecer un marco descriptivo, cercano al lenguaje natural, para que el usuario especifique las propiedades temporales que desea verificar sobre el sistema en desarrollo. En ese marco se pretende experimentar con distintas alternativas sobre la interpretación de los operadores temporales, sus estados de aplicabilidad y la relación de satisfacción que se les asocia.

Como su nombre indica, la causalidad es un elemento de primer orden en la semántica de la lógica. Uno de los principales motivos de su creación fue el facilitar la descripción de relaciones de causalidad entre los diversos componentes de un requisito temporal. De esa forma, podremos condicionar unos requisitos al cumplimiento de otros, enriqueciendo la expresividad de nuestras fórmulas.

Por tal motivo, la sintaxis habitual de una fórmula temporal en la lógica LTCS obedecerá al siguiente patrón:

Premisa $\Longrightarrow_{\otimes}$ **Consecuencia**

es decir, una consecuencia sujeta al cumplimiento de una premisa. Si la premisa se cumple en el estado actual, la consecuencia deberá ser cierta en los estados que determine el operador temporal $\otimes$. En caso de aparecer sólo la consecuencia, se entenderá que la premisa es la constante $True$, que se cumple o es cierta en todo momento.

6.1.1. Eventos

Los eventos son los elementos básicos con los que trabaja esta lógica. Constituyen las afirmaciones elementales que las fórmulas de la lógica realizan sobre los distintos sucesos que pueden acaecer en un determinado instante de tiempo.

En nuestro caso, esas afirmaciones elementales se referirán a las posibles evoluciones del sistema en desarrollo. Una fórmula de la lógica establecerá una serie de afirmaciones sobre la posibilidad o necesidad de que en determinados estados del sistema sean posibles ciertos eventos y sus correspondientes transiciones a otros estados. Los constructores de la lógica permiten relacionar distintos eventos a lo largo del tiempo, estableciendo ciertas vinculaciones causa-efecto que se deben mantener en los distintos estados por los que pasa el sistema.

Para poder estudiar las transiciones del sistema y sus propiedades mediante la lógica es necesario dotar a ésta de los mecanismos apropiados para establecer afirmaciones lo más expresivas posible.

El primer paso en la definición de esa expresividad se deriva directamente de la sintaxis de los elementos cuya existencia queremos comprobar: los eventos simbólicos del STS que modela el sistema. Al igual que éstos, los eventos de la lógica deben permitir describir el flujo de información que llevan asociada. Por tanto, el segundo componente de un evento de la lógica (el primero es su identificador) será un patrón que, siguiendo la misma sintaxis que en la sección 5.2.1, nos permita realizar aseveraciones acerca de la clase de información que un evento simbólico del STS lleva asociada.

El tercer elemento de la sintaxis de un evento de la lógica será una condición que debe ser cierta siempre. Esta condición, que podrá involucrar a las variables

ligadas en éste y en anteriores eventos, nos permitirá establecer invariantes acerca de los valores de los datos que se intercambian. Su sintaxis será la de una expresión encerrada entre corchetes.

El cuarto y último elemento de un evento será otra expresión que, en este caso, debe entenderse como una acotación de nuestro interés por la verificación del evento. Este campo establecerá las condiciones en las cuales nos interesa comprobar que el evento es posible. Fuera de esas condiciones no nos importa el resultado. Su sintaxis será la de una expresión encerrada entre llaves.

Todos los campos de un evento, excepto su identificador, son opcionales.

En suma, la sintaxis de un evento será:

$$e_s := i_s \; patron \text{ ``[''} \; expresion \text{ ``]'' ``\{''} \; expresion \text{ ``\}''}$$

Algunos ejemplos de eventos válidos son:

- *a !3* $\longrightarrow$ El evento a ofrece al exterior el número 3.
- *b ?x:int [x>0]* $\longrightarrow$ El evento b es posible y acepta del entorno un entero que se almacenará en la variable x. Ese entero debe ser siempre mayor que cero.
- *c ?y:int {y>0}* $\longrightarrow$ El evento c es posible y acepta del entorno un entero que se almacenará en la variable y. Sólo nos interesa lo que ocurra con los valores estrictamente positivos de y. Para los posibles valores negativos de y (o cero) no nos importa si el evento c es o no posible.

La sintaxis del elemento *patron* está basada en la del campo equivalente del lenguaje LOTOS:

Patron := *"?" Variable*
| *"!" Expresion*
| *"(" Patron ("," Patron)+ ")"*
| *Patron":"Tipo*

Por su parte, en el tipo *Expresion* hemos incluido los constructores habituales que se emplean en los lenguajes de expresiones, enriqueciéndolo con

algunos elementos propios de nuestro formalismo. En concreto, una expresión puede tener la siguiente estructura:

Expresion := *Constante*
| *Variable*
| *Funcion "(" [Expresion ("," Expresion)*] ")"*
| *Expresion Operacion Expresion*
| *Operacion Expresion*
| *"(" Expresion ")"*
| "∃" "{" *Variable* ("," *Variable*)* "}" "(" *Expresion* ")"
| "∀" "{" *Variable* ("," *Variable*)* "}" "(" *Expresion* ")"
| *"[" Expresion "]" "(" Expresion ")"*

Constante = ∅ | *REC "(" string ")"* | *TConstante*

Funcion, Operación, Variable := string

Las constantes ∅ y REC son propias de nuestro formalismo y serán explicadas en las secciones 8.3 y 8.6.1 respectivamente.

TConstante representa a todas las constantes propias de los tipos de datos que se empleen: $True$, $False$, una cadena, un entero, etc.

Las funciones y operaciones serán aquellas que se implementen en los tipos de datos que se empleen: $+$, $/$, $\geq$, $\wedge$, $\neg$, log, etc.

Se incluyen los habituales cuantificadores existencial ($\exists$) y universal ($\forall$) sobre los valores de las variables libres contenidas en las expresiones (sección 6.1.3.1).

Por último, ($[exp_1](exp_2)$) representa un nuevo elemento de nuestro formalismo, empleado para representar la acotación del escenario de interés antes mencionada. Sus reglas de evaluación serán presentadas en la sección 8.3.

6.1.2. Estructura de un requisito

El objetivo principal de la **LTCS** es permitir el establecimiento de requisitos sobre las ordenaciones temporales de los eventos de un sistema. Estos requisitos, individualmente o mediante combinaciones, formarán las propiedades que pretendemos verificar sobre el sistema en desarrollo.

Para ello, definiremos la estructura de un requisito, que nos permitirá declarar los distintos elementos de que se compone, entre ellos la fórmula que lo implementa:

REQ *id* $\{e_1{:}t_{e1}, ...\}$ $(p_1{:}t_{p1}, ...)$ [**OF** *tipo*] **IS**

fórmula en la lógica LTCS

[**DESC** *descripción concisa*]

ENDREQ

Como vemos, esta estructura contiene una serie de palabras clave (**REQ, OF, IS, DESC** y **ENDREQ**) que permiten declarar de forma ordenada los campos del requisito:

- **id**. El nombre del requisito. A partir de este nombre podremos declarar propiedades que combinen varios requisitos mediante instanciaciones y recursiones.
- $\mathbf{e_i{:}t_{ei}}$. Los eventos que intervienen en la fórmula del requisito, junto con el tipo de la información que intercambian con el entorno.
- $\mathbf{p_i{:}t_{pi}}$. Los parámetros del requisito, junto con su tipo. Estos parámetros nos permitirán no sólo generar propiedades de significado variable, sino poder modificar éste de forma dinámica dentro de una misma propiedad.
- **tipo**. El tipo al que pertenece el requisito. Se empleará para clasificar las propiedades de una forma homogénea, en función de sus objetivos.
- La fórmula LTCS que implementa el requisito.
- Una descripción, opcional, en lenguaje natural.

6.1.3. Las fórmulas LTCS

El núcleo expresivo fundamental de la **LTCS** serán las fórmulas lógicas, que estarán compuestas por combinaciones de los siguientes constructores:

$$\text{F} := \text{True} \mid \text{False} \mid \; e_v \; \mid \; \neg F \; \mid (\text{F}) \mid \; \text{F1} \wedge \text{F2} \; \mid \; \text{F1} \vee \text{F2} \; \mid \\ \text{F1} \Longrightarrow\otimes [\text{Æ}\{\vec{v}\}]\ \text{F2} \; \mid \; \text{R}\ \{\vec{e}\}\ (\vec{p})$$

Además de las habituales constantes y conectivas lógicas, podemos encontrar eventos (e_v), según la sintaxis definida en la sección 6.1.1. La constante $True$, siguiendo la sintaxis de los eventos simbólicos, podrá ir acompañada de dos expresiones: una para establecer invariantes y otra para restringir el escenario de nuestro interés.

Mediante la instanciación de la cabecera de un requisito previamente definido, R $\{\vec{e}\}(\vec{p})$, podemos enlazar unos requisitos con otros o declarar propiedades recursivas.

El símbolo $\otimes$ representa los operadores temporales de la lógica, que son tres:

- "$\bigcirc$". Se empleará para especificar que la fórmula que le sucede debe verificarse en los estados de aplicabilidad inmediatamente posteriores al actual. Los estados de aplicabilidad (sección 6.1.3.2) se derivan a partir de aquellos que se alcanzan a través de un evento especificado como posible en la premisa.
- "$\bigodot$". A través de este símbolo expresamos que la fórmula que le acompaña debe verificarse en los estados previos al actual.
- " ". La ausencia de operadores (el tercero) implica la necesidad de verificación en el estado actual, aquel en el que se verifica la premisa.

El operador $\bigcirc$ establece un ámbito universal para la verificación del requisito que le sucede. Es decir, el citado requisito debe verificarse en todos los estados de aplicabilidad del operador. Si no hay estados de aplicabilidad, el requisito se cumple.

Además de los operadores temporales mencionados, vamos a definir otros dos que, aunque derivables a partir de los anteriores, nos ayudarán a conseguir fórmulas más compactas y claras. Son:

- "ⓔ". Este operador restringe la necesidad de verificación en el futuro a un ámbito existencial, es decir, a uno sólo de los estados de aplicabilidad. Si no hay estados de aplicabilidad, el requisito no se cumple. Por tanto:

  $$ⓔR \equiv \neg\bigcirc\neg R$$

- "@". Este operador mantiene el ámbito universal de $\bigcirc$, aunque se diferencia de éste en que exige que exista algún estado de aplicabilidad: si no hay estados de aplicabilidad, el requisito no se cumple. Por tanto:

  $$@R \equiv ⓔ true \wedge \bigcirc R$$

6.1.3.1. Cuantificadores

El empleo de un formalismo lógico que permite razonar sobre el paso de valores entre entidades al producirse las transiciones en el sistema, implica la aparición en el proceso de verificación de fórmulas abiertas. Estas fórmulas abiertas o abstracciones contienen variables libres cuyo valor le ha sido asignado previamente y, por tanto, no son autocontenidas, al depender de información externa a la propia fórmula.

Para tratar con este grado de indeterminación, introduciremos los habituales cuantificadores existencial ($\exists$) y universal ($\forall$). Estos cuantificadores, representados por el símbolo Æ en el conjunto de constructores antes presentado, nos permitirán especificar el ámbito de verificación para los valores de las variables que acepten datos al ocurrir un evento.

Si x es una variable que toma un valor en el requisito R_1:

- $R_1 \Longrightarrow \otimes \exists\{x\}(R)$

  Si se verifica R_1, existe al menos un valor de x para el que se verifica R en los estados de aplicabilidad que determine $\otimes$. Por ejemplo:

  $$a\ ?x{:}int \Longrightarrow ⓔ\ \exists\{x\}\ b\ !(x{+}1)$$

Si en el estado actual se puede producir el evento a, que acepta un entero del entorno, existe al menos un valor de ese entero para el que, en alguno de los estados a los que se puede transicionar, es posible el evento b, que ofrece al entorno el sucesor de ese entero.

- $R_1 \Longrightarrow \otimes \forall\{x\}(R)$

  Si se verifica R_1, el requisito R se verifica para todos los valores de x en los estados de aplicabilidad que determine $\otimes$. Por ejemplo:

  $$a\ ?x{:}int \Longrightarrow \bigcirc\ \forall\{x\}\ b\ !(x{+}1)$$

  Si en el estado actual se puede producir el evento a, que acepta un entero del entorno, para todos los valores de ese entero y en todos los estados a los que se puede transicionar, es posible el evento b, que ofrece al entorno el sucesor de ese entero.

Generalizaremos este procedimiento para especificar conjuntos de variables ($\vec{x}$) en lugar de variables aisladas (x).

6.1.3.2. Estados de aplicabilidad

Los estados de aplicabilidad para un operador de futuro se derivan directamente del requisito de la premisa. El tipo de operador de futuro ($\bigcirc$, ⓔ ó @) determinará en qué estados (de entre los estados de aplicabilidad) debe verificarse la consecuencia.

A continuación, identificamos los estados de aplicabilidad en función del tipo de requisito que forma la premisa:

- **True** $\longrightarrow$ Todos los estados inmediatamente posteriores, no importa cual sea el evento que etiqueta la transición.

  Si el operador es $\bigcirc$ ó @, la consecuencia debe verificarse en todos los estados de aplicabilidad. Si se trata de ⓔ, es suficiente con que la consecuencia se verifique en uno de ellos.

- **False** $\longrightarrow$ Ninguno.

- $\neg\ \mathbf{e_v}$ $\longrightarrow$ Todos, igual que en **True**.

- $\mathbf{e_v} \longrightarrow$ Los estados futuros cuya transición vaya etiquetada con un evento e_v.

  Si el operador es $\bigcirc$ ó @, la consecuencia debe verificarse en todos los estados de aplicabilidad. Si se trata de Ⓔ, es suficiente con que la consecuencia se verifique en uno de ellos.

- $\mathbf{R_1} \vee \mathbf{R_2} \longrightarrow$ Los estados de aplicabilidad de R_1 y los de R_2.

  Si el operador es $\bigcirc$ ó @, la consecuencia debe verificarse en todos los estados de aplicabilidad que se deriven de R_1 o en todos los que se deriven de R_2.

  Si el operador es Ⓔ, la consecuencia debe verificarse en uno de los estados de aplicabilidad que se deriven de R_1 o en uno de los derivados de R_2.

- $\mathbf{R_1} \wedge \mathbf{R_2} \longrightarrow$ Los estados de aplicabilidad de R_1 y los de R_2.

  Si el operador es $\bigcirc$ ó @, la consecuencia debe verificarse en todos los estados de aplicabilidad que se deriven de R_1 y en todos los que se deriven de R_2.

  Si el operador es Ⓔ, la consecuencia debe verificarse en uno de los estados de aplicabilidad que se deriven de R_1 y en uno de los derivados de R_2.

- $\mathbf{R_1} \Longrightarrow_{\otimes} \mathbf{R_2} \longrightarrow$ Todos, igual que en **True**.

- $\mathbf{R}\{\vec{e}\}(\vec{p}) \longrightarrow$ Aquellos que se deriven de la fórmula de R.

En algunos casos, la interpretación de los estados de aplicabilidad podría ser distinta, puesto que existen varias lecturas que, intuitivamente, podrían ser razonables. Simplemente, cada interpretación derivaría en una lógica distinta, con su correspondiente expresividad.

La elección descrita, además de resultar intuitiva, responde a un criterio de maximización de la diversidad expresiva de las fórmulas sencillas. Es decir, allí donde existían varias opciones intuitivamente correctas, se elegía después de descartar aquellas posibilidades equivalentes a otras fórmulas sencillas ya definidas. De esta forma se ofrece al especificador el máximo de posibilidades para, con fórmulas sencillas, distinguir comportamientos similares.

Por ejemplo, supongamos la siguiente fórmula:

$$(R_1 \vee R_2) \Longrightarrow_{\bigcirc} R$$

Podríamos decidir que, en caso de que ambos requisitos fuesen ciertos, la consecuencia debería verificarse en todos los estados de aplicabilidad de R_1 y en todos los de R_2. Pero, en ese caso, la fórmula sería completamente equivalente a la siguiente:

$$(R_1 \Longrightarrow \bigcirc R) \wedge (R_2 \Longrightarrow \bigcirc R)$$

6.2. Especificación de propiedades con LTCS

El universo de propiedades que podemos formular a través de la lógica LTCS abarca propiedades de muy diversa índole y complejidad.

Los algoritmos y mecanismos de verificación de propiedades que nos ocuparán en la siguiente fase deben ser aplicables a cualquier propiedad descrita mediante esta lógica, siempre que la semántica de la propiedad sea coherente. Sin embargo, en aras de lograr una herramienta que facilite la tarea del diseñador y lo guíe en su trabajo, conviene identificar conjuntos de propiedades que manifiesten características comunes para, ulteriormente, crear mecanismos de confección automática de propiedades a partir de la semántica de la propiedad. Es decir, que el diseñador sólo tenga que elegir el modelo general de la propiedad y especificar ciertos parámetros que caractericen su caso particular.

Este procedimiento presenta la ventaja adicional de dispensar al diseñador del conocimiento de la lógica que subyace en el procedimiento de especificación y verificación de propiedades. En caso de estimar oportuno el cambio de la lógica temporal subyacente, bastaría con reescribir los algoritmos de verificación para la nueva lógica y las traducciones de los modelos de cada propiedad a la nueva sintaxis. Si la expresividad de la nueva lógica es suficiente, los usuarios de la herramienta no tendrían por qué apreciar el cambio.

Siguiendo los esquemas habituales de caracterización de propiedades, podemos establecer dos grandes grupos: finalidades e invarianzas.

Finalidades Las finalidades son, en general, propiedades de viveza. Están fundamentalmente orientadas a describir el hecho de que un sistema, en la evolución natural de su comportamiento, pueda alcanzar un determinado estado o ejecutar ciertos eventos. En definitiva, el concepto de finalidad

es el que manejamos cuando nos preguntamos si el sistema será capaz, en un futuro, de ejecutar cierto evento (o combinación de eventos), si eso sucederá de forma inevitable o si depende del camino elegido por el sistema, etc.

Invarianzas Las invarianzas, por contra, son propiedades de seguridad. En general, hablamos de una invarianza cuando expresamos el hecho de que el sistema nunca alcanzará un estado que no deseamos. Nos sirven para expresar el hecho de que ciertos eventos (o combinaciones de eventos) nunca podrán tener lugar en el devenir del sistema, sea cual sea su evolución en el futuro. O también para todo lo contrario, para garantizar que cierto evento siempre es posible en cualquier momento del tiempo, no importa por donde evolucione el sistema.

En esta sección vamos a examinar cómo se traducen los conceptos de finalidad e invarianza en propiedades de uso frecuente (que representen hechos cuya verificación se aborda de forma habitual). Veremos la sintaxis que presenta en cada caso un requisito definido mediante la lógica LTCS y los parámetros con los que hay que rellenar el modelo de la propiedad.

En general, las finalidades e invarianzas tienen dos posibles enfoques: el existencial y el universal. En el primero, se exige que exista, al menos, un camino en el que se verifique un determinado requisito. En el enfoque universal se exige que el requisito se verifique en todos los posibles caminos del sistema. El enfoque existencial es, por tanto, una relajación del universal y, por ello, suele hablarse de propiedades de viveza o seguridad débiles [Sti96].

6.2.1. Finalidades

6.2.1.1. Existencial

Una *finalidad existencial* se emplea para expresar el hecho de que cierto estado se alcanza en alguna de las posibles evoluciones temporales del sistema. Es decir, entre los caminos que puede seguir el sistema, hay al menos uno a través del cual se alcanza el estado deseado.

Ese estado deseado lo identificaremos mediante un requisito temporal (R_1). Es decir, será un estado en donde se verifique el requisito, ya sea la posibilidad

de que ocurra un evento o cualquier propiedad más compleja representable mediante la lógica LTCS. Su modelo será:

$$\mathrm{R} = R_1 \vee ⓔ\mathrm{R}$$

Por ejemplo, si hemos transmitido una trama (cuyo número es x) y queremos expresar el hecho de que, en algún caso, *"es posible la recepción de una notificación de error"*, tendríamos que elegir el modelo *finalidad existencial* y especificar que el requisito R_1 es el evento *nack*:

$$\mathrm{R} = \textit{nack}\ ?y{:}\textit{int}\ \{y{=}x\} \vee ⓔ\mathrm{R}$$

El empleo de la acotación de interés establece que queremos comprobar que la variable y puede tomar el mismo valor que la variable x. El requisito no prohibe que tome otros valores, aunque eso no nos interesa.

6.2.1.2. Existencial con persistencia

Con el concepto de *persistencia* denotamos el hecho de que la finalidad se cumple de forma reiterada. Es decir, una vez cumplida se volverá a cumplir, y así indefinidamente.

En este caso, *finalidad existencial con persistencia*, cierto requisito R_1 se cumplirá al menos por un camino y, después de cumplirse, se volverá a cumplir al menos por un camino de entre todos los posibles.

$$\mathrm{R} = (R_1 \wedge ⓔ\mathrm{R}) \vee ⓔ\mathrm{R}$$

Por ejemplo, la temperatura del sistema es la adecuada o, por algún camino, es posible que logremos que lo sea. Cuando lo consigamos, no perderemos esa capacidad.

$$\mathrm{R} = (True\ [\textit{temp} \leq TMAX] \wedge ⓔ\mathrm{R}) \vee ⓔ\mathrm{R}$$

6.2.1.3. Existencial con persistencia condicional

Una variante posible del tipo anterior consistiría en especificar que, una vez alcanzado un estado donde se cumple R_1, volverá a cumplirse si se verifica una cierta condición R_2 (p.e. si evolucionamos por algún camino en particular) sin importar lo que sucede si no se verifica esa condición. A esto lo denominaremos *persistencia condicional*

$$\mathrm{R} = (R_1 \wedge (R_2 \Longrightarrow ⓔ\mathrm{R})) \vee ⓔ\mathrm{R}$$

Por ejemplo, por algún camino es posible realizar una transmisión y, si lo hacemos, existe algún camino por el cual la situación se vuelve a repetir.

$$\mathrm{R} = (tx\ !data \wedge (tx\ !data \Longrightarrow ⓔ\mathrm{R})) \vee ⓔ\mathrm{R}$$

6.2.1.4. Universal

Una *finalidad universal* expresa el hecho de que el sistema alcanzará cierto estado en cualquier caso. Es decir, no importa por donde evolucione, tarde o temprano, terminará alcanzando el estado deseado (representado, como antes, por el requisito R_1 que debe cumplirse en ese estado).

$$\mathrm{R} = R_1 \vee @\mathrm{R}$$

Por ejemplo, si el sistema debe retransmitir tramas de datos y ha recibido la número x y queremos comprobar que en cualquier caso, *"en un futuro se retransmitirá esa trama"*, sólo tendríamos que elegir el modelo *finalidad universal* y especificar que el requisito R_1 es el evento rtx correspondiente:

$$\mathrm{R} = rtx\ !x \vee @\mathrm{R}$$

6.2.1.5. Universal con persistencia

En el caso de una *finalidad universal con persistencia* expresamos que cierto estado se alcanzará vayamos por donde vayamos y que, una vez alcanzado

ese estado, se volverá a alcanzar en el futuro sin importar por donde evolucione el sistema.

$$\mathrm{R} = (R_1 \wedge @\mathrm{R}) \vee @\mathrm{R}$$

Por ejemplo, si nuestro escenario es el de una red con paso de testigo y deseamos verificar que, pase lo que pase, *"dispondremos del turno de transmisión de forma reiterada"*, podríamos utilizar el requisito:

$$\mathrm{R} = (rx\ ?x\ \{x{=}miturno\} \wedge @\mathrm{R}) \vee @\mathrm{R}$$

6.2.1.6. Universal con persistencia condicional

Igual que en el caso existencial, una posible variante consistiría en establecer que, una vez alcanzado el estado deseado, la finalidad se volverá a cumplir si se verifica una condición.

$$\mathrm{R} = (R_1 \wedge (R_2 \Longrightarrow @\mathrm{R})) \vee @\mathrm{R}$$

Por ejemplo, en el caso anterior, sólo se nos garantiza volver a recibir el turno de transmisión si no hemos alcanzado un máximo.

$$\mathrm{R} = (rx\ ?x\ \{x{=}turno\} \wedge (True\ [NOMAX] \Longrightarrow @\mathrm{R})) \vee @\mathrm{R}$$

6.2.2. Invarianzas

6.2.2.1. Universal

El modelo general de una invarianza se emplea para especificar que el sistema cumple una determinada propiedad en todos sus estados o, lo que es lo mismo, nunca alcanzará un estado en donde se incumple la propiedad, no importa por donde evolucione.

Su modelo más sencillo sería el siguiente, donde establecemos que el requisito R_1 debe cumplirse siempre, en todos los estados que alcance el sistema, sea cual sea su evolución.

$$\mathrm{R} = R_1 \wedge @\mathrm{R}$$

Por ejemplo, la temperatura del sistema siempre está por debajo del máximo:

$$\mathrm{R} = read_t\ ?t{:}Temp\ [t<TMAX] \wedge @\mathrm{R}$$

Si deseáramos verificar la ausencia de bloqueo, podríamos emplear el siguiente requisito:

$$\mathrm{R} = True \wedge @\mathrm{R}$$

6.2.2.2. Universal no reactiva

En caso de que el sistema no sea reactivo (puede llegar a un bloqueo en su funcionamiento normal) y sólo nos interese sus propiedades mientras esté funcionando, eliminaríamos del anterior arquetipo la necesidad de que siempre exista algún estado futuro.

$$\mathrm{R} = R_1 \wedge \bigcirc\mathrm{R}$$

6.2.2.3. Universal condicional

En caso de que queramos especificar que la invarianza no debe verificarse en todos los futuros, sino sólo en aquellos que se deriven de ciertas transiciones, emplearemos la siguiente derivación:

$$\mathrm{R} = R_1 \wedge (R_2 \Longrightarrow @\mathrm{R})$$

6.2.2.4. Existencial

En el caso de una *invarianza existencial*, expresamos el hecho de que existe un camino a través del cual siempre es verdadero un determinado requisito R_1.

$$R = R_1 \wedge ⓔR$$

También podríamos definir la *invarianza existencial condicional.*

$$R = R_1 \wedge (R_2 \Longrightarrow ⓔR)$$

Análogamente, podríamos definir la *invarianza existencial no reactiva*, en caso de que el sistema no sea reactivo.

$$R = R_1 \wedge (ⓔTrue \Longrightarrow ⓔR)$$

6.2.2.5. Contigüidad

Un caso particular de invarianza sería el de la *contigüidad* entre dos requisitos R_1 y R_2. Es decir, lo que tratamos de expresar en este caso es el hecho de que si en un estado se verifica un cierto requisito R_1, en los estados inmediatamente posteriores debe verificarse otro requisito R_2.

$$R = (R_1 \Longrightarrow \bigcirc R_2) \wedge @R$$

Por ejemplo, en un *router* debe ser posible encaminar un paquete tras haberlo recibido.

$$R = (rx\ ?x \Longrightarrow \bigcirc\ \forall\ \{x\}(tx\ !x)) \wedge @R$$

En caso de sistemas no reactivos, podríamos definir la *contigüidad no reactiva*:

$$R = (R_1 \Longrightarrow @R_2) \wedge \bigcirc R$$

6.2.3. Precedencias

Por *precedencias* entendemos propiedades que expresan una relación temporal directa entre ciertos requisitos (normalmente eventos).

6.2.3.1. Habilitación

Por *habilitación* expresamos el hecho de que el cumplimiento de un cierto requisito R_1 permite el cumplimiento de otro R_2. Es decir, R_2 no puede satisfacerse hasta que se satisfaga R_1. Una vez que se satisface R_1 en un estado, R_2 puede satisfacerse o no en ese estado y en los estados futuros.

$$\mathrm{R} = R_1 \vee (\neg R_2 \wedge @\,\mathrm{R})$$

Y la *habilitación no reactiva*:

$$\mathrm{R} = R_1 \vee (\neg R_2 \wedge \bigcirc \mathrm{R})$$

Por ejemplo, en una puerta de apertura retardada, podríamos comprobar que realmente cumple su principal característica.

$$\mathrm{R} = True\ [hora \geq H] \vee (\neg abrir \wedge @\,\mathrm{R})$$

6.2.3.2. Inhibición transitoria

En este caso, expresamos que la habilitación debe producirse tarde o temprano. Es decir, el requisito R_2 no puede estar inhibido indefinidamente. Recurrimos a un requisito compuesto:

$$R_a = R_1 \vee (\neg R_2 \wedge @\,R_a)$$

$$R_b = R_1 \vee @\ R_b$$

$$\mathrm{R} = R_a \wedge R_b$$

6.2.3.3. Inhibición continua

En este caso, la relación del apartado anterior es permanente. Es decir, un requisito R_2 no puede satisfacerse nunca si no se satisface en ese mismo estado otro requisito R_1. Es un caso particular de invarianza.

$$\mathrm{R} = (R_1 \vee \neg R_2) \wedge @\mathrm{R}$$

Y la *inhibición continua no reactiva*:

$$\mathrm{R} = (R_1 \vee \neg R_2) \wedge \bigcirc\mathrm{R}$$

Por ejemplo, si quisieramos comprobar el hecho de que *"no es posible enviar tramas si no es nuestro turno"*, emplearíamos el requisito:

$$\mathrm{R} = (True\ [miturno] \vee \neg tx_trama) \wedge @\mathrm{R}$$

6.2.3.4. Precedencia Inicial

Para expresar el hecho de que R_1 debe preceder estrictamente a R_2 (para que se satisfaga R_2 es preciso que se haya satisfecho R_1 en algún estado previo) recurriríamos al siguiente modelo:

$$\mathrm{R} = \neg R_2 \wedge (R_1 \vee \bigcirc\mathrm{R})$$

Por ejemplo, si quisieramos comprobar el hecho de que *"no es posible recibir la confirmación de la segunda trama si no se ha enviado"*, emplearíamos el requisito:

$$\mathrm{R} = \neg ack\ ?x{:}int\ \{x{=}2\} \wedge (trama\ !2 \vee \bigcirc\mathrm{R})$$

Capítulo 7

Solapamiento de las trazas iniciales

7.1. Introducción

Siguiendo el procedimiento descrito en el capítulo 4, tras elegir el conjunto de eventos que vamos a utilizar para describir el comportamiento del sistema, procederemos a especificar las trazas o secuencias de actuación que creemos que deben ser posibles en su dinámica.

Una vez especificadas las trazas iniciales, es necesario someterlas a un proceso de solapamiento que elimine redundancias y unifique comportamientos parcialmente equivalentes. Este proceso dará como resultado un sistema de transiciones simbólico que recogerá todos los comportamientos descritos mediante las trazas.

El árbol simbólico resultante del proceso de solapamiento será empleado en el capítulo 8 como la representación del sistema sobre la cual estudiaremos el cumplimiento de las propiedades de interés, especificadas mediante la lógica temporal LTCS que hemos descrito en el capítulo 6.

En caso de ser necesaria la modificación del árbol inicial, el proceso de adición de nuevas trazas también podría recurrir a este procedimiento de solapamiento para integrar los nuevos comportamientos.

En este capítulo abordamos esos dos objetivos: la especificación de las

trazas de comportamiento conocidas y su unificación en un sistema de transiciones simbólico.

Además de este enriquecimiento del árbol simbólico mediante el solapamiento de nuevos comportamientos, también es posible la agregación directa de una traza en un determinado nodo del árbol, sin tener en cuenta las restantes transiciones de ese nodo. Este proceso, que no envuelve complejidad alguna, puede derivar en la existencia de varias transiciones equivalentes a partir del citado nodo. Esto introduce un grado de indeterminación en el comportamiento del sistema, habitual en este tipo de mecanismos, que permite modelar su dinámica sin descender en el detalle, quizás inaccesible al nivel de abstracción en el que nos encontramos.

7.2. Especificación de trazas mediante LTCS

Para especificar las trazas de comportamiento precisamos un lenguaje o mecanismo que permita describir la sucesión ordenada de los eventos (junto con la información que ofrecen o aceptan y las restricciones que pueda tener cada transición), la conjunción de varios eventos, referenciar comportamientos definidos previamente, etc.

En nuestra implementación del procedimiento descrito, vamos a emplear la lógica temporal **LTCS**, definida en la sección 6.1. Esta lógica, pensada para especificar requisitos temporales que debe cumplir un sistema, puede servirnos para describir las trazas iniciales si entendemos que la secuencia de eventos descrita por una traza es, precisamente, un requisito que imponemos al comportamiento del sistema. Además, aunque no ofrezca ventajas respecto a lenguajes descriptivos más simples, sí simplifica la implementación de una herramienta al reutilizar tipos de datos abstractos y analizadores léxico-sintácticos creados para las tareas de verificación.

Sin embargo, toda la capacidad expresiva de la lógica LTCS no será necesaria para este propósito. Para la confección de las trazas iniciales, emplearemos solamente un subconjunto de esta lógica. Este subconjunto será el formado por los constructores:

$$\mathrm{T} := \quad e_v \quad | \quad (\mathrm{T}) \quad | \quad \mathrm{T1} \wedge \mathrm{T2} \quad | \quad \mathrm{T1} \Longrightarrow \bigcirc\, \mathrm{T2} \quad | \quad \mathrm{T}\ \{\vec{e}\}\ (\vec{p})$$

Es decir, una traza podrá consistir en un evento simbólico, una composición secuencial de un evento y una traza, una conjunción de dos trazas (la combinación de dos trazas distintas) o la instanciación de (o recursión a) otra traza.

Los eventos seguirán la sintaxis de la lógica (sección 6.1.1), excepto el último campo de acotación o restricción de interés, que no existirá. Por tanto, estarán compuestos por su identificador, un patrón opcional de intercambio de información con el entorno y una condición (posiblemente involucrando variables libres) que debe ser cierta para que la ocurrencia del evento sea posible y que, por tanto, condicionará la transición.

Por ejemplo, una traza que describa la sucesión de los eventos *recibir*, *anotar* y *transmitir* (donde *recibir* acepta un número entero que luego ofrece *transmitir*), se expresaría como:

$$\text{T} := \text{recibir ?y:int} \Longrightarrow\!\bigcirc\ (\text{anotar} \Longrightarrow\!\bigcirc\ \text{transmitir !y})$$

Claramente, la conjunción no es imprescindible, ya que una traza con una conjunción puede descomponerse en dos trazas distintas. La incluimos, sin embargo, con el propósito de simplificar la tarea de descripción de trazas, y permitir la especificación de una traza un poco más compleja en lugar de varias simples.

Una traza puede emplear varios requisitos para describir el comportamiento deseado (en ese caso, el nombre de la traza será el nombre del requisito principal). Esto puede ser debido a diferentes motivos: por claridad, por estructuración, para aprovechar trazas ya confeccionadas, etc. Esto será una obligación en el caso de trazas recursivas donde la recursión conduzca a algún punto intermedio de la traza. En este caso, lo más sencillo será convertir al punto de recursión en el comienzo de una traza recursiva que se instancia en ese momento. Por ejemplo, en la figura 7.1, la traza **T** utiliza a la traza recursiva **R** para describir el comportamiento del gráfico.

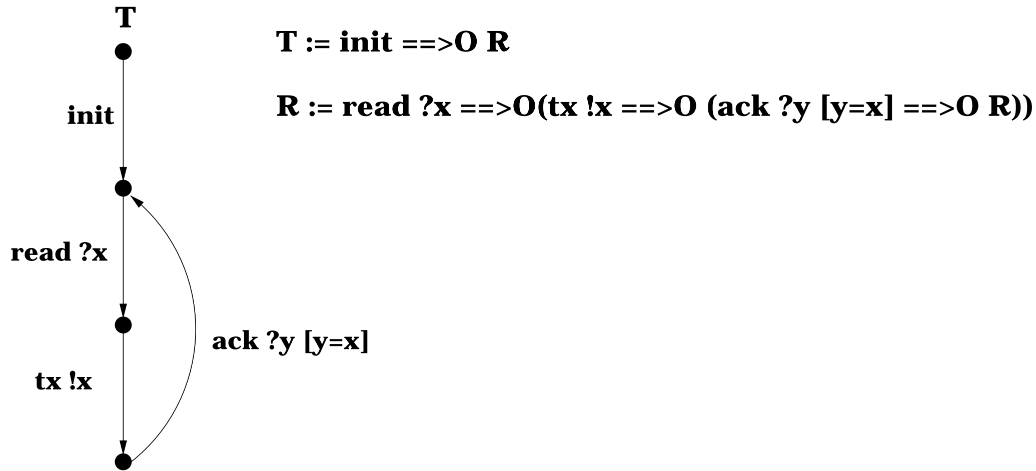

Figura 7.1: Traza recursiva

7.3. Solapamiento de las trazas

En esta sección abordaremos el procedimiento a seguir para solapar una nueva traza sobre el sistema de transiciones simbólico fruto del solapamiento de las trazas previas. Evidentemente, la primera traza no se solapará con nada; simplemente se creará un árbol simbólico que refleje el comportamiento de esa traza.

A grandes rasgos, el procedimiento de solapamiento consiste en añadir transiciones etiquetadas con ciertos eventos en los distintos nodos del árbol existente, junto con el comportamiento posterior que se deriva de cada una de esas transiciones. En algunas ocasiones, el evento que queremos solapar en un determinado nodo será equivalente a otro ya existente en ese nodo. En ese caso, unificaremos ambos eventos en uno sólo, que conducirá tanto al comportamiento habilitado por el evento previo como al ofrecido por el nuevo evento que tratamos de solapar.

7.3.1. Unicidad de las variables

A la hora de proceder al solapamiento de las trazas definidas, hay que evitar posibles confusiones derivadas de coincidencias en los nombres de las

variables. Una traza describe un comportamiento o sucesión de eventos que debe permitir el sistema, incluyendo posibles intercambios de información con el exterior. Este comportamiento debe ser completamente independiente de los nombres de las variables que empleemos para modelar la comunicación con el entorno.

Por tanto, para evitar confusiones derivadas de coincidencias parciales en el espacio de nombres empleado, someteremos a las trazas a un filtrado que nos garantice diversos tipos de unicidad: unicidad de una variable dentro de un requisito, dentro de los requisitos de una traza y dentro del conjunto total de trazas.

Debido a esta unicidad de las variables, lo normal es que necesitemos definir una sustitución de variables (σ) para decidir la equivalencia parcial de dos trazas distintas.

7.3.2. Solapamiento de eventos

En primer lugar veamos el solapamiento de eventos simples, sin guardas que condicionen la transición que etiquetan.

Supongamos que pretendemos solapar una traza consistente en un evento seguido de un comportamiento genérico:

$$T := a_c \; p_1 \Longrightarrow\bigcirc \; T'$$

en un nodo de un árbol en donde, entre otras transiciones, hay una etiquetada con el evento $a_c \; p_2$. Para decidir si se puede unificar el evento a_c de la traza con el evento a_c del árbol, es necesario estudiar la equivalencia de los patrones p_1 y p_2.

Lo más probable es que, en caso de ser equivalentes, lo sean bajo una cierta sustitución de variables σ. Es decir, $p_2 = \sigma(p_1)$. En este caso, puesto que el comportamiento que describe la nueva traza es independiente de las variables empleadas, daremos por unificados los dos eventos a_c en la misma transición del árbol, realizaremos el cambio de variables en el resto de la traza y procederemos a solapar $\sigma(T')$ sobre el nodo del árbol al que conduce la transición del evento a_c.

Si no existe ninguna sustitución de variables σ bajo la cual sean equivalentes

los dos patrones, procederemos a crear una nueva transición a partir de ese nodo y etiquetarla con el evento $a_c\ p_1$. A esa nueva transición le añadiremos el comportamiento de T'.

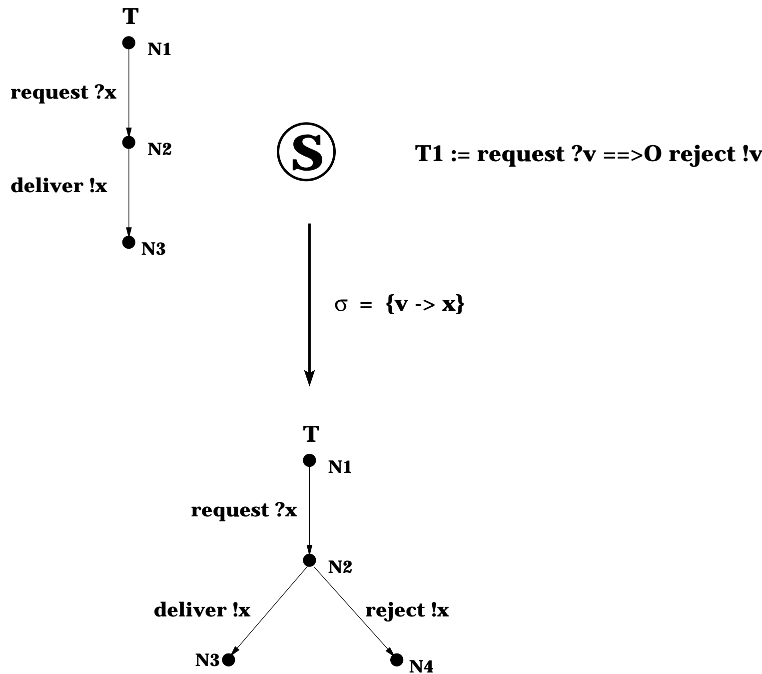

Figura 7.2: Solapamiento simple

En el ejemplo de la figura 7.2, tratamos de solapar la traza **T1** de la derecha en el árbol de la izquierda (la **S** dentro del círculo indica la operación de solapamiento).

Inicialmente, intentamos solapar el evento "*request* $?v$" sobre el nodo N_1. Vemos que existe en ese nodo un evento "*request* $?x$" y que ambos patrones son equivalentes bajo la sustitución $\sigma = \{\mathrm{v} \rightarrow \mathrm{x}\}$. Por tanto, damos por unificado el evento "*request* $?v$" y procedemos a intentar solapar el resto de la traza ($\sigma(reject\ !v) = reject\ !x$) sobre el nodo N_2. En este caso, vemos que es imposible tal solapamiento, puesto que no existe ningún evento *reject* en ese nodo del árbol simbólico y, por tanto, creamos una nueva rama en el árbol para ese evento.

7.3.3. Solapamiento de guardas

El problema del solapamiento de un evento de una traza sobre una transición de un árbol simbólico requiere un procedimiento más refinado en caso de que los eventos a solapar presenten guardas con expresiones que condicionen su viabilidad.

Si queremos unificar las dos transiciones habrá que integrar ambas guardas en una sola que condicione la transición resultante. Además, será necesario trasladar cada una de ellas por separado al resto del comportamiento que condicionaban originalmente.

Por ejemplo, en la figura 7.3, supongamos que queremos solapar la traza de la derecha sobre el nodo N_1 del árbol de la izquierda.

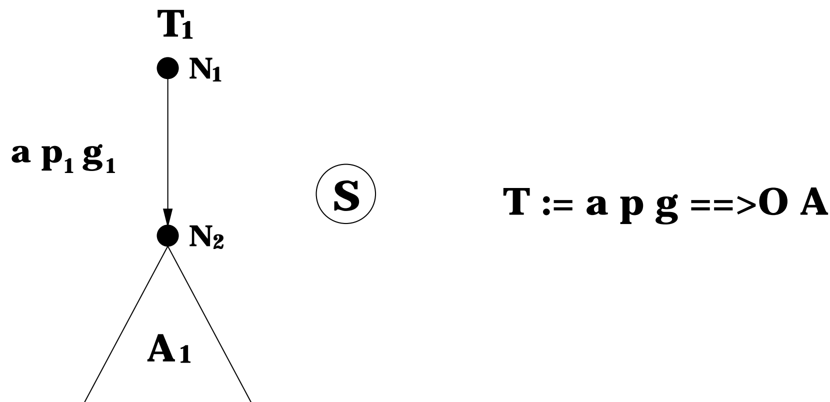

Figura 7.3: Solapamiento de guardas

Si no existe ninguna sustitución σ bajo la cual p sea equivalente a p_1, creamos otra rama en el nodo N_1 que represente el comportamiento de la nueva traza. El resultado sería el de la figura 7.4.

Supongamos ahora que sí existe una sustitución σ_1 tal que $p_1 \equiv \sigma_1(p)$.

Si $\sigma_1(g)$ es equivalente a g_1, entonces podemos dar por solapado el evento "*a p g*" y tratar de solapar el resto de la traza sobre el nodo N_2. A falta de ese solapamiento (lo que indicamos mediante la **S** dentro del círculo entre ambos comportamientos), el resultado sería el de la figura 7.5

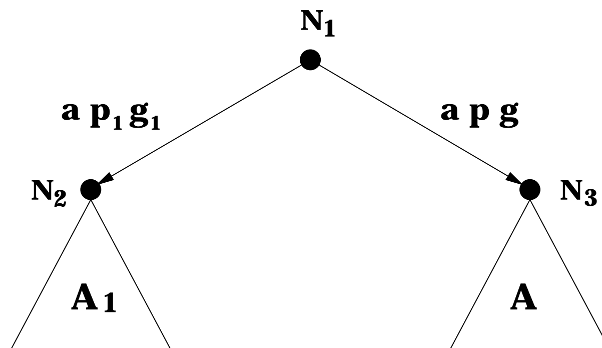

Figura 7.4: Patrones no equivalentes

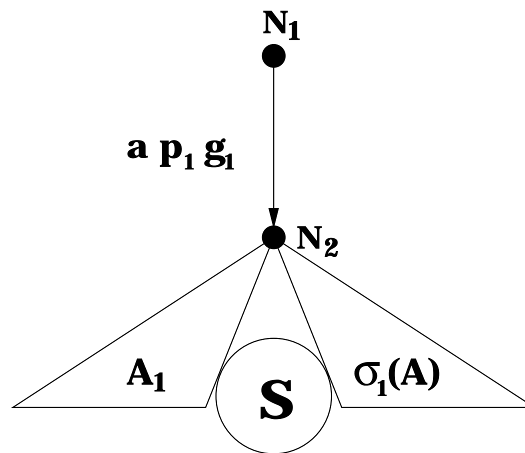

Figura 7.5: Guardas equivalentes

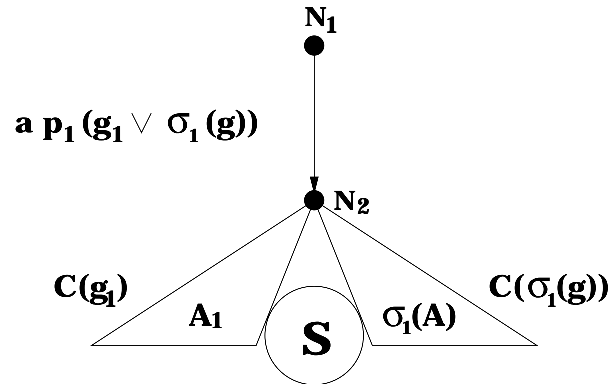

Figura 7.6: Guardas no equivalentes

Si existe la sustitución σ_1 pero $\sigma_1(g)$ y g_1 no son equivalentes, tendremos que transformar la guarda del evento en una disyunción de g_1 y $\sigma_1(g)$. Además, habrá que añadir la guarda g_1 a todos los eventos que sean posibles en A_1 (lo que indicamos en la figura 7.6 con el texto $\mathbf{C}(g_1)$ encima del comportamiento **A1**) y la guarda g a todas los eventos que sean posibles en A (**C(g)** encima del comportamiento **A**). Una vez hecho esto, como se ve en la figura 7.6, trataremos de solapar A_1 y $\sigma_1(A)$.

7.3.3.1. Generalización del procedimiento

La aplicación del procedimiento anterior al caso general en el que existan n transiciones etiquetadas con el mismo evento es directa.

Si queremos solapar una traza T, que comienza con el evento "*a p g*" y continúa con T_c, sobre el nodo N_0, tendremos que observar el conjunto de transiciones que parten de N_0. De esas transiciones, nos quedaremos con las n etiquetadas con eventos "$a\ p_i\ g_i$" (i $\in$ [1..n]) para las cuales existan los correspondientes σ_i tal que $p_i \equiv \sigma_i(p)$.

Si n=0, es decir, de N_0 no parte ninguna transición etiquetada con un evento equivalente al primer evento de la traza que queremos solapar, creamos una nueva rama en el nodo N_0 y añadimos el nuevo comportamiento.

Si existen uno o varios eventos con el mismo identificador y patrones equivalentes al del evento de la traza (bajo la apropiada sustitución σ_i), sometere-

mos a cada uno de ellos a un proceso de solapamiento con “$a\ p\ g$”, seguido del solapamiento de $\sigma_i(T_c)$ en el nodo al que conducen.

En cada caso, la equivalencia entre la guarda del evento de la traza (g) y las posibles guardas (g_i) que condicionen cada transición será tratada según lo descrito en la sección anterior.

7.3.4. Procedimiento general de solapamiento

Como resumen de los anteriores apartados, podemos establecer un procedimiento general para solapar una traza sobre un nodo de un árbol simbólico. Supongamos que queremos solapar la traza **T** en un nodo del que parten un conjunto de transiciones etiquetadas con “$a_i\ p_i\ g_i$”. Según sea **T**:

- T := $a\ p\ g \Longrightarrow \bigcirc$ A

  Supongamos que en el nodo existen m transiciones tal que $a_i = a$.

  1. Si m=0, creamos una nueva rama para el evento de la traza.
  2. Supongamos que de las m transiciones anteriores, para n de ellas existe una σ_i tal que $p_i \equiv \sigma_i(p)$. Si n=0, creamos una nueva transición para el evento y la añadimos al nodo, junto con el comportamiento posterior.
  3. Para cada una de esas n transiciones, si $g_i \equiv \sigma_i(g)$, se da por solapado el evento de la traza en cada transición y procederemos a solapar $\sigma_i(A)$ sobre el nodo al que se transiciona.
  4. Para cada una de las n transiciones en las que g_i no sea equivalente a $\sigma_i(g)$, damos por solapado el evento de la traza en la transición (cambiando la guarda g_i de acuerdo a lo descrito en la sección 7.3.3) y solapamos A_i y $\sigma_i(A)$ (añadiendo a sus eventos la nueva guarda que corresponda en cada caso).

- T:= T1 $\wedge$ T2

  Solapamos por separado sobre ese nodo ambas trazas.

- T := T1 $\{a_1,..,a_j\}$ $(p_1,...,p_k)$

Si se trata de una instanciación, cogemos el cuerpo de la definición de T1 y, tras las oportunas sustituciones de los eventos y parámetros formales por los actuales, procedemos a solaparla sobre el nodo.

7.4. Ejemplo

A continuación vamos a ilustrar el procedimiento descrito en este capítulo mediante un sencillo ejemplo.

El sistema que vamos a describir es un sencillo servidor de cierto tipo de recursos de naturaleza indefinida. El funcionamiento normal del servidor consiste en esperar una solicitud de conexión, la demanda de un recurso que enviará y la confirmación de la recepción correcta de los datos, tras lo cual el servidor liberará la conexión.

Por tanto, la primera traza que conocemos de su comportamiento sería la de su funcionamiento correcto:

$$\begin{array}{l} T := connect\ ?ses \Longrightarrow \bigcirc(confirm\ !ses \Longrightarrow \bigcirc(\\ \quad request\ ?obj \Longrightarrow \bigcirc(\\ \quad\quad answer\ (!OK, !all_data(obj))\ [available(obj)] \Longrightarrow \bigcirc(\\ \quad\quad\quad ack\ ?v\ [v = obj] \Longrightarrow \bigcirc(release\ !ses \Longrightarrow \bigcirc\ T))))) \end{array}$$

El *STS* creado estaría compuesto por una sola secuencia donde se sucederían los eventos que describe la traza.

Vamos a definir otras cuatro trazas conocidas que se corresponden con otros tantos comportamientos donde se producen errores. En $T1$, tras la conexión, vence la temporización sin haber recibido la solicitud del recurso. Se libera la conexión y se inicializa el protocolo.

$$\begin{array}{l} T1 := connect\ ?op \Longrightarrow \bigcirc(confirm\ !op \Longrightarrow \bigcirc(\\ \quad tout \Longrightarrow \bigcirc(release\ !op \Longrightarrow \bigcirc\ T1))) \end{array}$$

Al solapar la traza con el árbol existente, se genera el *STS* reflejado en la parte izquierda de la figura 7.7.

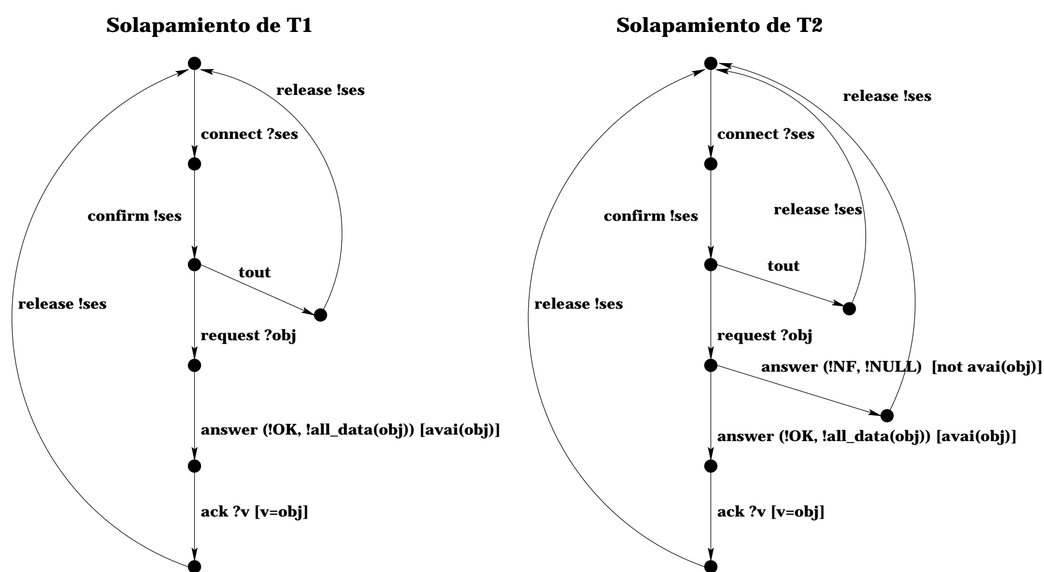

Figura 7.7: Solapamiento de T1 y T2

En la traza $T2$, el recurso solicitado no existe en la base de datos del servidor. Se envía una notificación, se libera la conexión y se inicializa el protocolo.

$$\begin{aligned}
&T2 := connect\ ?con \Longrightarrow\bigcirc(confirm\ !con \Longrightarrow\bigcirc(\\
&\qquad\quad request\ ?rec \Longrightarrow\bigcirc(\\
&\qquad\qquad answer\ (!NF, !NULL)\ [\neg available(rec)] \Longrightarrow \bigcirc(\\
&\qquad\qquad\quad release\ !con \Longrightarrow\bigcirc\ T2))))
\end{aligned}$$

Al solapar la traza, se genera el STS reflejado en la parte derecha de la figura 7.7.

En $T3$, el cliente indica que alguna trama del recurso enviado no llegó correctamente. Se reenvía la trama en cuestión y se vuelve a esperar el resultado de la transmisión.

$$\begin{aligned}
&T3 := connect\ ?link \Longrightarrow\bigcirc(confirm\ !link \Longrightarrow\bigcirc(\\
&\qquad\quad request\ ?item \Longrightarrow\bigcirc(\\
&\qquad\qquad answer\ (!OK, !all_data(item))\ [available(item)] \Longrightarrow \bigcirc\ T31)))
\end{aligned}$$

$$\begin{aligned}
&T31 := nack\ ?nt \Longrightarrow\bigcirc(\\
&\qquad answer\ (!OK, !trama(nt, item)) \Longrightarrow\bigcirc\ T31))
\end{aligned}$$

Al solapar, obtenemos el árbol de la figura 7.8.

En $T4$, tras enviar el recurso solicitado, vence el temporizador sin haber recibido ninguna indicación del resultado de la transmisión. Se envía un requerimiento para que el cliente informe y se vuelve a esperar el resultado de la transmisión.

$$\begin{aligned}
&T4 := connect\ ?serv \Longrightarrow\bigcirc(confirm\ !serv \Longrightarrow\bigcirc(\\
&\qquad\quad request\ ?elem \Longrightarrow\bigcirc(\\
&\qquad\qquad answer\ (!OK, !all_data(elem))\ [available(obj)] \Longrightarrow \bigcirc\ T41)))
\end{aligned}$$

$$T41 := tout \Longrightarrow\bigcirc(\ query\ !elem \Longrightarrow\bigcirc\ T41)$$

Al solapar, obtenemos el árbol de la figura 7.9.

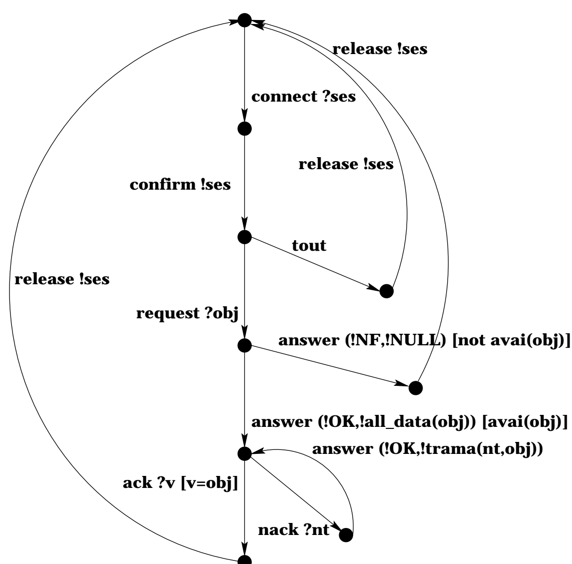

Figura 7.8: Solapamiento de T3

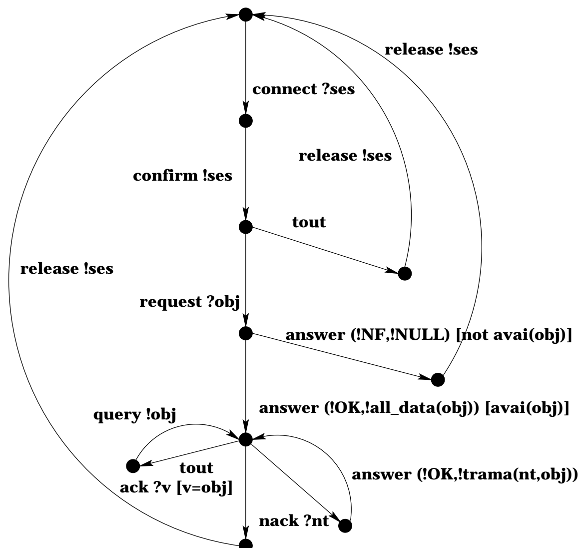

Figura 7.9: Solapamiento de T4

Capítulo 8

Verificación de propiedades temporales

8.1. Introducción

A lo largo de los capítulos anteriores hemos introducido y descrito los elementos que intervienen en el proceso de generación de una representación adecuada del sistema y de las propiedades que deseamos que cumpla éste. Llegado este punto y siguiendo el procedimiento trazado en el capítulo 4, procederemos a formalizar un mecanismo que nos permita decidir el cumplimiento o no de las propiedades.

Este es, en esencia, el principal objetivo de un formalismo de este tipo: la verificación automática de propiedades sobre un modelo del sistema.

Tal como se describió en la sección 3.3, en la verificación automática de las propiedades de un sistema intervienen, fundamentalmente, tres elementos:

- Un modelo computacional que nos permita representar el sistema real con el suficiente grado de fidelidad como para sernos útil. En nuestro caso, un sistema de transiciones simbólico (capítulo 5) ha sido el modelo elegido a tal efecto.

- Un lenguaje de especificación de propiedades que nos sirva para describir aquellas características que deseamos comprobar en el sistema. La lógica

LTCS (capítulo 6) ha sido definida con tal propósito y se ha mostrado como una herramienta muy expresiva para realizar tal especificación.

- Un procedimiento de decisión que, basándose en una relación de satisfacción, nos informe sobre el cumplimiento de las propiedades en el modelo computacional. En la sección 3.3 introducíamos el *model checking* temporal como un mecanismo de gran potencia que permitía tomar ese tipo de decisiones.

Sin embargo, el procedimiento de *model checking* y su interpretación en función de la existencia de una estrategia ganadora para un juego de bisimulación, introducida en el capítulo 3, no es suficiente para nuestros propósitos. Nuestro formalismo difiere notablemente de los allí descritos y citados [Sti96], sobre todo en un aspecto: el modelado del intercambio de información entre el sistema y su entorno.

Los elementos que intervienen en la descripción del sistema y las propiedades que nos interesan se caracterizan por ser simbólicos, utilizando variables para representar conjuntos de valores, quizás de extensión no finita. Por contra, los procedimientos de *model checking* descritos en el capítulo 3 son aplicables, solamente, a sistemas de transiciones etiquetadas. En ellos no hay ningún tipo de intercambio de información en los eventos, condiciones en las transiciones, variables libres en los estados del sistema, etc.

Por tanto, nuestro objetivo en este capítulo será diseñar un algoritmo de *model checking* temporal que tenga en cuenta todas las características de nuestra representación del sistema y de la lógica temporal empleada, la LTCS.

8.2. Relación de satisfacción

El modelo general de verificación del que vamos a partir es el descrito en la sección 3.3.1: la búsqueda de una estrategia ganadora para un jugador que defienda que el sistema cumple la propiedad.

Para ello, nuestra tarea será demostrar que el STS que constituye la descripción actual del sistema es capaz de seguir a la propiedad en todas sus afirmaciones a lo largo de su espacio de estados. Es decir, cuando la propiedad afirme que un evento debe ser posible, el sistema deberá ofrecerlo; cuando

afirme que un evento está prohibido, éste no deberá ser ofrecido por el sistema en ese instante.

La naturaleza simbólica de la representación del sistema y de la lógica empleada implicará que, en general, el hecho de que el sistema pueda seguir a la propiedad dependerá de ciertas condiciones derivadas de las guardas de las transiciones, de los eventos de la propiedad, de los valores asignados a las variables libres, etc. De cada afirmación de la propiedad se desprenderá una condición: aquella que debe cumplirse para que la afirmación que realiza la propiedad sea cierta en ese estado del sistema.

Por tanto, si la propiedad realiza afirmaciones a lo largo de un cierto camino, las distintas condiciones que se deriven de cada afirmación se irán agregando, según los contructores lógicos empleados, para formar una condición final cuya veracidad o falsedad nos indicará el resultado de la verificación. Esta condición la denominaremos **Condición de Verificación** (en adelante **CV**).

Si la propiedad realiza una afirmación en un estado y ésta no es cierta, evidentemente, esa parte de la propiedad no se cumple. Pero además, es posible que las modificaciones que haya que realizar en el sistema para lograr que se cumpla vayan en contra de ciertos invariantes establecidos por propiedades anteriores (por ejemplo, eliminar un evento caracterizado como fijo o añadir otro que ha sido catalogado como prohibido). En ese caso, diremos que el sistema no sólo no cumple esa parte de la propiedad, sino que la rechaza. Es decir, no es posible modificar el sistema para que cumpla esa parte de la propiedad, pues ello iría en contra de ciertos invariantes establecidos por la verificación de propiedades anteriores.

Al igual que antes, las distintas condiciones que implican el rechazo de cada afirmación se agregarán, según los constructores lógicos que articulen la propiedad, para formar una condición final que nos informará del rechazo de la propiedad en su conjunto. La denominaremos **Condición de Rechazo** (en adelante **CR**).

El resultado de nuestro algoritmo de *model checking* temporal será, por tanto, un par de condiciones: **CV** y **CR**. Si CV es cierta, el sistema cumple la propiedad. Si CR es cierta, el sistema rechaza la propiedad. Si ambas son falsas, el sistema no cumple la propiedad pero puede ser modificado para que lo haga. El hecho de que ambas fueran ciertas denotaría una clara incoherencia en el sistema, que ofrece y prohibe simultáneamente un comportamiento.

En la sección 8.7 formalizaremos la construcción de esas dos condiciones. Como es habitual en *model checking*, seguiremos un proceso de inducción estructural sobre las propiedades escritas en la lógica LTCS, definiendo el comportamiento del algoritmo ante todos y cada uno de los elementos que pueden formar parte de una fórmula lógica.

En las próximas secciones, hasta llegar a la 8.7, introduciremos algunos de los elementos que intervienen en la descripción del algoritmo y que resultan imprescindibles para su interpretación.

8.2.1. Simplificaciones

En una primera aproximación al diseño de un algoritmo general realizaremos dos simplificaciones:

1. Emplearemos las negaciones de la lógica sólo sobre los eventos simbólicos. En caso de ser necesario, una sencilla traducción a su forma normal positiva nos proporcionaría una fórmula equivalente que cumple este requisito.

   Esta simplificación no supone una pérdida real de expresividad y sí simplifica notablemente la implementación del algoritmo.

2. A la hora de contemplar los invariantes introducidos por la verificación de propiedades previas, tendremos en cuenta sólo los eventos totalmente fijos o prohibidos.

   Esta simplificación, por contra, sí implica una pérdida de información en el resultado final. El hecho de no contemplar los eventos parcialmente fijos o prohibidos nos impide conocer si el rechazo de una propiedad puede solventarse con una simple reasignación de los invariantes establecidos por otra propiedad o es necesario renunciar a ella completamente.

   Sin embargo, su implementación supone un notable incremento de complejidad del algoritmo descrito en la sección 8.7 por lo que, en una primera aproximación, renunciaremos a este objetivo.

8.3. Una lógica de tres estados

Las condiciones de las que hablamos en la sección anterior tienen su origen, en su mayor parte, en la comparación de un evento simbólico del sistema con un evento de una propiedad.

Una vez que se asignen valores a las variables libres del sistema, el resultado de tal comparación podrá ser $True$ ó $False$. Sin embargo, también podremos llegar a otro resultado distinto de estos. Este tercer resultado, que dará origen a una lógica de tres estados, es fruto de la expresividad de la lógica LTCS.

En la sección 6.1.1 presentábamos la sintaxis de un evento simbólico en la lógica LTCS:

$$e_s := i_s \; \textit{patron} \text{ ``}\texttt{[}\text{''} \; \textit{expresion} \text{ ``}\texttt{]}\text{'' ``}\{\text{''} \; \textit{expresion} \text{ ``}\}\text{''}$$

Los elementos de esta sintaxis determinan las distintas facetas de los eventos simbólicos del sistema sobre los que podemos realizar afirmaciones.

En aquel momento definíamos la segunda expresión (la encerrada entre llaves) como una acotación del escenario en el que nos interesaba llevar a cabo la verificación. Es decir, a través de esa acotación podemos explicitar cierta condición que debe cumplirse para que nos interese el resultado de la comparación entre la afirmación de la lógica y los eventos simbólicos del sistema.

Esta expresividad adicional es la que da pie a una lógica de tres estados en la que expresar el resultado de la comparación:

- *True (cierto)*. La afirmación es cierta.
- *False (falso)*. La afirmación es falsa.
- $\emptyset$ *(irrelevante)*. El resultado es irrelevante, ya que la acotación especificada deriva en un escenario vacío. En definitiva, la condición que describe el escenario de interés nunca es cierta.

Para reflejar el resultado de este operador de acotación de interés, introduciremos un nuevo elemento en nuestro lenguaje de expresiones: la restricción de escenario. Su sintaxis será:

$$[expresion_1](expresion_2)$$

Sus reglas de evaluación son sencillas: si $expresion_1$ evalúa a $true$, el conjunto evalúa a $expresion_2$. Si $expresion_1$ evalúa a $false$, el conjunto evalúa a $\emptyset$, independientemente de $expresion_2$.

$$[True](expresion_2) \equiv expresion_2$$

$$[False](expresion_2) \equiv \emptyset$$

8.3.1. Los operadores lógicos

La introducción de un tercer estado ($\emptyset$) en nuestra lógica de expresiones nos obliga a especificar el comportamiento de los operadores lógicos tradicionales frente a este nuevo valor.

Por un lado, podemos encontrar expresiones que combinen elementos de esta nueva lógica (fruto de la comparación entre eventos simbólicos) con otros de la lógica tradicional, que funcionen como simples habilitadores de las transiciones. En este caso, nos interesa que los operadores que los combinen propaguen este tercer estado. Para ello, el comportamiento de los operadores lógicos tradicionales ante este tercer valor sería:

$$\neg\emptyset \equiv \emptyset \qquad true \wedge \emptyset \equiv \emptyset \qquad \emptyset \wedge \emptyset \equiv \emptyset$$

$$\emptyset \Rightarrow e \equiv \emptyset \qquad false \vee \emptyset \equiv \emptyset \qquad \emptyset \vee \emptyset \equiv \emptyset$$

En el resto de los casos, el resultado sería el habitual del álgebra de Bool.

Por otro lado, cuando combinemos dos expresiones de esta nueva lógica puede que nos interese tomar decisiones que eliminen este tercer valor. Veamos un ejemplo.

Imaginemos que una propiedad afirma que en un estado no es posible un evento simbólico y que en ese estado el sistema ofrece dos eventos simbólicos que pueden coincidir con la descripción. El resultado de la verificación será una conjunción de dos expresiones, una por cada comparación. Ahora bien, si una

de las dos expresiones da como resultado $True$ (la propiedad se cumple en lo que respecta a ese evento) y la otra $\emptyset$, debemos entender que la propiedad se cumple, ya que uno de los dos elementos así lo dice mientras que el otro dice que, por su parte, es irrelevante.

En este caso, es evidente que no nos sirve la interpretación de conjunción que se deriva del operador lógico tradicional (que, al propagar el valor $\emptyset$, establecería que el resultado es irrelevante). En este caso, necesitamos otra interpretación que nos permita tomar decisiones como la comentada.

Por ello, vamos a definir otros operadores semejantes a los tradicionales, que nos permitan tomar ese tipo de decisiones y eliminar el $\emptyset$. Estos operadores serán $\bigtriangledown$ y $\bigtriangleup$ y su comportamiento es:

$$true \bigtriangleup \emptyset \equiv true \qquad \emptyset \bigtriangleup \emptyset \equiv \emptyset$$

$$false \bigtriangledown \emptyset \equiv false \qquad \emptyset \bigtriangledown \emptyset \equiv \emptyset$$

En el resto de los casos, su comportamiento será exactamente igual al de sus equivalentes en la lógica tradicional.

Como vemos, el principio que rige el comportamiento de estos operadores ante el valor $\emptyset$ es la neutralidad: el resultado final queda determinado por los restantes valores de la operación.

8.4. Verificación de eventos simbólicos

Las afirmaciones básicas que realizan las fórmulas de una lógica sobre el modelo de un sistema tienen que ver con la posibilidad de que determinados eventos ocurran en un cierto estado. En el caso de sistemas de transiciones etiquetados, la relación de satisfacción es muy simple, precisando de una simple observación del conjunto de eventos permitidos en el estado. Si una propiedad afirma que un evento es posible, la relación de satisfacción consiste en comprobar si en el estado actual del modelo del sistema existe una transición etiquetada con ese evento.

En nuestro caso, las afirmaciones que realicen las fórmulas LTCS sobre un estado del sistema de transiciones simbólico serán más complejas y el algoritmo

deberá tener en cuenta una serie de factores para tomar una decisión sobre la posibilidad o no de un evento simbólico. Estos factores son:

- El identificador del evento simbólico sobre el que estamos realizando la afirmación.
- La naturaleza del flujo de datos especificado, tanto en el evento del sistema como en el de la propiedad.
- Las posibles condiciones que gobiernan la transición del sistema etiquetada por el evento.
- La guarda del evento en la fórmula lógica.
- La posible acotación del escenario de verificación realizada en la fórmula lógica.
- Los condicionantes (invariantes) que la verificación de propiedades previas haya establecido sobre el evento del sistema. Es decir, si el evento en cuestión es, total o parcialmente, fijo; si el evento está, total o parcialmente, prohibido, etc.

8.4.1. Condición de verificación

Si el requisito afirma que un evento debe ser posible, el sistema debe ofrecerlo en ese instante (en ese estado) y sus patrones de datos deben ser equivalentes. Además, la condición que regula la transición del sistema etiquetada por ese evento y la guarda del evento del requisito suponen condiciones adicionales que deben ser ciertas para que la verificación sea positiva. Todo ello, condicionado a que la posible restricción del escenario de interés del evento del requisito sea cierta.

El resultado será una condición V, posiblemente involucrando a variables libres del estado actual y a variables ligadas en el evento que estamos analizando:

Sistema: $a\ p_s\ [g_s]$

Requisito: $a\ p_r\ [g_r]\ \{r_r\}$

$$\Longrightarrow V := [r_r](C_{(p_s \equiv p_r)} \wedge g_s \wedge g_r)$$

$C_{(p_s \equiv p_r)}$ representa la condición que debe ser cierta para que el intercambio de información que se establece a través del evento del requisito sea equivalente al que se establece a través del evento simbólico del sistema.

La anterior equivalencia, sobre todo cuando se modele la aceptación de información procedente del entorno, dependerá de un cambio de variables. Es decir, el requisito describirá el comportamiento que exige al sistema utilizando un conjunto de variables, generalmente distinto del empleado por el sistema. En la condición de equivalencia entre patrones de datos se establecerá una función de sustitución (σ) que recogerá esa equivalencia de variables y ligará las variables del requisito a las del sistema.

Todas las variables declaradas en los eventos del requisito serán sustituidas por sus homólogas (*alias*) del sistema, que se habrán identificado en el proceso de comparación. En el futuro, todas las expresiones serán sometidas a esa sustitución para eliminar las variables declaradas en los requisitos y trabajar sólo con las del sistema. Por ello, en realidad, la condición de verificación será:

$$\begin{array}{l} S\text{: } a\ p_s\ [g_s] \\ R\text{: } a\ p_r\ [g_r]\ \{r_r\} \end{array} \Longrightarrow V := [\sigma(r_r)](\sigma(C_{(p_s \equiv p_r)}) \wedge g_s \wedge \sigma(g_r))$$

Si el patrón de datos implica la asignación de valores a una o más variables ($\vec{v}$), la anterior condición debe ser cierta para todos los valores de esas variables. Por tanto, a la expresión anterior le añadiremos un cuantificador universal sobre esas variables.

$$\begin{array}{l} S\text{: } a\ p_s\ [g_s] \\ R\text{: } a\ p_r\ [g_r]\ \{r_r\} \end{array} \Longrightarrow V := \forall\ \{\vec{v}\}\ ([\sigma(r_r)](\sigma(C_{(p_s \equiv p_r)}) \wedge g_s \wedge \sigma(g_r)))$$

$\vec{v}$ incluye a todas las variables a las que se le asignan datos en p_s.

Si, por contra, el requisito afirma que cierto evento no es posible, para que el sistema cumpla el requisito es necesario que no exista ningún valor para el que la condición sea cierta. Por tanto:

$$\begin{array}{l} S\text{: } a\ p_s\ [g_s] \\ R\text{: } \neg\ a\ p_r\ [g_r]\{r_r\} \end{array} \Longrightarrow V := \neg\exists\ \{\vec{v}\}\ ([\sigma(r_r)](\sigma(C_{(p_s \equiv p_r)}) \wedge g_s \wedge \sigma(g_r)))$$

Las anteriores son, por tanto, las condiciones de verificación (CV) que se desprenden de sendas afirmaciones de una propiedad: que un evento es posible y que otro está prohibido.

8.4.2. Condición de rechazo

Un proceso de razonamiento equivalente nos conduce a las condiciones de rechazo (CR) en ambos casos.

Si la propiedad afirma que un evento es posible, comparando ese evento con los eventos prohibidos en ese estado del sistema obtendremos la condición que, de ser cierta, nos informa de que no es posible añadir ese evento al sistema para que verifique ese requisito. Eso iría en contra de uno o varios invariantes establecidos por propiedades verificadas previamente.

Esa condición es:

$$\begin{array}{l} S\text{: } \bar{a}\ p_s\ [g_s] \\ R\text{: } a\ p_r\ [g_r]\ \{r_r\} \end{array} \Longrightarrow R := \exists\ \{\vec{v}\}\ ([\sigma(r_r)](\sigma(C_{(p_s \equiv p_r)}) \wedge g_s \wedge \sigma(g_r)))$$

Por tanto, la única forma de lograr que el sistema verifique esa propiedad (es decir, que ofrezca ese evento) es eliminando el invariante que genera el rechazo que hemos encontrado. Para ello, o bien renunciamos a que el sistema cumpla la propiedad que generó el invariante, o bien tratamos de que la susodicha propiedad establezca un invariante alternativo que no entre en contradicción con la nueva propiedad.

En el caso de las propiedades que afirman la imposibilidad de un evento en un estado, el rechazo vendrá derivado de los eventos del sistema que coinciden con el especificado y han sido catalogados como fijos por una propiedad anterior. Igual que en el caso anterior, tendremos que eliminar el evento fijo del sistema y renunciar a la otra propiedad o volver a estudiarla buscando un invariante alternativo.

La condición que se genera en este caso es:

$$\begin{array}{l} S\text{: } \hat{a}\ p_s\ [g_s] \\ R\text{: } \neg\ a\ p_r\ [g_r]\{r_r\} \end{array} \Longrightarrow R := \exists\{\vec{v}\}\ ([\sigma(r_r)](\sigma(C_{(p_s \equiv p_r)}) \wedge g_s \wedge \sigma(g_r)))$$

8.5. Estados de aplicabilidad

Tal como se describe en el capítulo 6, las fórmulas de la lógica LTCS obedecen a un formato genérico constituido por una premisa, un operador temporal y una consecuencia. En caso de que la premisa sea cierta en el estado actual, la consecuencia deberá serlo en los estados que determine el operador temporal.

En el caso de los operadores temporales de futuro, esos estados en los cuales debe verificarse la consecuencia serán elegidos por el operador en cuestión a partir de un conjunto, denominado conjunto de estados de aplicabilidad (**CEA**), que depende exclusivamente del requisito que forme la premisa.

En la sección 6.1.3.2 especificamos cómo se construye el conjunto de estados de aplicabilidad en función de la forma de la premisa. Allí se especificaban una serie de reglas que cubrían todos los elementos que pueden aparecer en una fórmula de la lógica.

Por tanto, cuando nuestro algoritmo verifique un requisito, además de extraer las condiciones que nos informarán del resultado, deberá generar el conjunto de estados de aplicabilidad que se deriven de ese requisito. Si este requisito es la premisa de otro, el CEA calculado será utilizado para determinar el resultado.

Veamos algún ejemplo, según la forma de la premisa:

- $Req = True$

  Los estados de aplicabilidad serán todos los estados inmediatamente siguientes, es decir, los que se alcancen a través de una única transición.

- $Req = a\ ?x{:}TipoX$

  Los estados de aplicabilidad serán todos los estados que se puedan alcanzar a través de cualquier transición etiquetada con un evento simbólico $a\ ?x{:}TipoX$.

- $Req = c\ ?y{:}TipoY \wedge b\ !0$

  Los estados de aplicabilidad serán todos los estados que se puedan alcanzar a través de cualquier transición etiquetada con un evento simbólico $c\ ?y{:}TipoY$ ó $b\ !0$.

Como vemos, cuando el requisito que compone la premisa está compuesto por otros subrequisitos, sus estados de aplicabilidad se agregan. Sin embargo, esta agregación debe conservar la información sobre su origen, tanto del evento simbólico que etiqueta la transición como de las conectivas lógicas que unían a los subrequisitos. Esta información, como veremos posteriormente, es determinante para la articulación del algoritmo de verificación.

Por tanto, el CEA será una estructura de datos que unirá, mediante ciertos constructores, a los distintos conjuntos de estados de aplicabilidad de cada subrequisito.

8.5.1. La estructura CEA

El CEA, al que designaremos por la letra Ω, será una estructura compuesta por un constructor de entre cinco posibles. Cada uno de ellos tendrá su origen en un elemento de la lógica y almacenará la información necesaria para identificar el origen de cada subconjunto de estados de aplicabilidad.

Estos cinco constructores son:

- **LV**

  Este constructor representará un conjunto de estados de aplicabilidad vacío, derivado de una premisa cuya semántica implica la inexistencia de estados de aplicabilidad.

  Por ejemplo, $False$.

- **TF**

  Este constructor representa a la totalidad de posibles futuros en el estado actual. Irá acompañado por una lista donde se almacenará información sobre cada una de las transiciones: los eventos que las provocan y las posibles condiciones que las regulen.

Por ejemplo, si la premisa es $True$, el CEA estará compuesto por todos los estados futuros posibles y, por tanto, derivará en un operador **TF**.

- **LF**

  El constructor **LF** será el encargado de representar a la lista de futuros que se derive de la afirmación de que un evento simbólico es posible.

  En el estado correspondiente del sistema es posible que haya varios eventos que coincidan con el especificado. Cada uno de ellos conducirá a un futuro y todos ellos serán agrupados en una lista que acompañará a este constructor.

  Para cada una de esas transiciones se almacenará cierta información, que cobrará sentido en el momento de articular el algoritmo. Esta información será una cuádrupla:

  - El evento que etiqueta la transición y la posible condición de ésta.
  - La condición de verificación que se deriva de la comparación del evento con el especificado en el requisito.
  - La posible función de sustitución que se derive de la comparación entre las variables empleadas en ambos elementos.
  - La posible restricción del escenario de interés impuesta por el evento del requisito.

- **AF**

  Este constructor será empleado para representar la unión de los conjuntos de estados de aplicabilidad que se deriven de dos requisitos unidos por una conjunción lógica.

  Por tanto, tendrá como argumentos dos constructores que representarán los CEA que el mismo algoritmo habrá derivado de cada uno de los dos subrequisitos.

- **OF**

  Este último constructor realizará el mismo papel que el anterior en el caso de que los dos subrequisitos estuvieran relacionados por una disyunción lógica.

8.6. Recursividad

El formalismo que estamos desarrollando está pensado para su aplicación a sistemas reactivos, sin un final previsible en su comportamiento. Por ello, su evolución será, fundamentalmente, recursiva.

Además, como vimos en la sección 6.2, las propiedades en cuya verificación estamos interesados poseen, en su mayoría, una naturaleza recursiva.

Por tanto, un elemento fundamental de este procedimiento de verificación (y de todo aquel relacionado con sistemas reactivos) es el tratamiento de la recursividad.

La forma tradicional de manejar la recursividad en los formalismos lógicos más extendidos se basa en la teoría de puntos fijos [Tar55]. Por ejemplo, el *model checking* temporal mediante *μ-calculus* trata la recursividad de acuerdo a las siguientes reglas:

- Las propiedades recursivas se especifican mediante puntos fijos máximos (invarianzas) y mínimos (finalidades).
- Los algoritmos de verificación asignan constantes recursivas a las fórmulas lógicas y, en el proceso de inducción estructural, se detienen cuando se repite una comparación entre una constante recursiva y un nodo del árbol.
- La decisión de si la propiedad se cumple o no se basa en la neutralidad de las constantes recursivas.

  La veracidad de una finalidad no puede ser determinada por un hallazgo de recursividad. Por tanto, la repetición en un estado de una constante recursiva que denota una finalidad se traduce a $False$.

  Por otro lado, el cumplimiento de una invarianza no puede ser rechazado por la aparición de una recursividad. Por tanto, la repetición en un estado de una constante recursiva que denota una invarianza se traduce a $True$.

El tratamiento que daremos a la recursividad en nuestro algoritmo de verificación se basará en el mismo principio de neutralidad, aunque su implementación será un poco más compleja. Esto es debido, principalmente, a dos motivos:

1. Las fórmulas lógicas de la LTCS no se autocaracterizan, como ocurre en el *μ-calculus*. La simple observación de una recursividad no permite adivinar si representa una invarianza o una finalidad.

2. La determinación del cumplimiento de una propiedad dependerá, generalmente, de los valores de las variables del sistema. Por tanto, al encontrar una recursividad no se podrá tomar una decisión, sino que deberá quedar reflejada en las condiciones de verificación y rechazo a la espera de la asignación de valores a las variables libres.

8.6.1. Implementación

Como hemos mencionado en la sección anterior, no será el algoritmo de verificación el que tome una decisión al detectar una recursión. Simplemente, se limitará a reflejar la recursión en las condiciones de verificación y rechazo que está construyendo.

Por tanto, es necesario dotar al lenguaje de expresiones empleado de los mecanismos necesarios para plasmar en él las situaciones de recursividad que puedan aparecer en las distintas comparaciones y poder tomar decisiones una vez resueltos los valores de las variables.

Para ello, definiremos una constante recursiva **REC** que llevará asociado el estado en donde se detectó la recursión. El algoritmo de verificación, tras detectar una recursión, terminará la condición que está construyendo añadiendo esa constante recursiva.

Será, por tanto, en el proceso de evaluación de las condiciones de verificación y rechazo en donde se debe tomar una decisión según el citado principio de neutralidad.

Esta decisión vamos a confiarla a los operadores $\triangle$ y $\bigtriangledown$ (introducidos para el tratamiento de la constante $\emptyset$). Para ello, vamos a añadirle también a esos operadores el estado en el que se generan. Este estado será irrelevante a la hora de tomar decisiones sobre la constante $\emptyset$, pero habrá que tenerlo en cuenta en el caso de **REC**.

El principio de actuación de los operadores $\triangle$ y $\bigtriangledown$ será similar al que rige el tratamiento de $\emptyset$: el resultado será determinado por los restantes elementos que intervienen en la operación. Si en un estado se despliega un requisito compuesto

por la conjunción o disyunción de dos subrequisitos y uno de ellos deriva en una condición mientras que el otro deriva en una recursión, el resultado será la condición.

$$\text{C} \bigtriangleup(s)\ REC(s) \longrightarrow \text{C} \qquad\qquad \text{C} \bigtriangledown(s)\ REC(s) \longrightarrow \text{C}$$

Sobre este comportamiento general es necesario realizar varias puntualizaciones:

- En el caso de una conjunción, si la recursión es a un estado previo s_1, el operador $\bigtriangleup$ tomará la misma decisión.

  $$\text{C} \bigtriangleup(s)\ REC(s_1) \longrightarrow \text{C}$$

- En el caso de una disyunción, si la recursión es a un estado previo s_1, el operador $\bigtriangledown$ no tomará ninguna decisión. Por tanto, si la condición C es $True$ el resultado será $True$ y si es $False$, el operador propagará la recursión para que un posible operador $\bigtriangleup$ que los contenga pueda tener la oportunidad de tomar una decisión.

  $$True \bigtriangledown(s)\ REC(s_1) \longrightarrow True$$

  $$False \bigtriangledown(s)\ REC(s_1) \longrightarrow REC(s_1)$$

- En el caso de que ambos ambos requisitos deriven en sendas recursiones hacia el mismo estado, propagaremos ese resultado para que un operador que los contenga pueda tomar una decisión.

  $$REC(s) \bigtriangleup(s)\ REC(s) \longrightarrow REC(s)$$

  $$REC(s) \bigtriangledown(s)\ REC(s) \longrightarrow REC(s)$$

Por su parte, los operadores lógicos tradicionales propagarán la recursión en cualquier caso.

Para detectar los bucles de una recursión es necesario recordar los requisitos que ya se han contrastado con todos y cada uno de los estados del árbol del

sistema durante el proceso de verificación. Esta información se almacenará en cada nodo del árbol, en una estructura definida a tal efecto, denominada **Lista de Requisitos Desplegados** (**LRD**). La LRD de un determinado estado S almacenará la lista de las cabeceras de los requisitos que ya se han desplegado en ese estado.

$$\mathbf{LRD}_S = [R_1\ \{\vec{e_1}\}\ (\vec{p_1});\ ..\ ;\ R_I\ \{\vec{e_I}\}\ (\vec{p_I})\ ]$$

8.6.2. Ámbito de aplicación

El tratamiento de la recursión, tal como ha sido descrito, tiene como punto de partida el hallazgo de una situación recursiva en el desplegamiento de un requisito en un estado del sistema. Sin embargo, la incorporación del intercambio de información con el entorno en la descripción del sistema enriquece notablemente el conjunto de situaciones recursivas representables, lo que dificulta su identificación y resolución.

Los distintos enfoques con los que formalismos similares abordan el tratamiento de la recursividad en estas condiciones convergen en la necesidad de imponer restricciones en alguno de los parámetros involucrados [HL95b, HR97]: los modelos de sistemas susceptibles de análisis, el tipo de propiedades verificables, los tipos de recursiones cubiertas, etc. En cada caso, se suelen proporcionar algoritmos adecuados a un escenario concreto, donde rige alguna de las restricciones mencionadas.

En nuestro caso, hemos optado por establecer restricciones al tipo de recursiones cubiertas, permitiendo solamente el empleo de recursiones en las cuales tanto los eventos simbólicos del requisito instanciado como sus parámetros sean los mismos en los dos desplegamientos que definen la recursividad. Este tipo de limitación permite la obtención de algoritmos no excesivamente complejos, al tiempo que todavía posibilita la verificación de un espacio de sistemas muy amplio.

Por tanto, el ámbito de aplicación del algoritmo de verificación presentado se limita a sistemas y propiedades que contengan únicamente recursiones sin parámetros o con los mismos parámetros en los dos desplegamientos que determinan una recursión.

8.7. Algoritmo de verificación

En las anteriores secciones hemos descrito los diversos elementos auxiliares definidos para desarrollar el algoritmo. En esta sección procedemos a su descripción.

El algoritmo de verificación toma como entradas un requisito LTCS (Req), el estado del árbol simbólico en el que deseamos verificarlo (S_a) y la función de sustitución σ_a vigente hasta ese momento (inicialmente será una sustitución vacía).

A su salida, el algoritmo devolverá tres resultados: la condición de verificación (**CV**), la condición de rechazo (**CR**) y el conjunto de estados de aplicabilidad (**CEA**). El CEA, como ya se ha mencionado, sólo se emplea en las etapas intermedias del algoritmo y no tiene utilidad al finalizar éste.

$$(CV,\, CR,\, \Omega) = \mathbf{Alg}\ (Req,\, S_a,\, \sigma_a)$$

Con todo esto, el algoritmo se describe en función del requisito, por inducción estructural:

Alg(Req, S_a, σ_a) := Según sea el requisito:

- Req = True [exp] {res}

  Devolver ($[\sigma_a(res)](\sigma_a(exp))$, $False$, **TF**)

- Req = False

  Devolver ($False$, $False$, **LV**).

- Req = ¬ a p_r [g_r] {r_r} $\quad\quad \tilde{a} = \{a\} \cup \{\hat{a}\}$

  $\exists n\ \tilde{a}\ p_i$ [g_i] $\in S_a$ y $\exists m\ \hat{a}\ p_j$ [g_j] $\in S_a$

  Sólo se tienen en cuenta los eventos cuyos patrones de datos son equivalentes al patrón de datos del evento del requisito. Esta equivalencia se producirá, en cada caso, bajo una cierta sustitución de variables σ_i ó σ_j.

  $CV_i = \neg\exists\{\vec{v_i}\}([\sigma_I(r_r)](\sigma_I(C_{p_r \equiv p_i}) \wedge g_i \wedge \sigma_I(g_r)))$

  donde $\sigma_I = \sigma_i \cup \sigma_a$

  $\vec{v_i}$ será la lista de variables que toman un valor en p_i.

  $CV = \mathop{\triangle}_{i=1}^{n} CV_i$

  $CR_j = \exists\{\vec{v_j}\}([\sigma_J(r_r)](\sigma_J(C_{p_r \equiv p_j}) \wedge g_j \wedge \sigma_J(g_r)))$

  donde $\sigma_J = \sigma_j \cup \sigma_a$

  $\vec{v_j}$ será la lista de variables que toman un valor en p_j.

  $CR = \mathop{\triangledown}_{j=1}^{m} CR_j$

  Devolver (CV, CR, **TF**)

- Req = a p_r `[`g_r`]` $\{r_r\}$

  $\exists n\ \tilde{a}\ p_i$ `[`g_i`]` $\in S_a$ y $\exists m\ \bar{a}\ p_j$ `[`g_j`]` $\in S_a$

  $CV_i = \forall\{\vec{v_i}\}([\sigma_I(r_r)](\sigma_I(C_{p_r \equiv p_i}) \wedge g_i \wedge \sigma_I(g_r)))$

  donde $\sigma_I = \sigma_i \cup \sigma_a$

  $\vec{v_i}$ será la lista de variables que toman un valor en p_i.

  CV = $\mathop{\bigtriangledown}\limits_{i=1}^{n} CV_i$

  $CR_j = \exists\{\vec{v_j}\}([\sigma_J(r_r)](\sigma_J(C_{p_r \equiv p_j}) \wedge g_j \wedge \sigma_J(g_r)))$

  donde $\sigma_J = \sigma_j \cup \sigma_a$

  $\vec{v_j}$ será la lista de variables que toman un valor en p_j.

  CR = $\mathop{\bigtriangledown}\limits_{j=1}^{m} CR_j$

  Para cada futuro posible creamos una cuádrupla de información que será utilizada cuando se traten los operadores de futuro.

  $f_i = (\tilde{a}\ p_i$ `[`g_i`]`$,\ CV_i,\ \sigma_I,\ \sigma_I(r_r))$

  Ω = **LF** ($\bigcup\limits_{i=1}^{n} f_i$ **)**

  Devolver $(CV,\ CR,\ \Omega)$

- Req = $Req_1 \vee Req_2$

Aplicamos el algoritmo a cada uno de los dos elementos y componemos el resultado.

$(CV_i, CR_i, \Omega_i) = \mathbf{Alg}(Req_i, S_a, \sigma_a)$

$CV = CV_1 \bigtriangledown CV_2$

$CR = CR_1 \bigtriangleup CR_2$

$\Omega = \mathbf{OL}\ (\Omega_1, \Omega_2)$

Si Ω_1 y Ω_2 son **LV**, devolvemos **LV** en lugar de **OL**. Si una es **LV**, devolvemos la otra.

Devolver $(CV,\ CR,\ \Omega)$

- Req = $Req_1 \wedge Req_2$

  Aplicamos el algoritmo a cada uno de los dos elementos y componemos el resultado.

  $(CV_i, CR_i, \Omega_i) = \mathbf{Alg}(Req_i, S_a, \sigma_a)$

  $CV = CV_1 \bigtriangleup CV_2$

  $CR = CR_1 \bigtriangledown CR_2$

  $\Omega = \mathbf{AL}\ (\Omega_1,\ \Omega_2)$

  Si Ω_1 y Ω_2 son **LV**, devolvemos **LV** en lugar de **AL**. Si una es **LV**, devolvemos la otra.

  Devolver $(CV,\ CR,\ \Omega)$

- Req = $R_P \Longrightarrow\!\otimes\ [\text{Æ}\{\vec{v}\}]\ R_C$ Æ $\in \{\exists, \forall\}$

  $(CV_P, CR_P, \Omega_P) = \mathbf{Alg}(R_P, S_a, \sigma_a)$

Según sea el operador:

1. $\otimes =$" " $\longrightarrow (CV_C, CR_C, \Omega_C) = \mathbf{Alg}(R_C, S_a, \sigma_a)$

   CV $= CV_P \Rightarrow CV_C$

   CR $= \neg CR_P \wedge CR_C$

   Devolver (CV, CR, **TF**)

2. $\otimes = \bigodot \longrightarrow (CV_C, CR_C, \Omega_C) = \mathbf{Alg}(R_C, S_{ant}, \sigma_a)$

   S_{ant} es el estado anterior al actual, S_a.

   CV $= CV_P \Rightarrow CV_C$

   CR $= \neg CR_P \wedge CR_C$

   Devolver (CV, CR, **TF**)

3. $\otimes = \bigcirc$, ⓐ ó ⓔ

   Sólo en este caso puede existir Æ$\{\vec{v}\}$

   $(CV_C, CR_C) = Eval_\Omega$ Æ$\{\vec{v}\}$ $R_C \otimes \Omega_P$

   CV $= CV_P \Rightarrow CV_C$

   CR $= \neg CR_P \wedge CR_C$

   Devolver (CV, CR, **TF**)

La función $Eval_\Omega$ se define de la siguiente manera:

$Eval_\Omega$ Æ$\{\vec{v}\}$ $R_C \otimes \Omega_P$:= En función de Ω_P:

LV $\longrightarrow$ Devolver (False, False)

LF L $\longrightarrow$

$$\text{Si } L = [] \begin{cases} \otimes = \bigcirc & \longrightarrow \quad Devolver(True, False) \\ \\ \otimes = \text{@ ó ⓔ} & \longrightarrow \quad Devolver(False, False) \end{cases}$$

En otro caso, $L = [(a_1, c_1, \sigma_1, r_1); ...; (a_I, c_I, \sigma_I, r_I)]$

$\text{C} = \bigvee_{i=1}^{I} c_i$

$(CV_{Ci}, CR_{Ci}, \Omega_{Ci}) = \mathbf{Alg}(R_C, S_{sig_i}, \sigma_a \cup \sigma_i)$

Según sea el operador:

$$\bigcirc \Longrightarrow \begin{cases} CV_C = \mathop{\triangle}_{i=1}^{I} (c_i \Rightarrow Æ\{\vec{v}\}([r_i](CV_{Ci}))) \\ \\ CR_C = \mathop{\triangledown}_{i=1}^{I} (c_i \wedge \bar{Æ}\{\vec{v}\}([r_i](CR_{Ci}))) \end{cases}$$

$$\text{@} \Longrightarrow \begin{cases} CV_C = C \wedge \mathop{\triangle}_{i=1}^{I} (c_i \Rightarrow Æ\{\vec{v}\}([r_i](CV_{Ci}))) \\ \\ CR_C = \mathop{\triangledown}_{i=1}^{I} (c_i \wedge \bar{Æ}\{\vec{v}\}([r_i](CR_{Ci}))) \end{cases}$$

$$\text{ⓔ} \Longrightarrow \begin{cases} CV_C = \bigvee_{i=1}^{I} (c_i \wedge Æ\{\vec{v}\}([r_i](CV_{Ci}))) \\ \\ CR_C = False \end{cases}$$

Devolver (CV_C, CR_C)

TF L $\longrightarrow$

$$\text{Si } L = [\,] \begin{cases} \otimes = \bigcirc & \longrightarrow \quad Devolver(True, False) \\ \\ \otimes = @ \text{ ó } ⓔ & \longrightarrow \quad Devolver(False, False) \end{cases}$$

En otro caso, L = $[a_1\ p_1\ [g_1], \ldots, a_I\ p_I\ [g_I]]$

C = $\bigvee_{i=1}^{I} g_i$

$(CV_{Ci}, CR_{Ci}, \Omega_{Ci}) = \textbf{Alg}(R_C, S_{sig_i}, \sigma_a)$

Según sea el operador:

$$\bigcirc \Longrightarrow \begin{cases} CV_C = \mathop{\triangle}_{i=1}^{I} \forall\ \{\vec{v_i}\}(g_i \Rightarrow CV_{Ci}) \\ \\ CR_C = \mathop{\triangledown}_{i=1}^{I} \exists\ \{\vec{v_i}\}(g_i \wedge CR_{Ci}) \end{cases}$$

$$@ \Longrightarrow \begin{cases} CV_C = C \wedge \mathop{\triangle}_{i=1}^{I} \forall\ \{\vec{v_i}\}(g_i \Rightarrow CV_{Ci}) \\ \\ CR_C = \mathop{\triangledown}_{i=1}^{I} \exists\ \{\vec{v_i}\}(g_i \wedge CR_{Ci}) \end{cases}$$

$$ⓔ \Longrightarrow \begin{cases} CV_C = \bigvee_{i=1}^{I} \forall\ \{\vec{v_i}\}(g_i \wedge CV_{Ci}) \\ \\ CR_C = False \end{cases}$$

Devolver (CV_C, CR_C)

AL(Ω_1, Ω_2) $\longrightarrow$

$(CV_{C1}, CR_{C1}) = Eval_\Omega$ Æ$\{\vec{v}\}$ $R_C \otimes \Omega_1$

$(CV_{C2}, CR_{C2}) = Eval_\Omega$ Æ$\{\vec{v}\}$ $R_C \otimes \Omega_2$

Según sea el operador:

$$\bigcirc \Longrightarrow \begin{cases} CV_C = CV_{C1} \bigtriangleup CV_{C2} \\ \\ CR_C = CR_{C1} \bigtriangledown CR_{C2} \end{cases}$$

$$ⓔ \Longrightarrow \begin{cases} CV_C = CV_{C1} \bigtriangleup CV_{C2} \\ \\ CR_C = False \end{cases}$$

Devolver (CV_C, CR_C)

OL(Ω_1, Ω_2) $\longrightarrow$

$(CV_{C1}, CR_{C1}) = Eval_\Omega$ Æ$\{\vec{v}\}$ $R_C \otimes \Omega_1$

$(CV_{C2}, CR_{C2}) = Eval_\Omega$ Æ$\{\vec{v}\}$ $R_C \otimes \Omega_2$

Según sea el operador:

$$\bigcirc \Longrightarrow \begin{cases} CV_C = CV_{C1} \bigtriangledown CV_{C2} \\ \\ CR_C = CR_{C1} \bigtriangleup CR_{C2} \end{cases}$$

$$ⓔ \Longrightarrow \begin{cases} CV_C = CV_{C1} \bigtriangledown CV_{C2} \\ \\ CR_C = False \end{cases}$$

Devolver $(CV_C,\, CR_C)$

- Req = Q $\{e_A\}$ (p_A)
  - Q $\{e_A\}$ $(p_A) \in LRD_{S_a}$

    Devolver (REC(S_a), REC(S_a), **LV**).

    Si Q perteneciese a LRD_{S_a} pero los parámetros de la recursión no fuesen los mismos que los de la declaración, habría que levantar una excepción de fallo de modelo.

  - Q $\{e_A\}$ $(p_A) \notin LRD_{S_a}$ $\longrightarrow$ Desplegamiento del requisito.

    $LRD_{S_a} := LRD_{S_a} \cup$ [Q $\{e_A\}$ (p_A)]

    Buscar Q en la lista de requisitos y extraer su fórmula F_Q.

    Devolver **Alg**(F_Q, S_a, σ_a)

8.8. Análisis de los resultados

Los resultados del algoritmo de verificación expuestos pueden llegar a ser bastante complejos. Dependiendo del sistema y de la propiedad, las condiciones de verificación y rechazo pueden ser muy extensas, involucrando a múltiples variables relacionadas por operadores de cuantificación, lógicos, de restricción del escenario de interés, etc.

El verdadero resultado del proceso de verificación (si el sistema cumple o no la propiedad) puede no ser fácilmente observable a partir de las citadas condiciones. La complejidad de la relación entre las variables del sistema y los

múltiples valores que éstas pueden tomar puede hacer necesario un análisis de las expresiones para determinar el resultado final.

El tratamiento de las expresiones de datos y la determinación de su veracidad o falsedad es un tema complejo que constituye una disciplina de investigación por sí misma. Como mencionábamos en el capítulo 2, el *narrowing* [Klo87] y la reescritura [Klo92] son las dos aproximaciones más habituales en el tratamiento de expresiones.

Aun así, la resolución de expresiones de datos con variables libres se convierte en un problema indecidible en cuanto el lenguaje de expresiones y los tipos de datos alcanzan un tamaño crítico.

Por ello, no es un objetivo de esta tesis el realizar un estudio de esas expresiones y proporcionar canales para establecer la veracidad o falsedad de esas condiciones. De forma semejante a otros formalismos de verificación formal [HL95b], la herramienta en la que se integrarán estos algoritmos podrá delegar en una herramienta auxiliar el análisis de los resultados cuando así sea necesario. Para tal tarea se podrá utilizar alguno de los múltiples sistemas de prueba existentes en la literatura [BvLV94].

En algunos casos, la veracidad o falsedad de las condiciones podrá ser decidida por este tipo de sistemas. Sin embargo, dependiendo de la complejidad de las mismas, esta tarea puede ser imposible sin una orientación sobre los valores o rangos de valores que las distintas variables pueden tomar. En cualquier caso, tal información seguramente aceleraría mucho el procedimiento de decisión aunque éste pudiese llegar a buen término sin ayuda.

Por otro lado, aunque los tipos de las variables que aparezcan en las condiciones puedan no ser acotados o su universo de valores muy grande, el conocimiento subjetivo que del sistema tiene el usuario puede llevarle a precisar su verificación para ciertos valores o rangos de valores de especial significación.

Por tanto, lo habitual en este tipo de análisis es que el usuario, tras observar las condiciones resultantes del proceso de verificación, especifique un conjunto de valores o rangos de valores para los que desea llevar a cabo el análisis y obtener un resultado. A esa información la denominaremos **Conjunto de Valores Seleccionados** (**CVS**) y jugará un papel importante en el algoritmo de cálculo de sugerencias del próximo capítulo.

El **CVS** de cada variable supone, en esencia, una relajación del rango de valores de la variable para el que hay que llevar a cabo el análisis de los

resultados. Esta relajación podrá ser tan tibia o contundente como se desee. Por regla general, cabe esperar que, a mayor relajación, con más rapidez se obtendrá una solución. Esta solución, sin embargo, nos indicará el resultado del proceso de verificación si el sistema se limita a intercambiar los valores especificados en el **CVS** de cada variable. En general, al aumentar la relajación, disminuirá el grado de confianza que tendremos en que el resultado se mantenga para otros valores fuera del **CVS**.

Capítulo 9

Modificaciones funcionales

9.1. Introducción

En el capítulo anterior dimos forma a un procedimiento que nos permite averiguar si el modelo del sistema bajo desarrollo cumple un conjunto de propiedades expresadas mediante la lógica temporal LTCS.

El resultado de tal proceso nos indicaba si el sistema cumplía o no una propiedad en función de los eventos que podían tener lugar y de la información que podía intercambiar con su entorno. Además, en caso de que la propiedad no se cumpliese, el procedimiento nos permitía saber si se debía a un defecto en la especificación del sistema (éste puede modificarse para cumplir la propiedad) o a una incoherencia entre las propiedades que le exigimos (no existe modificación posible que haga que el sistema cumpla todas las propiedades deseadas y es necesario alterar algún requisito).

En el caso de una incoherencia entre las propiedades que exigimos al sistema, resulta complicado definir mecanismos de ayuda automática que nos faciliten la continuación del proceso de diseño. El camino más intuitivo pasa por relajar parcial o totalmente alguna de las propiedades que entran en contradicción. Sin embargo, la decisión de qué propiedad relajar y en qué grado depende profundamente de cada sistema y propiedad, de la importancia que le atribuimos en cada caso y de una serie de condicionantes subjetivos que resultan de muy difícil captura en un formalismo matemático. Por tanto, este es un camino por el que no vamos a continuar más allá de la identificación de

las propiedades que entran en contradicción.

Sin embargo, sí parece más asequible realizar aportaciones en el caso de que, no verificando la propiedad, el resultado indique que el sistema es susceptible de ser modificado para que lo haga. En ese caso, la tarea del diseñador es encontrar las modificaciones adecuadas para conseguir que se satisfaga la nueva propiedad sin perder las propiedades verificadas con anterioridad.

En esa labor, el procedimiento de *model checking* empleado para la verificación nos proporciona poca o ninguna ayuda. Sí es cierto que nos dice que el objetivo es factible, pero no aporta ninguna información acerca de cuál es el camino para alcanzarlo.

La dificultad de tal cometido resulta de muy difícil cuantificación. Dependiendo del sistema y de la propiedad, las modificaciones pueden ser obvias y estar muy localizadas o abarcar alteraciones en múltiples estados, con relaciones de difícil visibilidad. Parece razonable esperar que, en sistemas de gran tamaño, existan múltiples alternativas válidas de modificación, cada una de ellas con distinto grado de complejidad y consecuencias diversas para el futuro del desarrollo.

Por tanto, parece del todo punto indicado que el diseñador disponga del máximo grado de ayuda en la identificación de las modificaciones posibles del árbol simbólico y de los efectos que cada una de ellas puede producir en el futuro del sistema.

Ese es el objetivo general de este capítulo: la construcción de un procedimiento de ayuda que identifique las modificaciones que es necesario realizar en un sistema (en su modelo de transiciones simbólico) para que cumpla una determinada propiedad expresada en la lógica temporal LTCS. Este procedimiento debería:

- Generar un conjunto de sugerencias. Cada una de esas sugerencias englobará varias modificaciones del sistema de transiciones simbólico que es necesario realizar en su totalidad para conseguir que la nueva representación verifique la propiedad.

- Tener en cuenta el CVS (sección 8.8) indicado por el usuario para cada una de las variables que representan los valores de la información intercambiada con el entorno.

- Mantener la integridad. Las sugerencias calculadas no deben derivar en

modificaciones que provoquen el incumplimiento de otra propiedad verificada con anterioridad.

9.2. Modelo desarrollado

El enfoque básico del algoritmo descrito en el capítulo 8 para la verificación de propiedades consistía en el establecimiento de las condiciones que se debían satisfacer para asegurar que el sistema puede secundar las afirmaciones de la propiedad. En ese proceso, las diversas subcondiciones que impone cada parte de la propiedad en el mismo o diferentes instantes de tiempo se agregan, creando una única expresión potencialmente muy compleja.

La expresión resultante, construida a base de una fusión continuada de subcondiciones y un procedimiento de simplificación constante, impide discernir qué evento añade una determinada subcondición y en qué instante de tiempo (o, lo que es lo mismo, en qué estado del STS que representa al sistema).

Es decir, una vez obtenida la expresión global es imposible averiguar la influencia de cada estado y cada parte de la propiedad en la citada expresión. Esto, claramente, nos impide razonar sobre posibles modificaciones del sistema ya que, aunque supiéramos cuál es la subexpresión que no se cumple (lo cual no siempre es cierto debido al proceso continuo de simplificación), no podemos conocer qué evento de la propiedad la originó ni en qué estado del árbol simbólico.

Dado que el objetivo que perseguimos es identificar con precisión el porqué del incumplimiento de una expresión, parece apropiado preservar en todo momento el origen de sus distintos componentes. Por ello, el enfoque que vamos a adoptar consiste en modificar el procedimiento de *model checking* del algoritmo de verificación para que esa información quede reflejada en el resultado. Se trata de sustituir la lógica tradicional con las constantes y operadores del álgebra de Bool por una lógica de estructuras de información que nos permita localizar los incumplimientos parciales y generar sugerencias para corregirlos.

9.2.1. Fases del procedimiento

La implementación de este mecanismo de generación de sugerencias se llevará a cabo en dos fases:

1. En primer lugar definiremos las estructuras o registros de información que van a ocupar el lugar de las expresiones tradicionales en el mecanismo de *model checking*. Estos registros se derivarán de los diversos constructores de la lógica empleada y, además de contener a las expresiones originales, incluirán otros campos de información que permitirán determinar su origen.

   El objetivo principal de esta primera fase será diseñar un algoritmo similar al de verificación que nos proporcione una estructura que refleje las distintas comparaciones que se derivan de la verificación de una propiedad. Esta estructura estará compuesta por los registros de información definidos previamente.

2. En una segunda etapa analizaremos la estructura generada en la fase anterior y extraeremos las sugerencias sobre las modificaciones necesarias para que la nueva representación verifique la propiedad. El proceso consistirá en generar las sugerencias para solventar los incumplimientos parciales y combinarlas de acuerdo a los constructores lógicos de la propiedad.

   Para ello, previamente, definiremos el conjunto de modificaciones básicas que se van a sugerir y de las cuales estarán compuestas las sugerencias.

Para poder generar sugerencias parciales es necesario tomar decisiones sobre posibles incumplimientos parciales y eso, en muchos casos, depende de los valores asignados a las variables libres de las expresiones. Por ello, en este proceso juega un papel muy importente el CVS (ν) asignado a cada variable en el proceso de verificación (que, recordemos, dio un resultado negativo). A fin de cuentas, las modificaciones que se van a proponer son las necesarias para que la propiedad se verifique para los valores de las variables indicados.

Por tanto, a lo largo de los algoritmos desarrollados para implementar las dos fases mencionadas se realizará, cuando sea necesario, una evaluación de ciertas expresiones de acuerdo al CVS de las variables que aparecen en ellas. La evaluación de una expresión (exp) se denotará por $\|exp\|_\nu$ y tendrá como resultado $True$, $False$ ó $\emptyset$.

9.3. Generación de la información sobre el incumplimiento

Como hemos introducido en la sección anterior, el primer paso consistirá en definir los registros de información que sustituirán a las expresiones.

El registro fundamental será el que almacene la información que se derive de la comparación entre un evento simple de la propiedad y la información del estado correspondiente del árbol simbólico. A este registro lo denominaremos **RIAV**: **R**egistro de la **I**nfluencia de una **A**firmación básica sobre la condición de **V**erificación. Sus campos serán:

Es El estado en el que se origina la expresión de este registro.

Ev El evento del árbol, si alguno, con el que se compara el requisito.

Req El requisito que se analiza.

CV El resultado de la verificación de **Req**.

En esta misma línea, definiremos un registro de información por cada uno de los elementos de la lógica LTCS que pueden aparecer en la propiedad que estamos analizando. Estos registros, que contendrán toda la información necesaria para averiguar el motivo del incumplimiento, formarán una estructura que denominaremos **IVR**: **I**nformación sobre la **V**erificación de un **R**equisito. Esta estructura puede albergar uno de los siguientes registros:

$$\begin{aligned} IVR \;=\; & RIAV \mid OrR \mid AndR \mid OrF \mid AndF \mid Imp \\ & \mid Rec \mid Reject \mid RCond \end{aligned}$$

$$\mathbf{OrR} = \begin{cases} E \\ Req \\ IVR\ list \end{cases} \qquad \mathbf{Imp} = \begin{cases} E \\ Req \\ IVR * IVR \end{cases}$$

$$\textbf{AndR} = \begin{cases} E \\ Req \\ IVR\ \ list \end{cases} \qquad \textbf{OrF} = \begin{cases} E \\ Req \\ (Expr * Cuant * IVR)\ \ list \end{cases}$$

$$\textbf{Rec} = \begin{cases} E \\ Req \end{cases} \qquad \textbf{AndF} = \begin{cases} E \\ Req \\ (Expr * Cuant * IVR)\ \ list \end{cases}$$

$$\textbf{RCond} == (Expr * IVR * IVR)$$

Todos los registros, excepto *Reject* y *RCond*, contienen el estado en donde se generan y el requisito que los provoca. Cada uno de ellos se generará al encontrar un elemento determinado de la lógica LTCS:

RIAV Aparece al encontrar un evento o una constante lógica ($True$ ó $False$).

Reject Es una constante que indica que el sistema rechaza el requisito.

OrR Se genera a partir de una disyunción de requisitos. Contiene la IVR que genera cada uno de los subrequisitos que componen la disyunción.

AndR Se genera a partir de una conjunción de requisitos. Contiene la IVR que genera cada uno de los subrequisitos que componen la conjunción.

Rec Provocado por una recursión.

Imp Se genera a partir de una relación de implicación entre una premisa y una consecuencia. Contiene las IVR que generan ambos.

OrF Aparece por una disyunción de varios futuros, derivada de un operador ⓔ. Por cada futuro, además de la IVR que genera, se acompaña la expresión que condiciona la transición. Si es falsa, la transición no es posible. Si es $\emptyset$, esa rama es irrelevante para los valores especificados.

El segundo elemento que define cada futuro (el valor de tipo *Cuant*) recoge el cuantificador aplicado a las variables a las que se le asignan datos en el evento correspondiente. Podrá tomar tres valores: NoC (no

hay cuantificador), FA (cuantificador universal) y EX (cuantificador existencial).

AndF Aparece por una conjunción de varios futuros, derivada de un operador $\bigcirc$ ó @. Sus campos son los mismos que los del registro **OrF**.

RCond Refleja una disyuntiva entre dos IVR. Si la expresión del primer campo es cierta nos quedaremos con la primera IVR. Si no lo es, nos quedaremos con la segunda.

9.3.1. Algoritmo de generación de la IVR

En esta sección procederemos a definir el resultado IVR según sea la fórmula LTCS del requisito. Este resultado consistirá en una serie de registros de información unidos por derivaciones de los operadores lógicos clásicos.

Además, como antes, se devolverá un resultado auxiliar, el CEA (sección 8.5), que recogerá los estados de aplicabilidad que necesitan los operadores de futuro. Este resultado auxiliar, como en el algoritmo de verificación, sólo servirá para construir la información que se retorne en etapas intermedias; no tendrá significado al finalizar el algoritmo.

Como se puede ver, la información básica con la que se trabajaba en el algoritmo de verificación se mantiene (CV y Ω). Simplemente, se enriquece con datos adicionales que nos permitirán no sólo averiguar si una expresión es falsa, sino determinar en qué estado falla y por qué motivos.

El resultado de este nuevo algoritmo va a ser mucho más complejo que el del anterior ya que, en contra de lo que ocurre con las expresiones, no es posible la simplificación de varios registros en uno sólo (no sólo no es posible, sino que va en contra de nuestro objetivo).

El cuerpo central del algoritmo lo implementará la función Get_IVR, que recibe un requisito, un puntero a un estado y la sustitución σ_a vigente hasta el momento. Devolverá la IVR y el CEA.

$$(\text{IVR}, \Omega) = Get_IVR\ (\text{Req}, E_a, \sigma_a)$$

En función de la estructura del requisito:

- Req = True [exp] {res}

$$\text{I} = \textbf{RIAV} \begin{cases} Es = E_a \\ Ev = Ninguno \\ Req = True\ [exp]\{res\} \\ CV = [\sigma_a(res)](\sigma_a(exp)) \end{cases}$$

Devolver (I, **TF**)

- Req = ¬ a p_r $[g_r]$ $\{r_r\}$ $\quad\quad \tilde{a} = \{a\} \cup \{\hat{a}\}$

Suponemos que en el estado actual del árbol (E_a) existen n eventos posibles "$\tilde{a}$ p_i $[g_i]$" de los cuales m son eventos fijos "$\hat{a}$ p_j $[g_j]$".

Si no hay eventos posibles, evidentemente, el requisito se cumple.

$$\text{Si } n = 0 \longrightarrow \text{I} = \textbf{RIAV} \begin{cases} Es = E_a \\ Ev = Ninguno \\ Req = \neg a\ p_r\ [g_r]\{r_r\} \\ CV = True \end{cases}$$

Devolver (I, **LF** [])

Si hay eventos posibles, lo primero es ver si algún evento de los fijos coincide con el evento negado del requisito.

$$c_j = \exists\{\vec{v_j}\}([\sigma_J(r_r)](\sigma_J(C_{p_r \equiv p_j}) \wedge g_j \wedge \sigma_J(g_r))) \qquad \sigma_J = \sigma_j \cup \sigma_a$$

$\vec{v_j}$ son las variables que toman valores en p_j

$$\hat{C} = \mathop{\triangledown}_{j=1}^{m} c_j$$

Si $\|\hat{C}\|_\nu = True$, algún evento fijo coincide y este subrequisito es rechazado. En ese caso habría que devolver un $Reject$.

Para cada i tenemos:

$$I_i = \textbf{RIAV} \begin{cases} Es = E_a \\ Ev = a\ p_i\ [g_i] \\ Req = \neg a\ p_r\ [g_r]\{r_r\} \\ CV = \neg CF_i \end{cases}$$

$CF_i = \exists\{\vec{v_i}\}([\sigma_I(r_r)](\sigma_I(C_{p\equiv p_i}) \wedge g_i \wedge \sigma_I(g_r))) \qquad \sigma_I = \sigma_i \cup \sigma_a$

$\vec{v_i}$ son las variables que toman valores en p_i

Si $n = 1 \longrightarrow I = I_1$. Si $n > 1 \longrightarrow I = AndR \begin{cases} Es = E_a \\ Req = \neg a\ p_r\ [g_r]\{r_r\} \\ \bigcup\limits^{i} [I_i] \end{cases}$

$IvR = \textbf{RCond}(\hat{C},\ Reject,\ I)$

Devolver (IvR , **TF**)

- Req = a p_r $[g_r]$ $\{r_r\}$ $\qquad \tilde{a} = \{a\} \cup \{\hat{a}\}$

  Suponemos que en el estado actual del árbol (E_a) existen n eventos posibles "$\tilde{a}\ p_i\ [g_i]$" y m eventos negados "$\bar{a}\ p_j\ [g_j]$".

  Lo primero es ver si algún evento de los negados coincide con el evento posible del requisito.

  $c_j = \exists\{\vec{v_j}\}([\sigma_J(r_r)](\sigma_J(C_{p\equiv p_j}) \wedge g_j \wedge \sigma_J(g_r))) \qquad \sigma_J = \sigma_j \cup \sigma_a$

  $\vec{v_j}$ son las variables que toman valores en p_j

  $$\bar{C} = \mathop{\bigtriangledown}_{j=1}^{m} c_j$$

  Si $\|\bar{C}\|_\nu$=$True$, algún evento negado coincide y este subrequisito es rechazado. En ese caso habría que devolver un $Reject$.

Si $n = 0$ el requisito no puede cumplirse nunca.

$$I = \textbf{RIAV} \begin{cases} Es = E_a \\ Ev = Ninguno \\ Req = a\ p_r\ [g_r]\{r_r\} \\ CV = False \end{cases}$$

$IvR = \textbf{RCond}(\bar{C},\ Reject,\ I)$

Devolver $(IvR,$ **LF** $[])$

Si $n \neq 0$, para cada i tenemos:

$$I_i = \textbf{RIAV} \begin{cases} Es = E_a \\ Ev = a\ p_i\ [g_i] \\ Req = a\ p_r\ [g_r]\{r_r\} \\ CV = CV_i \end{cases}$$

$CV_i = \forall\{\vec{v_i}\}([\sigma_I(r_r)](\sigma_I(C_{p\equiv p_i}) \wedge g_i \wedge \sigma_I(g_r)))$ $\qquad \sigma_I = \sigma_i \cup \sigma_a$

$\vec{v_i}$ son las variables que toman valores en p_i

Si $n = 1 \longrightarrow I = I_1$. $\qquad$ Si $n > 1 \longrightarrow I = OrR \begin{cases} Es = E_a \\ Req = a\ p_r\ [g_r]\{r_r\} \\ \bigcup\limits^{i}[I_i] \end{cases}$

$IvR = \textbf{RCond}(\hat{C},\ Reject,\ I)$

$\Omega = \textbf{LF}\ (\bigcup\limits_{i=1}^{n}[(a\ p_i\ [g_i],\ CV_i,\ \sigma_i,\ \sigma_i(r_r))]\ )$

Devolver (IvR, Ω)

- Req = $R_1 \vee R_2$

  (IVR_1, Ω_1) = Get_IVR (R_1, E_a, σ_a)

  (IVR_2, Ω_2) = Get_IVR (R_2, E_a, σ_a)

  $$I = \mathbf{OrR} \begin{cases} Es = E_a \\ Req = R_1 \vee R_2 \\ [IVR_1; IVR_2] \end{cases}$$

  $$\Omega = \mathbf{OL}\ [\Omega_1; \Omega_2]$$

  Si uno de los CEA es **LV**, Ω es el otro directamente. Si ambos son **LV**, Ω es **LV**.

  Devolver (I, Ω)

- Req = $R_1 \wedge R_2$

  Aplicamos el algoritmo a cada uno de los dos elementos y componemos el resultado.

  (IVR_1, Ω_1) = Get_IVR (R_1, E_a, σ_a)

  (IVR_2, Ω_2) = Get_IVR (R_2, E_a, σ_a)

  $$I = \mathbf{AndR} \begin{cases} Es = E_a \\ Req = R_1 \wedge R_2 \\ [IVR_1; IVR_2] \end{cases}$$

  $$\Omega = \mathbf{AL}\ [\Omega_1; \Omega_2]$$

Si uno de los CEA es **LV**, Ω es el otro directamente. Si ambos son **LV**, Ω es **LV**.

Devolver (I, Ω)

- Req = $R_p \Longrightarrow R_c$

  Aplicamos el algoritmo a cada elemento y componemos el resultado.

  (IVR_p, Ω_p) = Get_IVR (R_p, E_a, σ_a)

  (IVR_c, Ω_c) = Get_IVR (R_c, E_a, σ_a)

$$I = \mathbf{Imp} \begin{cases} Es = E_a \\ Req = R_p \Longrightarrow R_c \\ (IVR_p, IVR_c) \end{cases}$$

  Devolver $(I, \mathbf{TF})$

- Req = $R_p \Longrightarrow\!\bigodot R_c$

  (IVR_p, Ω_p) = Get_IVR (R_p, E_a, σ_a)

  (IVR_c, Ω_c) = Get_IVR (R_c, E_{ant}, σ_a)

$$I = \mathbf{Imp} \begin{cases} Es = E_a \\ Req = Rp \Longrightarrow\!\bigodot Rc \\ (IVR_p, IVR_c) \end{cases}$$

  Devolver $(I, \mathbf{TF})$

- Req = Rp $\Longrightarrow\otimes$ Æ$\{\vec{v}\}$ Rc $\qquad (\otimes \in \{\bigcirc, @, ⓔ\})$

  (IVR_p, Ω_p) = Get_IVR (R_p, E_a, σ_a)

Calculamos $(C,\, IVR_c) = Eval_\Omega\ Æ\{\vec{v}\} \otimes R_c\ \Omega_p$

$$I = \textbf{Imp} \begin{cases} Es = E_a \\ Req = Rp \Longrightarrow \otimes Rc \\ (IVR_p, IVR_c) \end{cases}$$

Devolver $(I,\, \textbf{TF})$.

$(C_r, IVR) = Eval_\Omega\ Æ\{\vec{v}\} \otimes R_c\ \Omega_p$ **:=** En función de Ω_p

C_r es cierto si algún evento es posible en esa rama.

- $\Omega_p = \textbf{LV}$

  No puede haber estados de aplicabilidad. El requisito es semánticamente incorrecto. El algoritmo levanta una excepción y termina.

- $\Omega_p = \textbf{LF}\ []$

  $$I = \textbf{RIAV} \begin{cases} Es = E_a \\ Ev = Ninguno \\ Req = R_C \\ CV = False \end{cases}$$

  Devolver $(False,\, I)$.

- $\Omega_p = \textbf{LF}\ [(ev_1,\, c_1,\, \sigma_1,\, \sigma_1(r_r));\ ...\ ;\ (ev_I,\, c_I,\, \sigma_I,\, \sigma_I(r_r))]$

  $$\forall i \longrightarrow \begin{cases} (IVR_{c_i}, \Omega_{c_i}) = Get_IVR(R_c, E_{sig_i}, \sigma_a \cup \sigma_i) \\ \\ cu_i = \begin{cases} NoC & \text{Si no hay Æ} \\ FA(\sigma_i(\vec{v})) & \text{Si Æ} = \forall \\ EX(\sigma_i(\vec{v})) & \text{Si Æ} = \exists \end{cases} \end{cases}$$

$$\bigcirc \Rightarrow IVR_c = \textbf{AndF} \begin{cases} Es = E_a \\ Req = Rc \\ [(c_i, cu_i, IVR_{c_i})] \end{cases}$$

$$\textcircled{e} \Rightarrow IVR_c = \textbf{OrF} \begin{cases} Es = E_a \\ Req = Rc \\ [(c_i, cu_i, IVR_{c_i})] \end{cases}$$

$$\textcircled{a} \Rightarrow IVR_c = \textbf{RCond}(C,\, I_1,\, I_2) \qquad C = \bigvee_{i=1}^{I} c_i$$

$$I_1{=}\textbf{AndF} \begin{cases} Es = E_a \\ Req = Rc \\ [(c_i, cu_i, IVR_{c_i})] \end{cases} \qquad I_2{=}\textbf{AndF} \begin{cases} Es = E_a \\ Req = Rc \\ [] \end{cases}$$

Devolver $(C,\, IVR_c)$

- $\Omega_p = \textbf{TF}\ [a_1\ p_1\ [g_1],\ ....,\ a_I\ p_I\ [g_I]]$

$$\forall i \longrightarrow \begin{cases} (IVR_{c_i}, \Omega_i) = Get_IVR(R_c, E_{sig_i}, \sigma_a) \\ \\ cu_i = FA(\vec{v_i}) \end{cases}$$

$$\bigcirc \Rightarrow IVR_c = \textbf{AndF} \begin{cases} Es = E_a \\ Req = Rc \\ [(\forall\{\vec{v_i}\}(g_i), cu_i, IVR_{c_i})] \end{cases}$$

$$\textcircled{e} \Rightarrow IVR_c = \textbf{OrF} \begin{cases} Es = E_a \\ Req = Rc \\ [(\forall\{\vec{v_i}\}(g_i), cu_i, IVR_{c_i})] \end{cases}$$

$@ \Rightarrow IVR_c = \textbf{RCond}(\text{C},I_1,I_2) \qquad C = \bigvee_{i=1}^{I} g_i$

$I_1 = \textbf{AndF} \begin{cases} Es = E_a \\ Req = Rc \\ [(\forall\{\vec{v_i}\}(g_i), cu_i, IVR_{c_i})] \end{cases} \qquad I_2 = \textbf{AndF} \begin{cases} Es = E_a \\ Req = Rc \\ [] \end{cases}$

Devolver $(C,\ IVR_c)$

- $\Omega_p = \textbf{AL}\ [\Omega_1;\ \Omega_2]$

  $(C_i,\ IVR_{c_i}) = Eval_\Omega\ Æ\{\vec{v}\} \otimes R_c\ \Omega_i$

  $IVR_c = \textbf{AndF} \begin{cases} Es = E_a \\ Req = Rc \\ [(C_1, NoC, IVR_{c_1}); (C_2, NoC, IVR_{c_2})] \end{cases}$

  $C = C_1 \wedge C_2$

  Devolver $(C,\ IVR_c)$

- $\Omega_p = \textbf{OL}\ [\Omega_1;\ \Omega_2]$

  $(C_i,\ IVR_{c_i}) = Eval_\Omega\ Æ\{\vec{v}\} \otimes R_c\ \Omega_i$

  $IVR_c = \textbf{OrF} \begin{cases} Es = E_a \\ Req = Rc \\ [(C_1, NoC, IVR_{c_1}); (C_2, NoC, IVR_{c_2})] \end{cases}$

  $C = C_1 \vee C_2$

  Devolver $(C,\ IVR_c)$

- Req = Q $\{a_A\}$ (p_A)

  Al igual que en el algoritmo de verificación arrastraremos una estructura llamada *LRD* donde almacenaremos las instanciaciones de los requisitos desplegados hasta el momento en los estados de la rama actual.

- Q $\{a_A\}$ $(p_A) \in LRD_{E_a}$

  $$I = \textbf{Rec} \begin{cases} Es = Ea \\ Req = Q\{a_A\}(p_A) \end{cases}$$

  Devolver (I, **LV**)

  Si Q perteneciese a LRD_{E_a} pero los parámetros de la recursión no fuesen los mismos que los de la declaración, habría que levantar una excepción de fallo de modelo.

- Q $\{a_A\}$ $(p_A) \notin LRD_{E_a}$ $\longrightarrow$ Desplegamiento del requisito.

  $LRD_{E_a} = LRD_{E_a} \cup$ [Q $\{a_A\}$ (p_A)]

  Buscar Q en la lista de requisitos y extraer su fórmula F_Q.

  $(I,\Omega) = Get_IVR$ $(F_Q,\ E_a,\ \sigma_a)$

  Devolver (I,Ω)

9.4. Generación de las sugerencias

Como ya adelantamos, una vez generada la IVR que recoge la información sobre la verificación de una propiedad, es necesario analizar esa estructura, estudiar los diversos incumplimientos, generar sugerencias parciales y combinarlas de acuerdo a los constructores lógicos de la propiedad.

En este apartado describimos el algoritmo que se sigue para realizar ese análisis.

El algoritmo, de naturaleza recursiva, analizará la IVR recibida, contemplando todos los incumplimientos existentes y devolviendo una sugerencia en cada caso. Por tanto, lo primero que deberíamos establecer es el conjunto de posibles sugerencias que se pueden generar.

9.4.1. Tipos de sugerencias

Las sugerencias básicas serán:

Add_Ev(E**,**e_v**)** Añadir el evento e_v en el estado E.

Del_Ev(E**,**e_v**)** Eliminar el evento e_v del estado E.

Ace_Ev(E**,**e_o**,**e_d**)** Modificar el evento e_o del estado E para que coincida con el evento e_d.

Ale_Ev(E**,**e_o**,**e_i**)** Modificar el evento e_o del estado E para que no coincida con el evento e_i.

Las últimas dos sugerencias, **Ace_Ev** y **Ale_Ev**, sugieren modificaciones de eventos existentes en algún estado del árbol simbólico. Esas modificaciones afectarán a la guarda que condiciona la transición que etiqueta ese evento, probablemente a alguna restricción sobre los valores que pueden tomar las variables libres que almacenan la información que se intercambia con el entorno. Por el momento sólo se informa de que es necesario realizar alguna modificación en la dirección indicada, pero no se aportan sugerencias sobre esas modificaciones.

Cuando la IVR represente conjunciones o disyunciones, crearemos nuevas sugerencias **AndS** (también denotada por $\wedge_s$) y **OrS** (también denotada por $\vee_s$) para reflejar la conjunción o disyunción de las sugerencias que se deriven de los subelementos.

Además, tendremos un tipo de sugerencia llamado **Sin_Sug** (abreviado **SS**) que recogerá aquellas situaciones donde no existe ninguna sugerencia. En principio, existen seis posibles causas:

CVerif El requisito en cuestión se cumple.

CPremisa Las premisas de todos los futuros son falsas.

CRec(E**)** Existe recursión. Se acompaña el estado al que se recurre.

CRej El requisito es rechazado.

C$\emptyset$ El requisito es irrelevante para el rango de las variables que se ha especificado.

CNoS(E**,**Req**)** No se puede extraer ninguna sugerencia de modificación para que se verifique el requisito Req en el estado E.

En resumen, el tipo *Sugerencia* queda:

Sugerencia =
  Add_Ev(St,Ev)
| *Del_Ev(St,Ev)*
| *Ace_Ev(St,Ev,Ev)*
| *Ale_Ev(St,Ev,Ev)*
| *OrS(Sugerencia list)*
| *AndS(Sugerencia list)*
| *Sin_Sug(Causa)*

Causa =
  CVerif
| *CPremisa*
| *CRec(St)*
| *CRej*
| *C∅*
| *CNoS(St,Req)*

El algoritmo lo implementa la función *Get_Sugerencias*, que recibe una IVR y el CVS de cada variable (ν), devolviendo una sugerencia.

A continuación se describe el comportamiento del algoritmo para todos los valores posibles de una IVR. Los desarrollaremos todos agrupándolos por el constructor de la lógica LTCS en el cual tienen su origen.

Origen en Req = True [e]{r}

La IVR que recibiremos será un $RIAV$:

$$I = \mathbf{RIAV} \begin{cases} Es = E \\ Ev = Ninguno \\ Req = True\ [e]\{r\} \\ CV = exp \end{cases}$$

Si $\|exp\|_\nu = true$, el requisito se cumple y devolveremos $SS(CVerif)$.

Si $\|exp\|_\nu = \emptyset$, devolveremos $SS(C\emptyset)$.

De lo contrario, no se cumple y no ofrecemos ninguna sugerencia.

Por tanto, devolvemos $SS(CNoS(E, True\ [e]\{r\}))$.

Origen en Req = ¬ a p_r $[g_r]$ $\{r_r\}$

Podremos recibir:

- $I =$ **RIAV** $\begin{cases} Es = E \\ Ev = Ninguno \\ Req = \neg a\ p_r\ [g_r]\{r_r\} \\ CV = true \end{cases}$

  En este caso, el requisito se cumple siempre. Por tanto, devolvemos $SS(CVerif)$.

- **RCond**$(C,\ Reject,\ I) \longrightarrow$ Siendo $I =$ **RIAV** $\begin{cases} Es = E \\ Ev = a\ p_i\ [g_i] \\ Req = \neg a\ p_r\ [g_r]\{r_r\} \\ CV = exp \end{cases}$

  Si $\|C\|_\nu = True$, el requisito es rechazado y devolvemos $SS(CRej)$.

  Si $\|exp\|_\nu = True$, el requisito se cumple y devolvemos $SS(CVerif)$.

  Si $\|exp\|_\nu = \emptyset$, devolveremos $SS(C\emptyset)$.

  Si $\|exp\|_\nu = False$, el requisito no se cumple. Nuestra sugerencia es eliminar el evento "$a\ p_i\ [g_i]$" del estado E o modificarlo para que no coincida con el negado en el requisito ("$a\ p_r\ [g_r]\ \{r_r\}$").

  Devolveremos:

  $OrS(Del_Ev(E, a\ p_i\ [g_i]), Ale_Ev(E, a\ p_i\ [g_i], a\ p_r\ [g_r]\{r_r\}))$

- **RCond**(C, $Reject$, I) $\longrightarrow$ Siendo $I =$ **AndR** $\begin{cases} Es = E \\ Req = \neg a\ p_r\ [g_r]\{r_r\} \\ [I_i] \end{cases}$

  Si $\|C\|_\nu = True$, el requisito es rechazado y devolvemos $SS(CRej)$.

  De lo contrario, aplicaremos el algoritmo a cada uno de los I_i, obteniendo sus correspondientes sugerencias S_i, que podrán ser $SS(CRej)$, $SS(C\emptyset)$, $SS(CVerif)$ y sugerencias normales.

  - Si hay un $SS(CRej)$, el resultado será $SS(CRej)$.
  - Si todos son $SS(C\emptyset)$, el resultado será $SS(C\emptyset)$
  - Si todos son $SS(C\emptyset)$ ó $SS(CVerif)$, el resultado será $SS(CVerif)$
  - En caso contrario devolveremos un $AndS$ que contenga todas las sugerencias normales.

Origen en Req = a p_r $[g_r]$ $\{r_r\}$

Podremos recibir:

- **RCond**(C, $Reject$, $I_\emptyset$) $\longrightarrow$ Siendo $I_\emptyset =$ **RIAV** $\begin{cases} Es = E \\ Ev = Ninguno \\ Req = a\ p_r\ [g_r]\{r_r\} \\ CV = False \end{cases}$

  Si $\|C\|_\nu = True$, el requisito es rechazado y devolvemos $SS(CRej)$.

  En este caso, el requisito no se cumple. Devolveremos la sugerencia de añadir el evento "$a\ p_r\ [g_r]\ \{r_r\}$" al estado E. En caso de añadir tal evento, el usuario deberá especificar, completamente, el comportamiento del sistema a partir de ese evento.

  Devolveremos $Add_Ev(E, a\ p_r\ [g_r]\ \{r_r\})$

- **RCond**$(C,\ Reject,\ I)$ $\longrightarrow$ Siendo $I =$ **RIAV** $\begin{cases} Es = E \\ Ev = a\ p_i\ [g_i] \\ Req = a\ p_r\ [g_r]\{r_r\} \\ CV = exp \end{cases}$

  Si $\|C\|_\nu = True$, el requisito es rechazado y devolvemos $SS(CRej)$.

  Si $\|exp\|_\nu = true$, el requisito se cumple. Devolveremos $SS(CVerif)$.

  Si $\|exp\|_\nu = \emptyset$, devolveremos $SS(C\emptyset)$.

  Si $\|exp\|_\nu = false$, el requisito no se cumple. Devolveremos la sugerencia de añadir el evento "$a\ p_r\ [g_r]\ \{r_r\}$" al estado E del árbol o modificar el evento "$a\ p_i\ [g_i]$" del estado E para acercarlo a "$a\ p_r\ [g_r]\ \{r_r\}$".

  Devolveremos:

  $OrS(Add_Ev(E, a\ p_r\ [g_r]\{r_r\}), Ace_Ev(E, a\ p_i\ [g_i], a\ p_r\ [g_r]\{r_r\}))$

- **RCond**$(C,\ Reject,\ I)$ $\longrightarrow$ Siendo $I =$ **OrR** $\begin{cases} Es = E \\ Req = a\ p\ [e]\{r\} \\ [I_i] \end{cases}$

  Si $\|C\|_\nu = True$, el requisito es rechazado y devolvemos $SS(CRej)$.

  Aplicaremos el algoritmo a cada uno de los I_i, obteniendo sus correspondientes sugerencias S_i, que podrán ser $SS(CRej)$, $SS(C\emptyset)$, $SS(CVerif)$ y sugerencias normales.

  - Si hay un $SS(CVerif)$, el resultado será $SS(CVerif)$.
  - Si todos son $SS(C\emptyset)$, el resultado será $SS(C\emptyset)$
  - Si todos son $SS(C\emptyset)$ ó $SS(CRej)$, el resultado será $SS(CRej)$
  - En caso contrario devolveremos un OrS con todas las sugerencias normales.

Origen en recursiones

La IVR que recibiremos será:

$$I = \mathbf{Rec} \begin{cases} Es = E \\ Req = Q\{a_A\}(p_A) \end{cases}$$

Devolveremos $SS(CRec(E))$.

Origen en disyunciones

La IVR que recibiremos será:

$$I = \mathbf{OrR} \begin{cases} Es = E \\ Req = R_1 \vee R_2 \\ [IVR_i] \end{cases}$$

$S_i = Get_Sugerencias(IVR_i)$

Las distintas sugerencias se simplifican de acuerdo a las reglas:

- Una sola $S_i \longrightarrow S_i$
- Si hay un $SS(CVerif) \longrightarrow SS(CVerif)$.
- Se eliminan duplicados: $S_i + S_i \longrightarrow S_i$
- El resto se simplifican por pares de acuerdo a la tabla:

	CRej	**CNoS$_2$**	**CRec(E)**	**CRec($E\uparrow$)**	**SNormal**
C$\emptyset$	CRej	$CNoS_2$	C$\emptyset$	CRec($E\uparrow$)	SNormal
CRej		$CNoS_2$	CRej	CRec($E\uparrow$)	SNormal
CNoS$_1$		$\vee_s(NS_1,NS_2)$	$CNoS_1$	CRec($E\uparrow$)	$\vee_s(NS_1,SN)$
CRec(E)				CRec($E\uparrow$)	SNormal
CRec($E\uparrow$)				CRec(E_{min})	SNormal
SNormal$_1$					$\vee_s(SN_1, SN)$

$E\uparrow$ representa a cualquier estado anterior a E.

E_{min} es el estado que está antes en el árbol de los dos que se reciben.

Origen en conjunciones

La IVR que recibiremos será:

$$I = \textbf{AndR} \begin{cases} Es = E \\ Req = R_1 \wedge R_2 \\ [IVR_i] \end{cases}$$

$S_i = Get_Sugerencias(IVR_i)$

Las distintas sugerencias se simplifican de acuerdo a las reglas:

- Una sola $S_i \longrightarrow S_i$
- Si hay un $SS(CRej) \longrightarrow SS(CRej)$.
- Se eliminan duplicados: $S_i + S_i \longrightarrow S_i$
- El resto se simplifican por pares de acuerdo a la tabla:

	CRec(E)	**CRec($E\uparrow$)**	**C$\emptyset$**	**CNoS$_2$**	**SNormal**
CVerif	CVerif	CVerif	CVerif	$CNoS_2$	SNormal
CRec(E)		CRec($E\uparrow$)	C$\emptyset$	$CNoS_2$	SNormal
CRec($E\uparrow$)		CRec(E_{min})	CRec($E\uparrow$)	$CNoS_2$	SNormal
C$\emptyset$				$CNoS_2$	SNormal
CNoS$_1$				$\wedge_s(NS_1,NS_2)$	$\wedge_s(NS_1$,SN)
SNormal$_1$					$\wedge_s(SN_1$,SN)

Origen en un operador "⇒"

La IVR que recibiremos será:

$$I = \textbf{Imp} \begin{cases} Es = E \\ Req = R_p \Longrightarrow \otimes R_c \\ (IVR_p, IVR_c) \end{cases}$$

$S_p = Get_Sugerencias(IVR_p)$ $\qquad$ $S_c = Get_Sugerencias(IVR_c)$

Si $S_p = SS(C\emptyset)$, devolver $SS(C\emptyset)$.

Si $S_p \neq SS(CVerif)$, la premisa no se cumple. El requisito sí. Devolvemos $SS(CVerif)$.

Si $S_c = SS(CPremisa)$, devolver $SS(CNoS)$.

En otro caso, devolver S_c.

Origen en disyunciones de futuros

La IVR que recibiremos será:

$$I = \textbf{OrF} \begin{cases} Es = E \\ Req = R_C \\ [(c_i, cu_i, IVR_i)] \end{cases}$$

Cada trío de la lista representa a un futuro posible. En primer lugar, se evaluarán las premisas y se descartarán aquellos elementos cuya premisa no sea cierta.

Si todas las premisas son falsas no existirá ningún futuro posible para los valores indicados de las variables libres. Se devolverá $SS(CPremisa)$.

Si todas las premisas son falsas ó $\emptyset$, se devolverá $SS(C\emptyset)$.

De lo contrario, de cada elemento cuya premisa sea cierta se derivará una sugerencia S_i, teniendo en cuenta los cuantificadores (cu_i) de las variables libres que contiene.

$$S_i = Get_Sugerencias(IVR_i)$$

Las sugerencias así obtenidas (S_i) se simplificarán de acuerdo a las siguientes reglas:

- Una sola $S_i \longrightarrow S_i$

- Si hay un $SS(CVerif) \longrightarrow SS(CVerif)$.

- Se eliminan duplicados: $S_i + S_i \longrightarrow S_i$

- El resto, como en los casos anteriores, se simplifican por pares de acuerdo a la tabla:

	CRej	**CNoS**$_2$	**CRec**(E)	**CRec**($E\uparrow$)	**SNormal**
C$\emptyset$	C$\emptyset$	$CNoS_2$	CRec(E)	CRec($E\uparrow$)	SNormal
CRej		$CNoS_2$	CRec(E)	CRec($E\uparrow$)	SNormal
CNoS$_1$		$\vee_s(NS_1,NS_2)$	CRec(E)	CRec($E\uparrow$)	$\vee_s(NS_1,SN)$
CRec(E)				CRec($E\uparrow$)	SNormal
CRec($E\uparrow$)				CRec(E_{min})	SNormal
SNormal$_1$					$\vee_s(SN_1, SN)$

Origen en conjunciones de futuros

Las IVR que podemos recibir son:

- $I =$ **AndF** $\begin{cases} Es = E \\ Req = R_C \\ [(c_i, cu_i, IVR_i)] \end{cases}$

- **RCond**(C, I_1, I_2)

  Siendo $I_1 =$ **AndF** $\begin{cases} Es = E \\ Req = R_C \\ [(c_i, cu_i, IVR_i)] \end{cases}$ $I_2 =$ **AndF** $\begin{cases} Es = E \\ Req = R_C \\ [] \end{cases}$

En el segundo caso, si $\parallel C \parallel_\nu \neq True$, es que no hay futuros. En este caso devolveremos $SS(CPremisa)$.

En caso contrario, ambas posibilidades son iguales.

Se evaluarán las premisas y se descartarán aquellos elementos cuya premisa sea falsa ó $\emptyset$.

Si todas las premisas son falsas, se devolverá $SS(CVerif)$.

Si todas las premisas son falsas ó $\emptyset$, se devolverá $SS(C\emptyset)$.

$S_i = Get_Sugerencias(IVR_i)$

De lo contrario, de cada premisa cierta se derivará una S_i (teniendo en cuenta los cuantificadores cu_i), que se simplificarán de acuerdo a las reglas:

- Una sola $S_i \longrightarrow S_i$
- Si hay un $SS(CRej) \longrightarrow SS(CRej)$.
- Se eliminan duplicados: $S_i + S_i \longrightarrow S_i$
- El resto se simplifican por pares de acuerdo a la tabla:

	CRec(E)	**CRec**($E\uparrow$)	**C**$\emptyset$	**CNoS**$_2$	**SNormal**
CVerif	CRec(E)	CRec($E\uparrow$)	C$\emptyset$	$CNoS_2$	SNormal
CRec(E)		CRec($E\uparrow$)	CRec(E)	$CNoS_2$	SNormal
CRec($E\uparrow$)		CRec(E_{min})	CRec($E\uparrow$)	$CNoS_2$	SNormal
C$\emptyset$				$CNoS_2$	SNormal
CNoS$_1$				$\wedge_s(NS_1, NS_2)$	$\wedge_s(NS_1,$SN$)$
SNormal$_1$					$\wedge_s(SN_1, SN)$

Parte III

Conclusiones

Capítulo 10

Conclusiones y trabajo futuro

A lo largo de los capítulos precedentes de esta memoria hemos descrito el trabajo desarrollado en la presente tesis. Éste se ha centrado en el campo de los métodos formales y su aplicación al desarrollo de sistemas distribuidos.

La principal ventaja que se deriva de aplicar técnicas formales al desarrollo de un sistema es la posibilidad de realizar verificaciones del mismo a lo largo de todo su ciclo de vida. El empleo de notaciones matemáticas para su especificación permite eliminar ambigüedades y razonar sobre las propiedades del sistema desde fases muy tempranas. Ello ayuda a detectar los errores antes de que sus consecuencias se propaguen, lo que simplifica su reparación.

Sin embargo, el proceso de verificación de una especificación completa o de la corrección de una implementación respecto a su especificación suele ser una tarea muy costosa en tiempo y recursos. Por ello, es habitual que el desarrollo de un sistema con métodos formales se lleve a cabo mediante un proceso de refinamientos sucesivos. En ese proceso, es necesario verificar formalmente la corrección de cada refinamiento.

Muchos autores argumentan que las necesidades de las diversas fases del desarrollo son muy distintas y que, en consecuencia, no existe ningún método formal que sea apropiado para su aplicación a lo largo de todo el ciclo de vida del producto. Las técnicas utilizadas deben ser las más apropiadas para cada etapa de desarrollo, sus objetivos, nivel de detalle, etc.

En esta línea, existe un amplio consenso en considerar que en las fases iniciales de la especificación es conveniente utilizar técnicas orientadas a

propiedades (no constructivas), que permiten recoger y validar los requisitos de usuario con más fidelidad. Al avanzar el desarrollo y consolidarse el comportamiento del sistema, procede emplear técnicas constructivas, más idóneas para la estructuración del sistema de cara a la implementación.

LIRA es una herramienta orientada al desarrollo de sistemas distribuidos con métodos formales. Esta aplicación ofrece soporte para el diseño formal de un sistema siguiendo el principio de refinamientos sucesivos. Previamente a los trabajos realizados en esta tesis (ver apéndice A), LIRA ofrecía ayuda al desarrollo transformacional de especificaciones LOTOS. En concreto, permitía definir transformaciones automáticas sobre una especificación LOTOS y verificar formalmente que se mantenían las propiedades de interés.

Los mecanismos desarrollados a tal efecto parten de una arquitectura inicial del sistema descrita en el lenguaje de especificación formal LOTOS. Sin embargo, como ya hemos mencionado, una técnica constructiva como es LOTOS no es la opción más apropiada para comenzar la especificación formal del sistema (la captura de los requisitos de usuario).

El objetivo central de esta tesis ha sido enriquecer la funcionalidad de LIRA con mecanismos que ayuden a la captura de una arquitectura inicial que sirva como punto de partida para el proceso de refinamientos sucesivos.

Para ello, por estar orientados a las primeras fases del desarrollo, los mecanismos definidos e implementados están basados en lógica temporal, una técnica formal orientada a propiedades que favorece la captura y verificación de los requisitos de usuario.

Esta ampliación tan importante de la cobertura formal que LIRA ofrece al proceso de desarrollo de un sistema, ha sido aprovechada para llevar a cabo otros dos importantes objetivos, aunque de distinta naturaleza:

- Iniciar el porte de la funcionalidad original de la herramienta hacia el nuevo estándar E-LOTOS.

- Rediseñar la arquitectura de la herramienta para aprovechar las interesantes posibilidades que nos ofrecen los nuevos lenguajes de programación y tecnologías asociadas que han emergido con el rápido desarrollo que han experimentado las redes de ordenadores.

10.1. Contribuciones

Las aportaciones de esta tesis doctoral se articulan en torno a un procedimiento metodológico desarrollado para guiar la captura de la arquitectura inicial mencionada.

Este procedimiento, cuyas líneas generales presentamos en el capítulo 4, coordina varias etapas para obtener la descripción inicial del sistema de forma incremental. Cada fase tiene una duración indefinida y puede ser instanciada un número indeterminado de veces. Al sucederse las etapas, el diseñador formaliza los requisitos de usuario identificados, estudia su verificación en el sistema y, en caso necesario, modifica éste para lograr su cumplimiento.

El modelo del sistema que desarrollamos, un sistema de transiciones simbólico, evoluciona incorporando los diversos requisitos de usuario hasta alcanzar un estado en el que describe un comportamiento que satisface al diseñador. En ese momento, es automáticamente traducido a una especificación E-LOTOS.

Para realizar la automatización de las partes rutinarias de cada etapa, se han desarrollado una serie de notaciones y algoritmos que constituyen las principales contribuciones de esta tesis. Esas aportaciones han sido descritas a lo largo de los capítulos de la segunda parte de esta memoria y pueden resumirse en:

- En el capítulo 6 se ha desarrollado la lógica temporal **LTCS**, con el objetivo de especificar las propiedades del sistema que son objeto de nuestro interés. Esta lógica permite expresar de forma sencilla relaciones de causalidad entre diversos requisitos a lo largo del tiempo. Además, posibilita la realización de aseveraciones sobre el intercambio de información que lleva a cabo el sistema con su entorno: su naturaleza, el sentido en el que fluye, el rango de los valores intercambiados, etc.

  En el mismo capítulo se ofrece una clasificación de propiedades de uso frecuente. Esta clasificación obedece a criterios de homogeneidad funcional, basados en la conocida caracterización de seguridad y viveza. Las clases identificadas permiten la definición de arquetipos de propiedades, parametrizados en distintos elementos de la lógica, que pueden llegar a dispensar al diseñador del conocimiento de la lógica que subyace en los mecanismos de especificación y verificación.

- En el capítulo 8 se define un algoritmo de *model checking* para verificar

el cumplimiento en el sistema de las propiedades expresadas en la lógica LTCS.

Los cálculos del algoritmo involucran a las variables libres utilizadas para modelar el intercambio de información entre el sistema y su entorno. Por tanto, sus resultados serán expresiones de un lenguaje de datos que, en función de los valores que puedan tomar las variables libres, nos indicarán bajo qué condiciones el sistema cumple la propiedad.

Además, el algoritmo nos proporciona otro resultado, también una expresión de datos, que nos informa de la viabilidad de una modificación del sistema para cumplir la propiedad. Es decir, en caso de que el sistema no cumpla la propiedad, ese segundo resultado nos indica si el sistema puede ser modificado para que la cumpla. Si eso no es posible, es que existe una contradicción entre la propiedad en estudio y otra verificada con anterioridad.

- En el capítulo 9 se desarrolla un formalismo, también basado en *model checking*, para estudiar el incumplimiento de una propiedad y obtener, de forma automática, sugerencias sobre posibles modificaciones del sistema para solucionarlo.

  El algoritmo se describe en función de la representación del sistema, de los invariantes establecidos por las propiedades verificadas anteriormente y de la propiedad cuyo cumplimiento perseguimos.

  Las sugerencias aportadas consisten en modificaciones del conjunto de eventos posibles en los distintos estados del árbol simbólico que representa al sistema. Esas modificaciones abarcan la adición de nuevos eventos, la eliminación de eventos posibles o la modificación de los mismos para que coincidan o dejen de coincidir con las afirmaciones realizadas por la propiedad.

Todas las aportaciones indicadas en esta sección se han implementado sobre la herramienta LIRA. Asimismo, fruto de los trabajos de esta tesis se ha rediseñado la arquitectura de la herramienta. Tal como se describe ampliamente en el apéndice A, se ha dividido la aplicación en dos programas: un interfaz de usuario y un núcleo que gestiona la información e implementa la parte algorítmica. Ambos programas, que pueden residir en máquinas distintas, están enlazados por los mecanismos tradicionales de intercomunicación de procesos en la red Internet. Ello deriva en importantes ventajas:

- La separación de la parte gráfica de la algorítmica simplifica la programación, depuración y actualización del *software*.
- Ambos componentes tienen funcionalidades y necesidades completamente diferentes. Su separación permite la elección del lenguaje de programación más adecuado en cada caso.
- Se ha implementado el interfaz de usuario como un *applet* del lenguaje *Java*, lo que permite una distribución automática de las nuevas versiones.
- Dada la difusión actual de esta tecnología y su independencia de la máquina sobre la que se ejecuta, se ha incrementado notablemente la variedad de plataformas en las que se puede utilizar la herramienta.
- Permite situar el núcleo en máquinas con gran capacidad de procesado y acceder a ellas desde máquinas más simples situadas en cualquier lugar remoto.

10.2. Conclusiones

El desarrollo de sistemas distribuidos con métodos formales presenta importantes ventajas en cuanto a la comprensión del sistema y a la seguridad de que el resultado final es correcto. Sin embargo, es un proceso complejo y costoso en tiempo y recursos, que requiere de personal con alta preparación.

La disponibilidad de herramientas que automaticen y optimicen las tareas rutinarias es un objetivo indispensable para la consolidación de estas técnicas en los procesos de desarrollo de *software*. Además, es importante la definición de procedimientos que guíen a los diseñadores en su trabajo y les aporten pautas de comportamiento regulares.

LIRA es una herramienta que persigue tales objetivos, tratando de guiar al diseñador desde la captura de requisitos hasta la implementación final. Para ello, combina distintas técnicas formales, aplicando en cada fase del proyecto las más idóneas para resolver los problemas planteados.

En esta tesis doctoral se han definido e implementado los mecanismos que ofrece LIRA para ayudar al diseñador en la fase de captura de requisitos y obtención de la arquitectura inicial. Se ha aportado un procedimiento a seguir

y una serie de notaciones y algoritmos para lograr los objetivos de cada fase del procedimiento. Los mecanismos definidos e implementados ofrecen ayuda al diseñador en la especificación de las propiedades del sistema, el solapamiento y unificación de comportamientos, la verificación automática de propiedades temporales y el análisis de incumplimientos y búsqueda de las modificaciones oportunas.

El elemento sobre el que pivota la mayor parte del trabajo realizado es la lógica LTCS. La lógica temporal, y en especial la LTCS, se ha mostrado como un formalismo muy adecuado para la especificación y verificación de las propiedades funcionales del sistema (del modelo que representa al sistema).

La incorporación del intercambio de información con el entorno en la descripción del sistema dificulta la modelación y tratamiento de la recursión en los algoritmos de *model checking*. Los formalismos similares que se pueden encontrar en la literatura optan por imponer limitaciones en alguno de los elementos involucrados: los modelos de sistemas analizables, el tipo de propiedades verificables, los tipos de recursiones cubiertas, etc. Esta última limitación ha sido la elegida por nosotros, al entender que todavía posibilita la verificación de un espacio de sistemas muy amplio.

Un principio fundamental que ha regido la implementación desarrollada ha sido el tratar de abstraer al diseñador del soporte teórico empleado, dispensándolo del conocimiento de las herramientas matemáticas que articulan las ayudas ofrecidas. Para lograr este objetivo, se ha hecho especial hincapié en dos aspectos:

- Se ofrecen mecanismos de generación automática de las construcciones lógicas utilizadas en los procesos de verificación. Para ello se han definido una serie de arquetipos de uso frecuente, parametrizados en ciertos campos, para que el diseñador elija una propiedad en función de la finalidad perseguida y sea la herramienta la que la materialice, solicitando la identificación de los parámetros.

- Se ofrecen herramientas para interpretar los resultados de los algoritmos y clarificar los distintos matices de éstos en función de la información que el sistema puede intercambiar con el entorno.

10.3. Trabajo futuro

Los resultados obtenidos en esta tesis doctoral son un paso más en el proceso de construcción de la herramienta LIRA. Las líneas de trabajo futuro que consideramos más interesantes van encaminadas tanto a mejorar y ampliar las contribuciones de esta tesis como a enriquecer la funcionalidad global de la herramienta en diferentes aspectos.

Entre las más interesantes destacan:

- La modelación de la información que el sistema intercambia con su entorno es fundamental para lograr una buena capacidad expresiva en los procesos de especificación y verificación. Es necesario ampliar los tipos de datos que contempla LIRA, o incluso establecer un mecanismo abierto que permita el uso de librerías de datos externas, con sus elementos y reglas de evaluación.
- Los tipos de recursión que cubre el algoritmo de verificación propuesto en el capítulo 8 limitan su aplicación. Sería interesante ampliar el algoritmo en ese sentido, estableciendo mecanismos para detectar y decidir sobre recursiones que involucren parámetros dinámicos.
- El conjunto de sugerencias ofrecido en el capítulo 9 abarca la adición, eliminación y modificación de los eventos del árbol simbólico que representa al sistema. En el caso de la modificación se aporta una información parcial acerca de su objetivo. Sería de mucha utilidad ampliar los algoritmos allí propuestos para que se calculasen sugerencias sobre las modificaciones de las guardas de los eventos (las restricciones a los valores de las variables) para que el sistema cumpla la propiedad bajo estudio.
- En el diseño o estudio de sistemas de complejidad elevada, la eficiencia de los algoritmos es de gran importancia para su aplicabilidad. Por el carácter experimental del trabajo realizado, todavía no se ha procedido a un análisis de la eficiencia y posibles optimizaciones de los algoritmos presentados. Esa tarea supone una importante línea de trabajo a corto plazo.
- Sería interesante ampliar la funcionalidad de la herramienta para poder estudiar las propiedades no funcionales del sistema en desarrollo, por

ejemplo su eficiencia. Para ello, es necesaria una extensión de la lógica y demás formalismos con el objeto de integrar constructores que permitan una caracterización cuantitativa del paso del tiempo.

- El ámbito de aplicación de LIRA son los sistemas distribuidos. Consideramos muy interesante articular mecanismos en LIRA que permitan la integración de varios sistemas diseñados por separado y la verificación de propiedades del conjunto, especialmente las que no están presentes o no pueden ser estudiadas en los subsistemas por separado.

Parte IV

Apéndices

Apéndice A

La herramienta LIRA

A.1. Introducción

En esta tesis se ha presentado un procedimiento metodológico para la captura de la arquitectura inicial de un sistema distribuido. El procedimiento identifica una serie de etapas a seguir hasta alcanzar el objetivo y un conjunto de tareas a realizar en cada etapa. Para articular el procedimiento se han definido una serie de notaciones y formatos de representación que son utilizados por los algoritmos que automatizan las tareas propuestas.

El procedimiento descrito, aunque conceptualmente sencillo, entraña una dificultad práctica considerable en cuanto se incrementa la magnitud del sistema a desarrollar. La cantidad de información a manejar y el alto número de posibilidades a contemplar hace inviable su tratamiento sin la ayuda del ordenador.

En mayor o menor medida, en todas las etapas del procedimiento descrito es necesario el empleo de herramientas que deleguen en el ordenador las tareas rutinarias como puedan ser los procedimientos de solapamiento, verificación, modificación y síntesis.

Por ello, el procedimiento descrito (junto con las notaciones y algoritmos presentados) se ha implementado en la herramienta **LIRA** [PA95] (**L**OTOS **I**nteractive **R**easoning **A**id).

LIRA es un entorno transformacional que sirve de apoyo al desarrollo de sis-

temas distribuidos mediante el empleo de métodos formales. LIRA da soporte al desarrollo transformacional de sistemas desde la fase de captura de requisitos hasta la de implementación, eximiendo al diseñador de tareas rutinarias y permitiéndole concentrarse en los aspectos creativos del proceso.

A.2. El entorno transformacional LIRA

Con anterioridad a esta tesis doctoral, la funcionalidad de la herramienta LIRA se concentraba en el desarrollo transformacional de especificaciones LOTOS.

Siguiendo el modelo habitual de desarrollo con métodos formales descrito en el capítulo 1, el punto de partida del proceso era una especificación LOTOS que representaba la arquitectura inicial del sistema. Esa descripción inicial se sometía a una serie de refinamientos sucesivos que introducían en la especificación toda la información necesaria para realizar una implementación correcta.

Este enriquecimiento podía tener por objeto la estructuración del sistema para una identificación de subcomponentes, el incremento del nivel de detalle con el que se describe cada uno de ellos, un cambio de estilo en la especificación [VSSB89], etc.

Cada uno de esos refinamientos era formalmente verificado de acuerdo al conjunto de propiedades que deseábamos que se mantuviesen. Si era necesario, se podían sintetizar automáticamente bloques de comportamiento LOTOS que satisfacían un conjunto de propiedades.

El resultado final era una especificación LOTOS con una estructura y nivel de detalle suficientes para realizar una implementación directa, en muchos casos semiautomática.

A.2.1. Funcionalidades previas de LIRA

Los mecanismos que LIRA implementaba para lograr el objetivo reseñado se describen de forma exhaustiva en [PA95]. Allí pueden encontrarse las distintas notaciones y procedimientos desarrollados, sus fundamentos matemáticos y sus detalles de implementación.

En líneas generales, las distintas facilidades que la herramienta ofrecía incluyen:

- Una librería de transformaciones generales clasificadas por tipos. Esta librería es de carácter abierto, ampliable por el usuario (aunque bajo su responsabilidad) y fomenta la reutilización de transformaciones previas.
- Un lenguaje para especificar reglas de transformación entre especificaciones (o partes de especificaciones) LOTOS. Este lenguaje incluye unos elementos denominados *metavariables* que permiten la aplicación de reglas a comportamientos parciales.
- Un procedimiento automático para aplicar las anteriores reglas de transformación, siempre en función del cumplimiento de ciertos predicados definibles por el usuario, denominados condiciones de aplicabilidad.
- Un lenguaje de especificación de propiedades de comportamientos descritos mediante LOTOS. Este lenguaje, empleando la lógica temporal *μ-calculus*, permitía definir las propiedades de seguridad y viveza que deseábamos que el sistema cumpliese.
- Un algoritmo, basado en el *model checking* clásico y la construcción de un *tableau*, para verificar automáticamente el cumplimiento de las propiedades anteriores.
- Un procedimiento para, a partir de una o varias propiedades especificadas en el lenguaje reseñado, sintetizar automáticamente comportamientos LOTOS que cumplan esas propiedades.

A.2.2. Implementación

La implementación de esta versión inicial del entorno transformacional LIRA se llevó a cabo en el lenguaje funcional $CAML$ (versión 3.1) [MC98].

La estructura del programa estaba compuesta de una serie de módulos que implementaban las operaciones descritas, un núcleo que los coordinaba y un interfaz gráfico que mostraba los resultados, desarrollado mediante una librería X-Window integrada en la distribución del lenguaje.

La organización modular del entorno puede apreciarse en la figura A.1. Puede observarse que el núcleo coordina el interfaz, la base de datos y los tres grandes módulos funcionales de los que ya se ha hablado: el encargado de la aplicación de las reglas de transformación entre unidades LOTOS, el dedicado a la implementación del algoritmo de verificación de propiedades y el que articula la síntesis de comportamientos LOTOS con unas determinadas propiedades.

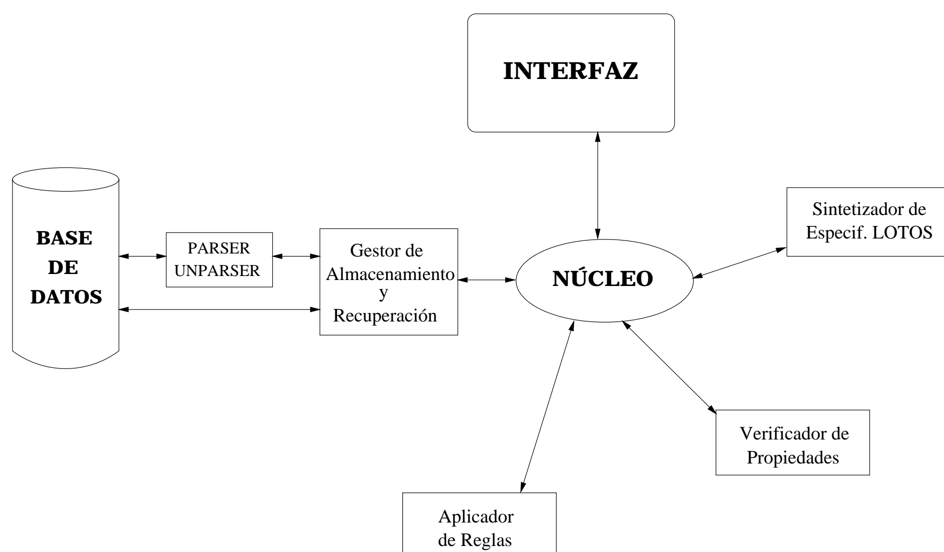

Figura A.1: Antigua organización modular de LIRA

A.3. Funcionalidades añadidas a LIRA

Como hemos visto hasta el momento, la funcionalidad original de la herramienta LIRA daba soporte al procedimiento de refinamientos sucesivos de una especificación LOTOS. El punto de partida era una especificación LOTOS incompleta que representaba la arquitectura inicial.

Fruto de los trabajos llevados a cabo en esta tesis, la herramienta ha sido modificada para incluir una serie de módulos que faciliten la captura de esa arquitectura inicial del sistema a partir de los requisitos de usuario.

Con este fin, hemos implementado en LIRA las notaciones y algoritmos propuestos para articular el procedimiento descrito en esta tesis. El resultado final será una especificación ELOTOS que describirá el comportamiento inicial del sistema en desarrollo y que servirá como punto de partida para el proceso de refinamientos sucesivos al cual LIRA ya daba soporte.

Los mecanismos que han sido implementados en LIRA para ayudar en la obtención de la arquitectura inicial del sistema incluyen las notaciones y algoritmos presentados en esta tesis. En concreto:

- Hemos añadido mecanismos gráficos para la especificación de los eventos posibles (identificador y tipo de información asociada) y de las trazas iniciales que conocemos del sistema. Para ello, se emplea la lógica LTCS descrita en el capítulo 6.

  Mediante el procedimiento descrito en el capítulo 7, la herramienta solapa las citadas trazas y construye un sistema de transiciones simbólico que comprende todos los comportamientos posibles que describen las trazas.

  Hemos añadido mecanismos de representación gráfica para observar más fácilmente todos los detalles de éste árbol simbólico.

- También mediante requisitos de la lógica LTCS, LIRA nos proporciona mecanismos (gráficos y textuales) para la especificación y almacenamiento de las propiedades temporales que creemos que el sistema debe cumplir.

  Hemos incluido en LIRA una serie de arquetipos de propiedades a los cuales podemos acogernos para especificar propiedades muy comunes. En este caso, sólo tendremos que elegir el modelo correspondiente e identificar los parámetros que caracterizan a nuestra propiedad.

- Se ha implementado en LIRA el algoritmo de verificación de propiedades descrito en el capítulo 8 para averiguar si el prototipo actual cumple los requisitos especificados.

  Generalmente, el resultado de la verificación será una expresión que dependerá de los valores de los datos que se intercambien con el entorno. Estas expresiones contendrán constructores y operadores propios del formalismo desarrollado y, por tanto, ajenos a la lógica tradicional. Ello implica una imposibilidad de evaluación del resultado para los usuarios potenciales, no introducidos en la notación. Para solventarlo, ofrecemos

mecanismos gráficos interactivos para averiguar el resultado en función de los valores especificados y establecer los rangos de esas variables para los cuales la verificación es positiva.

- En caso de no verificarse una propiedad, a través de la implementación del algoritmo descrito en el capítulo 9, LIRA nos aporta información acerca del incumplimiento, ofreciéndonos sugerencias acerca de posibles modificaciones para conseguir que la representación del sistema verifique la propiedad.

- Hemos articulado los mecanismos apropiados para que LIRA almacene toda la información necesaria en cada etapa para permitir (cada vez que modificamos el sistema) una revalidación automática de las propiedades verificadas hasta el momento. La propia herramienta realiza automáticamente el chequeo de integridad sobre las propiedades ya verificadas. Esto facilitará la documentación del sistema, al quedar registradas todas las fases por las que pasa, las propiedades que verificaba cada una y los motivos que provocaron la creación de una nueva fase.

- A partir de la representación final, LIRA sintetiza una especificación LOTOS cuyo comportamiento constituye la arquitectura inicial buscada.

  Debido a la cercana aprobación del nuevo estándar de LOTOS, denominado ELOTOS, todos los mecanismos de ayuda que LIRA proporciona para la fase de refinamientos sucesivos están en fase de actualización a la sintaxis del nuevo lenguaje. Por ello, en realidad, este resultado final se entrega ya en forma de una especificación ELOTOS.

A.4. Arquitectura de LIRA

Dentro de los trabajos llevados a cabo para implementar en LIRA los resultados de esta tesis, se ha modificado su arquitectura. Hemos dotado al entorno transformacional LIRA de una arquitectura distribuida. Su funcionalidad está dividida, fundamentalmente, en dos módulos:

Núcleo Un programa escrito en el lenguaje funcional *Camllight* [MC98] que almacena toda la información del sistema en desarrollo e implementa diversos algoritmos para procesar esa información: solapamiento de trazas

en un STS, verificación de una propiedad temporal, cálculo de sugerencias sobre modificaciones, los mecanismos ya existentes dedicados al proceso de refinamientos sucesivos, etc.

Interfaz Un *applet* escrito en el lenguaje *Java* que se encarga de la representación gráfica de la información y de todos los aspectos involucrados en la comunicación con el usuario.

El núcleo, que puede estar ejecutándose en una máquina distinta a la que utiliza el usuario para interaccionar a través del interfaz, puede atender a varios usuarios al mismo tiempo, disponiendo de un registro de información distinto para cada usuario, en donde se almacenan los datos del sistema que está desarrollando.

Cada usuario ejecutará el interfaz gráfico en su máquina, el cual establecerá una comunicación[1] con el núcleo para solicitarle información, enviársela, ordenarle operaciones sobre los datos, etc.

En caso de que ambos programas se ejecuten en máquinas distintas, el único requisito para un correcto funcionamiento es que ambas máquinas estén interconectadas a través de un enlace TCP/IP.

La arquitectura del sistema puede verse en la figura A.2.

En la citada figura puede observarse el modo de trabajo habitual. En primer lugar se ejecuta un navegador que implemente la máquina virtual *Java* y se realiza una solicitud a un servidor WWW que ofrezca el interfaz de LIRA. El servidor devolverá una página con el *applet* que implementa el interfaz. En ese momento tendríamos que indicarle al *applet* la dirección IP y puerto TCP del núcleo de LIRA con el que queremos trabajar (que tampoco tiene por qué coincidir con la máquina donde reside el servidor WWW). Como el *applet*, debido a restricciones de seguridad de la máquina virtual *Java*, sólo puede comunicarse con la máquina de la que procede, también es necesario proporcionar el puerto TCP de un programa puente que retransmitirá las solicitudes al núcleo de LIRA[2].

El enfoque descrito ofrece importantes ventajas:

[1]Comunicación realizada vía *sockets* TCP

[2]El porte a la versión 2.0 de *Java*, que implementa politicas de seguridad definibles por el usuario, permitirá eliminar este programa.

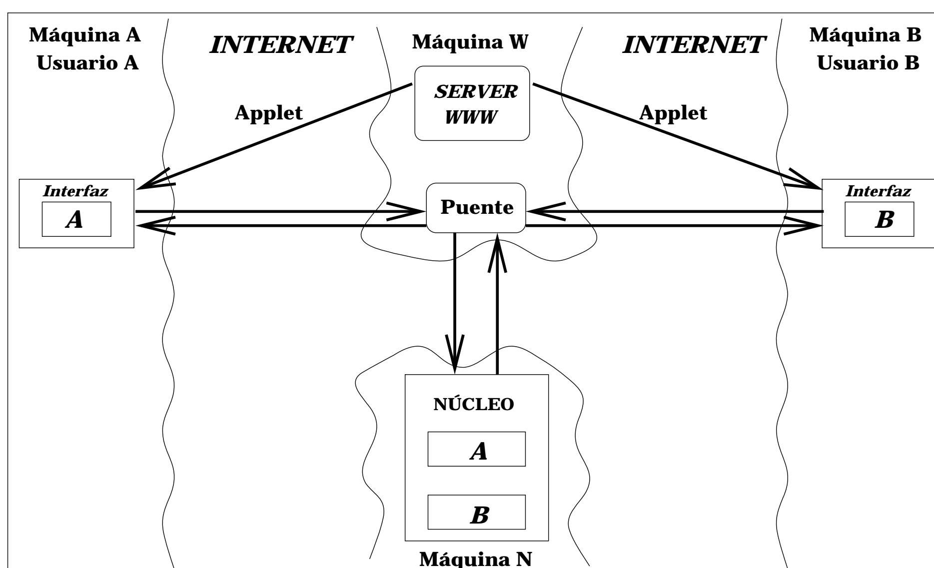

Figura A.2: Arquitectura de LIRA

- Permite una actualización automática del *software* del interfaz, ya que el *applet* se envía al cliente cada vez que inicia una sesión de trabajo.

- Existe una separación total entre las partes gráfica y algorítmica de la herramienta, simplificando la programación de ambas partes y permitiendo la elección del lenguaje de programación más adecuado en cada caso.

- Permite ubicar el núcleo en máquinas con gran capacidad de procesado, necesario para la parte algorítmica, y acceder a él desde máquinas más sencillas.

- El hecho de separar el núcleo del interfaz y programar este último en *Java* amplía notablemente el universo de máquinas que pueden ser empleadas para utilizar LIRA[3].

[3]La antigua implementación de LIRA limitaba su utilización a máquinas con sistema operativo Unix.

A.5. Implementación del núcleo

Como ya se ha comentado, el núcleo de LIRA está implementado en el lenguaje funcional *Camllight*[4] (versión 0.74) sobre el sistema operativo Unix (o Linux).

El lenguaje *Camllight*, refinamiento de $CAML$ (empleado para la primera versión de la herramienta) ofrece importantes ventajas respecto a su predecesor que han permitido solventar alguno de los problemas identificados en su momento y descritos en [PA95]:

- Su implementación ha sido ampliamente optimizada y, por ello, precisa máquinas bastante menos potentes para obtener eficiencias razonables.
- Implementa todas las facilidades modulares de los lenguajes modernos.
- Facilita la creación de analizadores léxicos y sintácticos mediante librerías que implementan el modelo $Lex/Yacc$.
- Implementa mecanismos de depuración.
- Proporciona interfaces con el lenguaje C para la inclusión de rutinas realizadas en este lenguaje.

Como tal lenguaje funcional, *Camllight* posee una serie de ventajas que lo hacen muy apropiado para la programación de herramientas que hagan un uso intensivo de computación simbólica y tipos de datos abstractos, como es el caso del entorno transformacional LIRA. Entre ellas:

- Es un lenguaje muy seguro. El compilador realiza un chequeo estático exhaustivo de todos los tipos de datos y el ejecutor simbólico maneja automáticamente la memoria, eximiendo al usuario de la responsabilidad de su asignación y liberación.
- Evaluación de expresiones en lugar de computación de estados. Aunque el modo habitual de trabajo es *"evaluación por valor"* (evaluación inmediata de expresiones), el lenguaje permite el uso de la llamada *"evaluación perezosa"* (no evaluar una expresión hasta que se necesite). Esto permite la utilización de estructuras potencialmente infinitas.

[4]Institut National de Recherche en Informatique et Automatique (INRIA), Francia.

- Ausencia de asignaciones, bucles, punteros y otros elementos que suelen provocar efectos laterales (aunque en realidad estos constructores sí están disponibles en el lenguaje si el usuario quiere programar en un estilo más próximo a los lenguajes imperativos).

- Potentes mecanismos recursivos, incluyendo a las funciones como un elemento de primer orden.

- Extensa biblioteca de funciones de uso general. Esta biblioteca incluye tipos de datos complejos como árboles, colas, tablas *hash*, pilas, etc.

- Diversas características que facilitan la programación, como: excepciones, polimorfismo, reconocimiento de patrones (*pattern matching*), etc.

En la figura A.3 puede observarse el diagrama de bloques del núcleo de la nueva implementación de LIRA. En ella se pueden identificar tres grandes subsistemas controlados por un módulo principal:

- La parte dedicada a implementar el procedimiento de captura de la arquitectura inicial.

  En ella podemos identificar un módulo por cada una de las tareas principales que soporta LIRA: solapamiento de las trazas en un STS, verificación de propiedades temporales, aportación de sugerencias sobre la modificación del sistema y traducción de éste a una especificación ELOTOS.

- La parte dedicada al proceso de refinamientos sucesivos.

  Podemos observar un módulo dedicado a las transformaciones entre elementos de una especificación LOTOS, otro dedicado a la verificación de propiedades sobre comportamientos LOTOS y un tercero dedicado a sintetizar comportamientos LOTOS que cumplan una serie de propiedades.

- La parte dedicada al control y almacenamiento de los datos.

  Aquí se puede observar un módulo que contiene un tipo de datos abstracto por cada objeto que maneja LIRA: especificaciones LOTOS y ELOTOS, sistemas de transiciones simbólicos, reglas de transformación, propiedades temporales, etc.

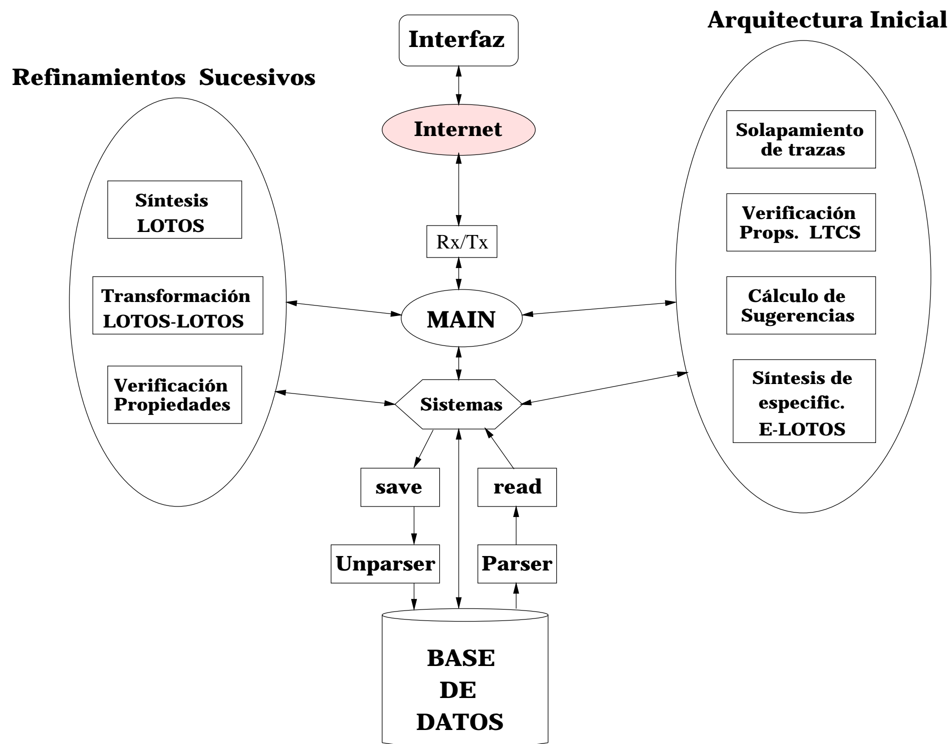

Figura A.3: Núcleo de LIRA

Además, podemos ver módulos que se encargan de la conversión entre el tipo de datos interno de cada objeto y la representación textual que le corresponde, según la sintaxis conocida en cada caso (especificaciones LOTOS, lógica LTCS, expresiones, etc.).

A.5.1. Evolución de la arquitectura inicial

Dentro del módulo encargado de almacenar la representación abstracta de todos los datos del entorno, existe una estructura que almacena la evolución de la arquitectura inicial del sistema. Esta estructura contiene todas las fases desarrolladas, desde el primer prototipo fruto del solapamiento de las trazas iniciales hasta la fase final de traducción a una especificación ELOTOS.

Esta estructura, como puede verse en la figura A.4, no sólo almacena la

representación del sistema en esa etapa, sino que también contiene información acerca de las propiedades que verifica esa representación. En concreto, la información que acompaña al STS de cada etapa consiste en la lista de propiedades que ya verificaba el STS de la etapa anterior, la propiedad que causó la generación de esta nueva etapa y la lista de propiedades que se han verificado a continuación.

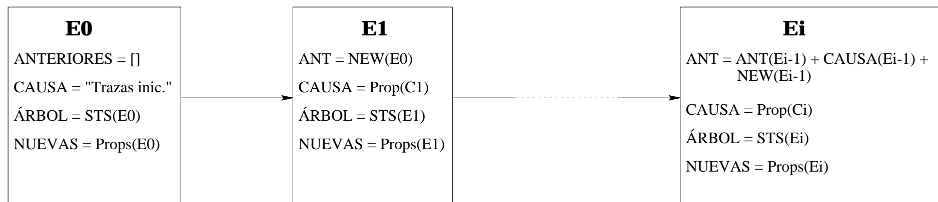

Figura A.4: Etapas del desarrollo

Para cada una de las propiedades mencionadas, LIRA almacenará los resultados obtenidos en su momento en la verificación y los CVS de cada variable que especificó el usuario. De esta forma, cada vez que realicemos un cambio en el sistema, la herramienta puede comprobar de forma automática si se mantiene la integridad del desarrollo y todavía se cumplen las propiedades que se han verificado en etapas previas.

A.6. Interfaz de LIRA

En la figura A.5 puede verse el interfaz de LIRA.

Como ya se ha mencionado, se trata de un *applet* del lenguaje *Java* que puede ejecutarse en cualquier máquina que esté comunicada con el núcleo a través de un enlace TCP/IP.

Las distintas órdenes que puede generar el usuario se enviarán al núcleo a través de un *socket* TCP. Éste las ejecutará y enviará la repuesta por el mismo medio al interfaz, quien presentará la información al usuario de una forma adecuada.

En la figura A.5 podemos observar que en el interfaz existen una serie de menús desplegables, cada uno ofreciendo un grupo homogéneo de tareas:

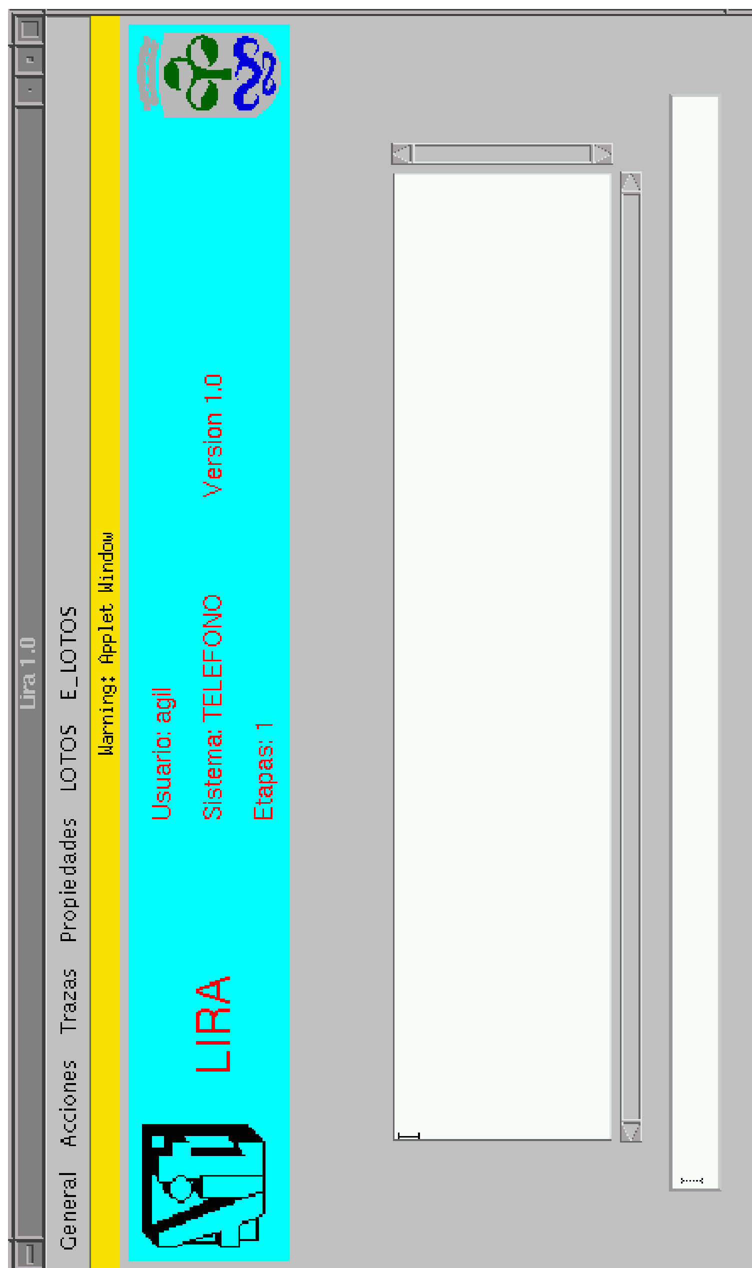

Figura A.5: Interfaz de LIRA

General. Ofrece comandos de carácter diverso, como pueden ser: conectarse, desconectarse, continuar un sistema ya empezado, iniciar un nuevo sistema, solicitar un resumen informativo sobre el sistema en desarrollo, cambiar el idioma de la aplicación, almacenar en disco toda la información modificada, desplegar el mecanismo de ayuda, etc.

Eventos. Ofrece una serie de posibilidades para trabajar con los eventos del sistema en desarrollo: añadir, borrar, mostrar, etc.

Trazas. Nos permite acceder a la captura y solapamiento de las trazas iniciales. Podremos añadir trazas, borrarlas, ver una representación gráfica del árbol simbólico que se genera, etc.

Propiedades. Mediante este menú podremos trabajar con todo lo relacionado con las propiedades temporales de la lógica LTCS: cargar nuevas propiedades, construirlas a partir de los arquetipos proporcionados por la herramienta, construir nuevos arquetipos, proceder a la verificación de una propiedad, analizar los resultados, etc. También podremos ver la representación gráfica del STS de la etapa actual y solicitar sugerencias sobre las modificaciones necesarias para que se cumpla una propiedad.

LOTOS y ELOTOS. Estos dos menús poseen la misma funcionalidad, uno para LOTOS y otro para ELOTOS, la extensión del lenguaje que está en proceso de normalización[5]. Con ellos se puede sintetizar la especificación que describe el STS de la etapa actual del desarrollo y comenzar el proceso de refinamientos sucesivos, crear nuevas reglas, aplicar transformaciones, especificar y verificar propiedades, sintetizar comportamientos LOTOS, etc.

[5]Algunas de las funcionalidades descritas todavía están en proceso de implementación en la versión de ELOTOS

Apéndice B

Ejemplo de desarrollo

En este apéndice vamos a desarrollar un ejemplo en donde se pueden observar las distintas fases por las que pasa la captura de la arquitectura inicial según el procedimiento descrito en esta tesis.

En concreto, el sistema que construiremos y verificaremos formalmente es el servidor de recursos cuyas trazas iniciales especificamos y solapamos en el capítulo 7.

Los resultados que se muestran a lo largo del apéndice han sido obtenidos con la herramienta LIRA, según la implementación que hemos llevado a cabo de los algoritmos descritos en los distintos capítulos de esta tesis.

A lo largo de las próximas secciones seguiremos los pasos que dicta el procedimiento general. En cada uno de ellos, trataremos de elegir diversos objetivos que permitan observar las distintas facetas de la ayuda que LIRA puede ofrecernos para construir la arquitectura inicial, haciendo uso de los mecanismos desarrollados en esta tesis.

El resultado final que se obtiene es una especificación ELOTOS que recoge el comportamiento del servidor de recursos, descrito mediante la ordenación válida de los eventos que espera del entorno y de los que puede generar por si mismo.

B.1. Identificación de los eventos

El primer paso en la construcción de un sistema es la identificación de los eventos que modelan su comportamiento y de la información que permiten intercambiar con el entorno.

Tal como vimos en el capítulo 7, los eventos que hemos identificado inicialmente en este sistema son:

Connect Identifica la llegada al servidor de una solicitud de conexión. Lleva asociado el identificador de la sesión que se abre, cuyo tipo denominaremos *TipoSes*.

Confirm Representa la respuesta del servidor confirmando la aceptación de la conexión. En ella irá el identificador de la conexión confirmada.

Request Significa la llegada al servidor de una petición de un recurso. La información que trae es el identificador del recurso, cuyo tipo denominaremos *TipoObj*.

Answer Se emplea para contestar a la solicitud anterior. El tipo de información que lleva asociada será *TipoAns* y consistirá un par de elementos: el resultado de la solicitud y un campo de datos.

Ack Representa la llegada de un asentimiento. Trae como información el identificador del recurso que se asiente (tipo *TipoObj*).

Release Este evento representa la liberación de la conexión y lleva asociado el identificador de la sesión que se cierra.

Tout Indica el vencimiento de una temporización de espera.

Nack Representa la llegada de un indicador de error en la transmisión del recurso. La información que lleva asociada es el número de la trama que llegó mal.

Query Se emplea para preguntar al cliente por el resultado de la transmisión del recurso en caso de que venza la temporización de espera por el asentimiento. La información que lleva aparejada es el identificador del recurso.

Tal como se mencionó en el capítulo 4, este conjunto de eventos no tiene por qué ser definitivo. De hecho, a lo largo de las próximas secciones, procederemos a añadirle varios elementos en función de las necesidades que se presenten.

B.2. Solapamiento de las trazas iniciales

Una vez identificados los eventos, el procedimiento establece la especificación de las trazas iniciales del sistema, aquellas que conocemos desde un primer momento.

En el capítulo 7 las trazas iniciales que conocíamos del comportamiento del sistema eran cinco.

En primer lugar, especificamos la traza que describe una solicitud y respuesta sin errores:

$$T0 := connect\ ?ses \Longrightarrow \bigcirc(confirm\ !ses \Longrightarrow \bigcirc(\ request\ ?obj \Longrightarrow \bigcirc(\\ answer\ (!OK, !all_data(obj))\ [available(obj)] \Longrightarrow \bigcirc(\\ ack\ ?v\ [v = obj] \Longrightarrow \bigcirc(release\ !ses \Longrightarrow \bigcirc\ T0)))))$$

La traza que describe el vencimiento de la temporización antes de la llegada de la solicitud del recurso:

$$T1 := connect\ ?op \Longrightarrow \bigcirc(confirm\ !op \Longrightarrow \bigcirc(\\ tout \Longrightarrow \bigcirc(release\ !op \Longrightarrow \bigcirc\ T1)))$$

La siguiente traza describe la solicitud de un recurso inexistente (resultado NF):

$$T2 := connect\ ?con \Longrightarrow \bigcirc(confirm\ !con \Longrightarrow \bigcirc(\ request\ ?rec \Longrightarrow \bigcirc(\\ answer\ (!NF, !NULL)\ [\neg available(rec)] \Longrightarrow \bigcirc(\\ release\ !con \Longrightarrow \bigcirc\ T2))))$$

La traza que describe un error en la transmisión del recurso:

$$T3 := connect\ ?link \Longrightarrow \bigcirc (confirm\ !link \Longrightarrow \bigcirc (\ request\ ?item \Longrightarrow \bigcirc (\ answer\ (!OK, !all_data(item))\ [available(item)] \Longrightarrow \bigcirc\ T31)))$$

$$T31 := nack\ ?nt \Longrightarrow \bigcirc (\ answer\ (!OK, !trama(nt, item)) \Longrightarrow \bigcirc\ T31)$$

Y la traza que describe la pérdida del asentimiento del recurso:

$$T4 := connect\ ?serv \Longrightarrow \bigcirc (confirm\ !serv \Longrightarrow \bigcirc (\ request\ ?elem \Longrightarrow \bigcirc (\ answer\ (!OK, !all_data(elem))\ [available(elem)] \Longrightarrow \bigcirc\ T41)))$$

$$T41 := tout \Longrightarrow \bigcirc (\ query\ !elem \Longrightarrow \bigcirc\ T41)$$

El resultado de solapar estas trazas puede verse en la figura B.1.

B.3. Verificación de propiedades

Una vez obtenida una representación inicial del sistema (la de la figura B.1), es necesario formular las propiedades que creemos que debe cumplir esa descripción y proceder a su verificación.

Vamos a realizar este procedimiento con varias propiedades que deriven en resultados distintos para observar la información que nos aporta el algoritmo de verificación.

B.3.1. P1: Ausencia de bloqueo

La primera propiedad que deseamos verificar es la ausencia de bloqueo. Nuestro sistema nunca debe quedarse bloqueado y debe existir al menos una evolución posible en cada estado. Esto debe ser siempre cierto, independientemente de los eventos que se produzcan y de los valores que se reciban del

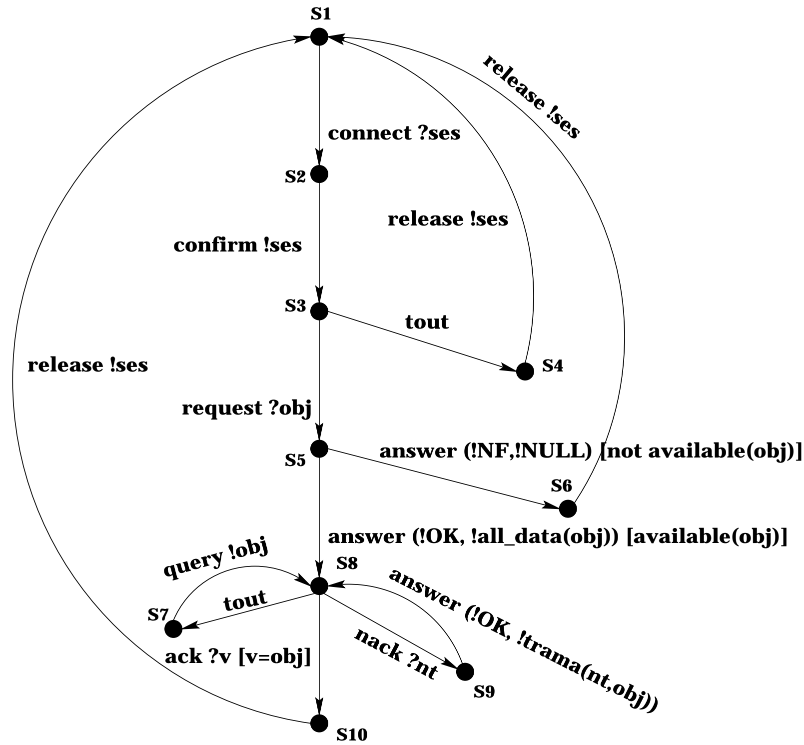

Figura B.1: Resultado del solapamiento

entorno y se asignen a las variables del sistema.

Para comprobar lo mencionado, la propiedad que debemos elegir es una invarianza universal, que especifique que siempre hay algún futuro. Esta propiedad es:

REQ No_deadlock IS

$(\textcircled{e}\,\textit{true}) \wedge \bigcirc\ \textit{No_deadlock}\ \{\}\ ()$

ENDREQ

El resultado del algoritmo de verificación cuando lo aplicamos a esta propiedad es el siguiente par de condiciones:

$CV = true$

$CR = false \bigtriangledown(S1)$ *(REC(S1)* $\bigtriangledown(S3)$ *(*$\exists$ $\{obj\}$*(((*$\neg$ *available(obj))* $\wedge$ *REC(S1))*
$\bigtriangledown(S5)$ *((available(obj))* $\wedge$ *(*$\exists$ $\{v\}$ *((v=obj)* $\wedge$ *REC(S1)))))))*

Tal como se esperaba, la condición de verificación es la constante *true*. Es decir, el requisito se cumple siempre, sin importar los valores recibidos del entorno.

Por contra, la condición de rechazo tiene una forma más compleja. Sin embargo, un análisis de esta condición nos revela que su resultado siempre es el mismo (*false*), sin importar los valores que se asignen a las variables. Este era, lógicamente, el resultado esperado, ya que todavía no se han verificado otras propiedades que pudieran establecer invariantes sobre los eventos de los estados.

Hay que destacar que, tal como se puede ver en la condición de rechazo, los resultados del algoritmo pueden contener constructores propios del formalismo, ajenos a la lógica tradicional. Esto, excepto en casos sencillos, impide el análisis de los resultados a los usuarios que no conozcan la interpretación de los elementos introducidos. Por tanto, cualquier aplicación que implemente este algoritmo debe incluir una herramienta que analice estos resultados en función de los valores de las variables que proporcione el usuario y de las reglas de evaluación definidas para los operadores citados.

B.3.2. P2: Todas las conexiones deben ser liberadas

Ahora queremos comprobar que ninguna conexión queda pendiente de liberación. Si se establece una conexión, cualquier evolución del servidor debe llevar a un estado en donde se libera esa conexión.

La semántica de esta propiedad es la de una finalidad universal y, por tanto, recurrimos al modelo correspondiente. La propiedad especificada es:

REQ Con_i_rel $\{$*connect:TipoSes, release:TipoSes*$\}$ *IS*

connect ?ses $\Longrightarrow \bigcirc$ $\forall\{ses\}$ *(Release* $\{$*release*$\}$*(ses))*

ENDREQ

REQ Release $\{$*free:TipoSes*$\}$*(s:TipoSes) IS*

free !s ∨ ⓐ *Release* {*free*}*(s)*

ENDREQ

El resultado de la verificación de esta propiedad es:

$CV = true$

$CR = false$

El resultado es el esperado en ambos casos, ya que en el árbol simbólico del sistema se puede ver que todas las trazas terminan con la liberación de la conexión en curso.

Para estudiar diversas posibilidades en las siguientes secciones, vamos a hacer que la herramienta establezca los invariantes oportunos para asegurar que se mantiene esta propiedad. En concreto, se van a convertir en fijos los tres eventos "*release* !*ses*" (en los estados S_4, S_6 y S_{10}) en donde se libera la conexión. Estos eventos no deben ser eliminados si deseamos que se mantenga la propiedad.

B.3.3. P3: Todos los recursos están disponibles

Con la siguiente propiedad tratamos de comprobar que, en caso de ser solicitado, cualquier recurso del sistema está disponible y puede ser entregado al cliente.

Para ello especificamos una propiedad con un parámetro: el identificador del recurso cuya disponibilidad queremos verificar.

La propiedad que utilizamos es una invarianza que establece una precedencia entre dos eventos:

REQ Recur_dispon {*request:TipoObj, answer:TipoAns*} *(rec:TipoObj) IS*

(request ?o:TipoObj ⟹ⓔ ∃ {*o*}*(answer (!OK,!all_data(rec))))*

∧ ◯ *Recur_dispon* {*request,answer*}*(rec)*

ENDREQ

En el momento de verificar podemos instanciar la propiedad dándole un valor al parámetro o dejarlo como una variable. Si optamos por esto último, dejar que la variable *rec* identifique al recurso cuya disponibilidad tratamos de verificar, el resultado es:

$CV = \forall\ \{rec\}(\exists\ \{obj\}((all_data(obj)=all_data(rec)) \wedge (available(obj))))$

$CR = false \bigtriangledown(S1)\ (REC(S1) \bigtriangledown(S3)\ (\exists\ \{obj\}(((\neg\ available(obj)) \wedge REC(S1)) \bigtriangledown(S5)\ ((available(obj)) \wedge (\exists\ \{v\}\ ((v=obj) \wedge REC(S1)))))))$

Como vemos, la condición de verificación nos informa de que el recurso puede ser entregado siempre que exista en el servidor y uno de los valores de la variable *obj* (los que solicita el cliente) coincida con él. En definitiva, el requisito se cumple ya que el servidor puede entregar cualquier recurso con tal de que se lo pidan.

La condición de rechazo es la misma que la de la primera propiedad verificada. Por tanto, siempre resultará en un $false$ y el requisito nunca es rechazado.

B.3.4. P4: Las solicitudes de recursos pueden duplicarse

Veamos ahora una propiedad que el sistema no cumple.

Por ejemplo, si el cliente no recibe la respuesta a la solicitud del recurso, vamos a suponer que la repite. En ese caso, podemos tratar de comprobar que el servidor está preparado para contemplar la posibilidad de que su respuesta se pierda y se produzca una repetición de la solicitud.

La propiedad que podríamos especificar para comprobar ese comportamiento es una invarianza que define una precedencia entre dos eventos:

REQ Anw_i_req {answer:TipoAns, request:TipoObj} IS
(answer (!OK,!all_data(o)) ⟹ⓔ request ?o2:TipoObj {o2=o})
∧ @ Anw_i_req {answer,request}()
ENDREQ

El resultado de la verificación de esta propiedad es:

$CV = \forall\ \{obj\}(\neg(available(obj)))$

$CR = false \bigtriangledown(S1)\ (REC(S1) \bigtriangledown(S3)\ (\exists\ \{obj\}(((\neg\ available(obj)) \wedge REC(S1))$
$\bigtriangledown(S5)\ ((available(obj)) \wedge (\exists\ \{v\}\ ((v{=}obj) \wedge REC(S1)))))))$

La condición de rechazo es la misma que la de la anterior propiedad (el sistema no rechaza la propiedad).

Por su parte, la condición de verificación nos plantea una situación imposible: el requisito sólo se cumple si todos los valores asignados a la variable *obj* cumplen $\neg available(obj)$. Es decir, que los recursos no estén disponibles. Si la variable *obj* puede tomar un valor que implique que el recurso esté disponible, la propiedad no se cumple.

Como vemos responde a lo que esperábamos desde un punto de vista intuitivo: la propiedad sólo se cumple si nunca se envía un recurso. Si se solicita un recurso disponible, la propiedad no se cumple.

B.3.5. P5: Hay que esperar el asentimiento

Con esta propiedad queremos comprobar que, después de enviar el recurso solicitado, el sistema no puede liberar la conexión. Esta propiedad formaría parte del proceso de verificación de que el sistema siempre espera a la recepción del asentimiento antes de liberar la conexión.

Para comprobar ese comportamiento podríamos especificar una invarianza donde, en todos los estados, se prohiba la liberación de la conexión tras enviar el recurso al cliente.

REQ Anw_i_nolib $\{$*answer:TipoAns, release:TipoSes*$\}$ *IS*
(answer (!r,!d) $\Longrightarrow\bigcirc\ \neg$ *release !s)*
$\wedge\ \bigcirc$ *Anw_i_nolib* $\{$*answer,release*$\}$*()*
ENDREQ

El resultado de la verificación de esta propiedad es:

$CV = \forall\ \{obj\}(available(obj))$

CR = false $\bigtriangledown$*(S1) (REC(S1)* $\bigtriangledown$*(S3) (*$\exists$ *{obj} ((*$\neg$*(available(obj)))* $\bigtriangledown$*(S5) (((*$\neg$*(available(obj)))* $\wedge$ *REC(S1))* $\bigtriangledown$*(S5) (available(obj)* $\wedge$ *(*$\exists$ *{v}((v=obj)* $\wedge$ *REC(S1))))))))*

Un análisis de los resultados en función de los valores posibles de las variables libres nos indica que la propiedad sólo se cumple si los valores posibles de la variable *obj* pertenecen al conjunto de los valores de los recursos disponibles.

Si *obj* puede tomar el valor de un recurso no disponible, la propiedad no sólo no se cumple, sino que incluso es rechazada por el sistema, puesto que la condición CR es $True$. En este último caso, la propiedad entra en contradicción con otra verificada previamente.

En concreto, el problema se deriva de la segunda propiedad y los invariantes que impuso: no se pueden eliminar los eventos *"release"*.

Y, sin embargo, eso es precisamente lo que necesita la propiedad para verificarse. Podemos observar que, en el caso de que el recurso solicitado no exista, después de la contestación viene inmediatamente la liberación de la conexión, ya que no es necesario esperar el asentimiento.

Para que la propiedad se cumpliese habría que eliminar ese evento y eso no lo permite la segunda propiedad.

En todo caso, el resultado del proceso de verificación nos muestra que el requisito se cumple para los valores de los recursos disponibles. En definitiva, esos son los recursos por los que hay que esperar asentimiento y, por tanto, podemos renunciar al cumplimiento de la propiedad tal cual está especificada sin temor a que el sistema sea incorrecto.

B.4. Obtención de sugerencias

Veamos ahora un conjunto de propiedades que no se cumplen. Para cada una de ellas solicitaremos sugerencias a LIRA y veremos el resultado junto con sus consecuencias.

B.4.1. S1: Las solicitudes de recursos pueden duplicarse

Esta era la propiedad **P4** que verificamos en el capítulo anterior y que, en condiciones normales, no se cumplía. Su forma era:

REQ Anw_i_req {answer:TipoAns, request:TipoObj} IS

(answer (!OK,!all_data(o)) ⟹ⓔ request ?o2:TipoObj {o2=o})

∧ @ Anw_i_req {answer,request}()

ENDREQ

La condición de verificación resultante nos informaba de que la única forma de que se cumpliese la propiedad era que nunca se entregasen recursos:

$CV = \forall \ \{obj\}(\neg(available(obj)))$

Para ejecutar el algoritmo de cálculo de sugerencias es necesario proporcionarle el CVS de cada variable (los valores para los que queremos que se verifique la propiedad). Por ejemplo, para la variable *obj* le indicaríamos el conjunto de valores de los recursos disponibles (la función $available(obj)$ daría un resultado $true$).

El resultado de la verificación sería $False$ y entonces podríamos solicitar sugerencias. En ese caso, el resultado que nos devuelve el algoritmo es:

Sugerencia = Add_Ev(S8, request ?o2:TipoObj [o2=o])

Es decir, la sugerencia de la herramienta consiste en añadir, en el estado *S8*, el evento *"request ?o2:TipoObj [o2=o]"* (tendríamos que cambiar la variable "*o*" de la guarda por "*obj*" que es la que realmente se adapta a la propiedad).

Evidentemente, tendríamos que añadir también la continuación del comportamiento del sistema a partir de ese nuevo evento. Por ejemplo, enviar de nuevo el recurso solicitado y volver al estado *S8*. El resultado puede verse en la figura B.2.

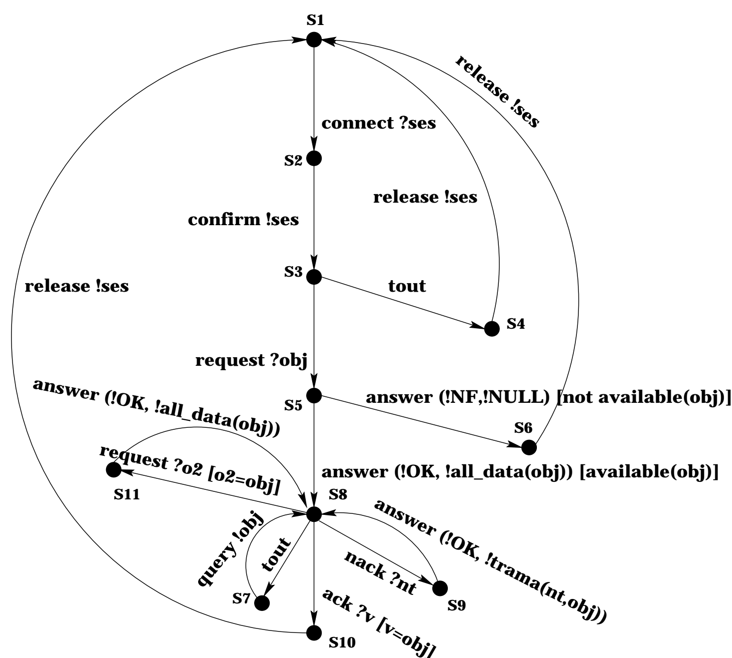

Figura B.2: Primera modificación

B.4.2. S2: Los asentimientos no pueden perderse

Si damos por válido que los asentimientos de los clientes no pueden perderse, entonces no hay necesidad de contemplar el tratamiento de un vencimiento de la temporización después de transmitir una respuesta.

Para estar seguros de los cambios que es necesario realizar, podemos tratar de verificar una propiedad en donde se prohiban los vencimientos de temporizaciones tras enviar un recurso. Por ejemplo:

REQ Anw_i_ntout $\{$*answer:TipoAns, tout:None*$\}$ *IS*

(answer (!OK,!i) $\Longrightarrow \bigcirc \neg$ *tout)* $\wedge$ @ *Anw_i_req* $\{$*answer,tout*$\}$*()*

ENDREQ

La condición de verificación resultante sería:

$CV = \forall\ \{obj\}(\neg(available(obj)))$

Es decir, el resultado es la misma condición que el de la propiedad anterior. Su significado es el mismo: la única manera de que se cumpla el requisito es que nunca lleguemos a transmitir una respuesta. Por tanto, la propiedad no se cumple.

Si especificamos el mismo CVS para la variable "*obj*" que en la propiedad anterior y solicitamos sugerencias, la respuesta que nos proporciona el algoritmo es:

Sugerencia = Del_Ev(S8, tout)

Es decir, la herramienta nos aconsejaría eliminar el evento *"tout"* del estado *S8*.

B.4.3. S3: Las solicitudes de conexión pueden duplicarse

Con esta propiedad queremos especificar el hecho de que es posible que se pierda la confirmación de la conexión y que el cliente volverá a realizar la solicitud. La propiedad es:

REQ Cof_i_cor {confirm:TipoSes, connect:TipoSes} IS
(confirm !s $\Longrightarrow$ⓔ *connect ?s2:TipoSes [s2=s])*
$\wedge$ @ *Cof_i_cor {confirm,connect}()*
ENDREQ

La condición de de verificación resulta:

$CV = false$

Como no se cumple, solicitamos sugerencias y la herramienta nos contesta con:

Sugerencia = Add_Ev(S3, connect ?s2:TipoSes [s2=s])

Es decir, la sugerencia que nos proporciona la herramienta consiste en añadir el evento *"connect ?s2:int [s2=s]"* en el estado *S3*. Evidentemente, tendríamos que añadir también la continuación del comportamiento del sistema a partir de ese nuevo evento. Por ejemplo, responder con una confirmación y volver al estado *S3*.

El resultado sería el de la figura B.3.

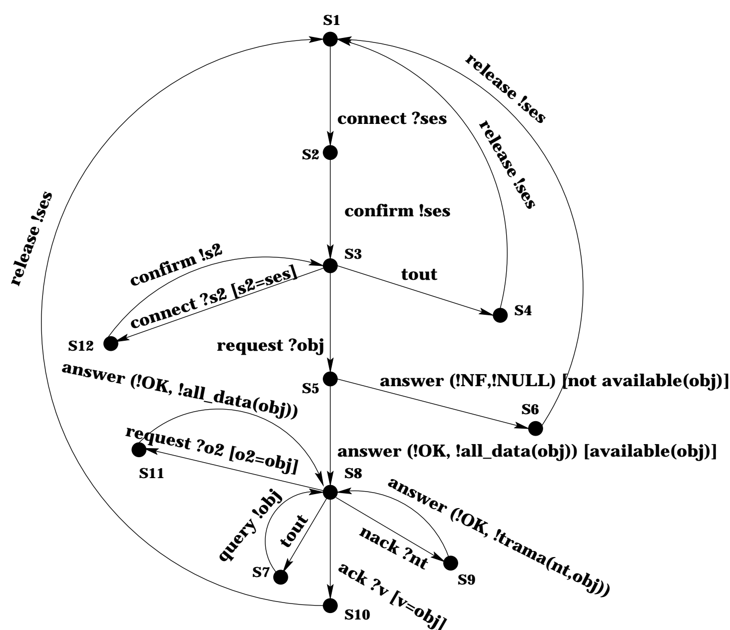

Figura B.3: Segunda modificación

B.4.4. S4: Puede aparecer otra solicitud de conexión

Con esta propiedad queremos especificar el hecho de que nuestro sistema debe estar protegido ante el suceso de que aparezca una repetición de la solicitud de conexión como la sección anterior, pero en la que no coincide el identificador de la sesión.

La propiedad sería:

REQ Cof_i_ocor {confirm:TipoSes, connect:TipoSes} IS
(confirm !s ⟹ⓔ connect ?s3:TipoSes [s3≠ s])
∧ @ Cof_i_ocor {confirm,connect}()
ENDREQ

La condición de de verificación resulta:

$CV = false$

Es decir, como antes, no se cumple. Solicitamos sugerencias y la herramienta nos contesta con:

Sugerencia1 = Ace_Ev(S3, connect ?s2:TipoSes [s2=s],
connect ?s3:TipoSes [s3≠s])

Sugerencia2 = Add_Ev(S3, connect ?s3:TipoSes [s3≠s])

Es decir, la herramienta nos proporcionaría dos sugerencias:

1. Modificar el evento añadido en la sección anterior para acercarlo al que requiere este requisito. Esto no es posible, puesto que dejaría de cumplir la propiedad por la cual se añadió ese evento.
2. Añadir el evento *"connect ?s3:TipoSes [s3≠s]"* en el estado *S3*.

La segunda sugerencia es la que vamos a seguir. Como en la sección anterior, hay que añadir también la continuación del comportamiento del sistema a partir del nuevo evento. Por ejemplo, generar un error y reiniciar el protocolo.

El resultado sería el de la figura B.4.

En principio podríamos pensar que con esta modificación conseguimos cumplir la propiedad sin problemas. Sin embargo, al dar de alta la modificación en el desarrollo del sistema, la herramienta procede a un chequeo de

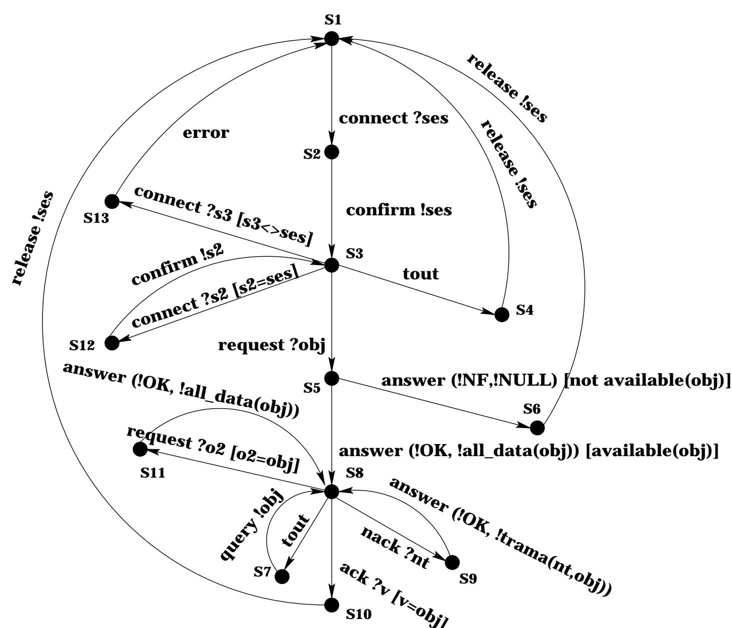

Figura B.4: Tercera modificación

integridad de las propiedades anteriores y descubre que ha dejado de verificarse la propiedad **P2**.

Vemos que es cierto. Ahora hay un camino a través del cual no se produce la liberación de la conexión.

Para que no se produjese este comportamiento anómalo, bastaría con que el evento *"error"* condujese al estado $S3$ en lugar de al $S1$.

B.4.5. S5: Avisar cuando venza una temporización

Veamos ahora una propiedad donde se verifique que el sistema avisa cada vez que vence una temporización. Para ello, supondremos que la acción de avisar se representa mediante el evento *"alarm"*.

La propiedad que podríamos verificar es:

REQ Avisar_tout {tout:None, alarm:None} IS

(tout $\Longrightarrow$*@ alarm)* $\wedge \bigcirc$ *Avisar_tout* {*tout,alarm*}*()*

ENDREQ

La condición de verificación es:

CV = false

Es decir, como antes, no se cumple. Solicitamos sugerencias y la herramienta nos contesta con:

Sugerencia = AndS(Sugerencia1, Sugerencia2)

Sugerencia1 = Add_Ev(S4, alarm)

Sugerencia2 = Add_Ev(S7, alarm)

Es decir, LIRA nos aconseja añadir el evento *"alarm"* en los estados $S4$ y $S7$. Lo lógico será añadirlos entre el evento *"tout"* y el que le siga.

B.4.6. S6: Los asentimientos negativos se registran

Veamos ahora una propiedad donde se verifique que el sistema registra todos los asentimientos negativos. Representaremos la acción de registrar por el evento *"log"*. Además, comprobaremos que la trama que llegó mal tiene un número mayor o igual que 1.

La propiedad que podríamos verificar es:

REQ Registrar {*nack:int, log:None*} *IS*

(nack ?nt:int $\Longrightarrow$*@ (True [nt*$\geq$*1]* $\wedge$ *log))*

$\wedge \bigcirc$ *Registrar* {*nack,log*}*()*

ENDREQ

La condición de verificación resulta:

$CV = \forall\ \{obj\}(\neg(available(obj)))$

Es decir, no se cumple. Solicitamos sugerencias y el resultado es:

Sugerencia = AndS(Sugerencia1, Sugerencia2)

Sugerencia1 = Add_Ev(S9, log)

Sugerencia2 = SS(NoS(S9, True [nt≥1]))

Es decir, LIRA nos aconseja añadir el evento *"log"* en el estado $S9$. Además, nos dice que en ese mismo estado no se cumple el requisito *"True [nt≥1]"*, aunque no puede ofrecernos ninguna sugerencia para arreglarlo.

B.5. Traducción a ELOTOS

Una vez que tenemos la arquitectura inicial del sistema, la traducimos al lenguaje de especificación formal ELOTOS.

La traducción que realiza LIRA carece de estructura, pero refleja el comportamiento del sistema haciendo uso de los constructores básicos de ELOTOS.

La especificación que vamos a generar es la correspondiente a la figura B.4 corregida, es decir, con el evento *"error"* conduciendo al estado $S3$.

Por simplicidad omitimos la instanciación de eventos y parámetros en las llamadas a los procesos.

El resultado se puede observar en la figura B.5

```
module mod_SR is

   process proc_S1 is
      connect ?ses:TipoSes; confirm !ses; proc_S3 [] ()
   endproc

   process proc_S3 is
        (request ?obj:TipoObj;
          ((answer (!OK,!all_data(obj)) [available(obj)]; proc_S8 [] ())
           []
           (answer (!NF,!NULL) [not(available(obj))];
                release !ses; proc_S1 [] ())))
        []
        (tout; release !ses; proc_S1 [] ())
        []
        (connect ?s2:TipoSes;
           ((confirm !s2 [s2=ses]; proc_S3 [] ())
            []
            (error [s2<>ses]; proc_S3 [] ())))
   endproc

   process proc_S8 is
      (ack ?v:TipoObj [v=obj]; release !ses; proc_S1 [] ())
      []
      (nack ?nt:int; answer (!OK,!trama(nt,obj)); proc_S8 [] ())
      []
      (tout; query !obj; proc_S8 [] ())
      []
      request ?o2:TipoObj [o2=obj]; answer (!OK,!all_data(obj));
          proc_S8 [] ()))))
   endproc
endmod

specification spec_SR is
   gates connect:int, confirm:int, tout:none, release:int, request:int,
         answer:int, nack:int, ack:int, query:int, error:None
   behaviour proc_1 [] ()
endspec
```

Figura B.5: Especificación ELOTOS del sistema

Bibliografía

[Abr99] J.R. Abrial. *The B Book*. Cambridge University Press, 1999.

[ABT95] P. Kearney A. Bloesch, E. Kazmierczak and O. Traynor. Cogito: A Methodology and System for Formal Software Development. *International Journal of Software Engineering and Knowledge Engineering*, 5(4):599–617, Diciembre 1995.

[AS85] B. Alpern and F. Schneider. Defining liveness. *Information Processing Letters*, 1985.

[BB89] T. Bolognesi and E. Brinksma. Introduction to the ISO specification language LOTOS. In P.H.J. van Eijk, C.A. Vissers, and M. Diaz, editors, *The Formal Description Technique LOTOS*. North-Holland, 1989.

[BBRTL90] D. Blyth, C. Boldyreff, C. Ruggles, and N. Tetteh-Lartey. The case for Formal Methods in standards. *IEEE Software*, Septiembre 1990.

[BCL+94] J.R. Burch, E.M. Clarke, D.E. Long, K.L. McMillan, and D.L. Dill. Symbolic model checking for sequential circuit verification. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 13(4):401–424, Abril 1994.

[Bea96] N. Bjorner et al. STeP: Deductive-algorithmic verification of reactive and real-time systems. In *Proc. of the 8th International Conference on Computer-Aided Verification*, volume 1102 of *Lectures Notes in Computer Science*, pages 415–418. Springer-Verlag, Julio 1996.

[BFM89] R. Bloomfield, P. Froome, and B. Monahan. SpecBox: a toolkit for BSI-VDM. *Safety Net*, 5, 1989.

[BGL94] A. Bouali, S. Gnesi, and S. Larosa. The Integration Project for the JACK environment. *Bulletin of the EACTS*, Octubre 1994.

[BH95a] J.P. Bowen and M.G. Hinchey. Seven more myths of formal methods. *IEEE Software*, 12(4):34–41, Julio 1995.

[BH95b] J.P. Bowen and M.G. Hinchey. Ten commandments of formal methods. *Computer*, 28(4):56–63, Julio 1995.

[BKM96] B. Brock, M. Kaufmann, and J.S. Moore. Heavy inference: Theorems about comercial microprocessors. In *Formal Methods in Computer-Aided Design, FMCAD'96*, 1996.

[BM79] R.S. Boyer and J.S. Moore. *A computational logic*. Academic Press, New York, 1979.

[Boe88] B. W. Boehm. A spiral model of software development and enhancement. *IEEE Computer*, 21(5):61–72, 1988.

[BR91] J Bicarregui and B. Ritchie. Reasoning about VDM development using the VDM Support Tool in Mural. In S. Prehn and W.J. Toetenel, editors, *VDM'91 - Formal Software Development Methods*, pages 371–388. Springer-Verlag, Octubre 1991.

[Bro96] M. Broy. Formal Description Techniques - How formal and descriptive are they? In Reinhard Gotzhein and Jan Bredereke, editors, *Formal Description Tecniques IX. Theort, application and tools*, pages 95–110. International Federation for Information Processing (IFIP), Chapman & Hill, 1996.

[BRRdS96] A. Bouali, A. Ressouche, V. Roy, and R. de Simone. The FCTOOLS user manual. Technical Report 191, INRIA, Abril 1996.

[Bry86] R. E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, 35(8), 1986.

[BS93] J. Bowen and V. Stavridou. Safety-critical systems, formal methods and standards. *Software Engineering Journal*, Julio 1993.

[BSS86] E. Brinksma, G. Scollo, and C. Steenbergen. LOTOS specifications, their implementations and their tests. In *6th International Workshop on Protocol Specification, Testing and Verification*, Montreal, Junio 1986.

[Bud92] Stanislaw Budkowski. Estelle Development Toolset (EDT). *Computer Networks and ISDN Systems*, 25(1):63–82, Agosto 1992.

[BvLV94] T. Bolognesi, J. v.d. Lagemaat, and C. A. Vissers, editors. *LOTOSphere: Software development with LOTOS*. Kluwer Academic Publishers, 1994.

[BY96] R. Boyer and Y. Yu. Authomated proofs of object code for a widely used microprocessor. *Journal of the ACM*, 43:166–192, Enero 1996.

[CCF+85] C. Cornes, J. Courant, J.C. Filliâtre, G. Huet, P. Manoury, C. Paulin-Mohring, C. Munoz, C. Murthy, C. Parent, A. Saïbi, and B. Werner. *The coq proof assistant reference manual*. INRIA, http://pauillac.inria.fr/coq/systeme_coq-eng.html, version 5.10 edition, 1985.

[CE81] E.M. Clarke and E.A. Emerson. Synthesis of syncronization skeletons for branching time temporal logic. In *Logic of Programs: Workshop, Yorktown Heights*, volume 131 of *Lectures Notes of Computer Science*. Springer-Verlag, 1981.

[Cea86] R. Constable et al. *Implementing Mathematics with the NuPRL Proof Development Environment*. Prentice-Hall, 1986.

[CES86] E.M. Clarke, E.A. Emerson, and A.P. Sistla. Authomatic Verification of Finite-State Concurrent Systems using Temporal Logic Specifications. *ACM Trans. on Progr. Languages and Systems*, pages 244–263, Abril 1986.

[CGH+93] E.M. Clarke, O. Grumberg, H. Hiraishi, S. Jha, D.E. Long, K.L. McMillan, and L.A. Ness. Verification of the Futurebus+ cache coherence protocol. In *CHDL*, 1993.

[CGL92] E.M. Clarke, O. Grumberg, and D.E. Long. Model checking and abstraction. In *Proc. of Principles of Programming Languages*, 1992.

[CGM+96] G. Chehaibar, H. Garavel, L. Mounier, N. Tawbi, and F. Zulian. Specification and verification of the Powerscale bus arbitration protocol: An industrial experiment con LOTOS. In Chapman & Hall, editor, *FORTE/PSTV 96*, Kaiserlautern, Germany, 1996.

[CKM+88] D. Craigen, S. Kromodimoeljo, I. Meisels, A. Neilson, B. Pase, and M. Saaltnik. m-EVES: A tool for veryfing software. In *Proc. of 10th International Conference on Software Engineering*, pages 324–333, Singapore, Abril 1988.

[CMP92] E. Chang, Z. Manna, and A. Pnueli. *Characterization of Temporal Property Classes*. Number 623 in Lecture Notes in Computer Science. Springer-Verlag, 1992. Berlín.

[CMP93] C. Clack, C. Myers, and E. Poon. *Programming with Standard ML*. Prentice-Hall, Londres, 1993.

[CMP95] C. Clack, C. Myers, and E. Poon. *Programming with Miranda*. Prentice-Hall, Londres, 1995.

[CPS93] R. Cleaveland, J. Parrow, and B. Steffen. The concurrency workbench: A semantics-based tool for the verification of concurrent systems. *ACM TOPLAS*, 15(1):36–72, Enero 1993.

[CW96] Edmund M. Clarke and Jeannette M. Wing. Formal Methods: State of the Art and Future Directions. Technical Report CMU-CS-96-178, Carnegie Mellon University, 1996.

[CZ93] E.M. Clarke and X. Zhao. Analytica: A theorem prover for Mathematica. *The Mathematica Journal*, pages 56–71, 1993.

[DDHY92] D.L. Dill, A.J. Drexler, A.J. Hu, and C.H. Yang. Protocol verification as a hardware design aid. In *IEEE Int'l Conference on Computer Design: VLSI in Computers and Processors*, pages 522–525, 1992.

[Dil96] D. Dill. The Murϕ verification system. In R. Alur and T.A. Henzinger, editors, *Computer Aided Verification*, volume 1102 of *Lecture Notes in Computer Science*. Springer-Verlag, 1996.

[DW89] F. Dederichs and R. Weber. *Safety and Liveness from a methodological point of view.* Facultad de Informática, 1989. Universidad de Passau.

[ECB94] W. Elseaidy, R. Cleaveland, and J. Baugh. Modeling and verifying active structural control systems. In *Real-Time Systems Symposium*, 1994.

[ECB96] W. Elseaidy, R. Cleaveland, and J. Baugh. Modelling and verifying active structural control systems. *Science of Computer Programming*, 1996.

[EGHT94] D. Evans, J. Guttag, J. Horning, and Y. Tang. LCLint: a tool for using specifications to check code. In *Symposium on the Foundations of Software Engineering*, Diciembre 1994.

[EH83] E. Emerson and J. Halpern. 'Sometimesánd 'Not never'revisited. In *10th ACM Simposium on principles of programming languages*, 1983.

[EL86] E.A. Emerson and C. Lei. Efficient model checking in fragments of the propositional mu-calculus. In *Proc. Symp. on Logic in Computer Science*, Cambridge, MA, 1986.

[ELL94] R. Elmstrom, P.G. Larsen, and P.B. Larsen. The IFAD VDM-SL Ttoolbox: a practical approach to formal specification. In *ACM Sigplan Notices*, 1994.

[ES88] E. Emerson and J. Srinivasan. *Branching time temporal logic.* Number 354 in Lecture Notes in Computer Science. Springer-Verlag, 1988.

[ES89] E.A. Emerson and J. Srinivasan. *Branching time temporal logic*. Number 354 in Lecture Notes in Computer Science. Springer-Verlag, Berlín, 1989.

[FEH83] W. Fey, H. Ehrig, and H. Hansen. Act-One: An algebraic language with two levels of semantics. Technical Report 83.103, Tech. Universitat Berlin, 1983.

[FGK+96] J.C. Fernandez, H. Garavel, A. Kerbrat, R. Mateescu, L. Mounier, and M. Sighireanu. CADP (CAESAR/ALDEBARAN development package): A protocol validation and verification toolbox. In *Proc. of the 8th International Conference on Computer Aided Verification*, number 1102 in Lecture Notes in Computer Science. Springer-Verlag, Julio 1996.

[FGM+94] J.C. Fernández, H. Garabel, L. Mounier, A. Rasse, C. Rodríguez, and J. Sifakis. A Toolbox for the Verification of LOTOS Programs. In L.A. Clarke, editor, *14th International Conference on Software Engineering*, Melbourne, Australia, Mayo 1994.

[FL79] M.J. Fischer and R.E. Ladner. Propositional dynamic logic of regular programs. *Journal Computer Systems Science*, 18(2):194–211, 1979.

[Flo67] R. Floyd. Assigning meanings to programs. In AMS, editor, *Proc. Symposium in Applied Mathmatics*, volume 19, 1967.

[Flo95] J. Floch. Supporting evolution and maintenance by using a flexible code generator. In *17th Int'l Conf. on Software Engineering*, Seattle, Abril 1995.

[GCR94] S. Gerhart, D. Craigen, and T. Ralston. Experience with formal methods in critical systems. *IEEE Software*, Enero 1994.

[GG88] S.J. Garland and J.V. Guttag. Inductive methods for reasoning about abstract data types. In *Proc. of the 15th Symposium on Principles of Programming Languages*, pages 219–228, 1988.

[GG91] S. Garland and J. Guttag. *A guide to LP: the Larch Prover*. MIT LCS, Diciembre 1991.

[GH93] J. Guttag and J. Horning. *Larch: Languages and Tools for Formal Specification*. Springer-Verlag, 1993.

[GL93] B. Ghribi and L. Logrippo. A validation environment for LOTOS. In *13th Int'l Conf. on Protocol Specification, Testing and Verification*. North-Holland, 1993.

[GM93] M.J.C. Gordon and T.F. Melham. *Introduction to HOL: A theorem proving environment for Higher Order Logic*. Cambridge University Press, 1993.

[GMW79] M.J. Gordon, A.J. Milner, and C.P. Wadsworth. *Edinburgh LCF*, volume 78 of *Lectures Notes in Computer Science*. Springer-Verlag, 1979.

[Got92] R. Gotzhein. Temporal logic and its applications. *Computer Networks and ISDN Systems*, 24:203–218, 1992.

[GT79] J. A. Goguen and J.J. Tardo. An introduction to OBJ: A language for writing and testing algebraic program specifications. In *Proc. of Conference on Specification of Reliable Systems*, Cambridge, MA, USA, 1979.

[Hal90] J.A. Hall. Seven myths of formal methods. *IEEE Software*, 7(5):11–19, Septiembre 1990.

[Hal96] A. Hall. Using formal methods to develop an ATC information system. *IEEE Software*, 12, Marzo 1996.

[Hau96] O. Haugen. SDL and MSC. *Computer Networks and ISDN Systems*, Junio 1996.

[HDMC96] J. Hintelmann, M. Difenbruch, and B. Muller-Clostermann. The QUEST Approach for the Performance Evaluation of SDL systems. In *FORTE/PSTV 96*, Kaiserlautern, Germany, Octubre 1996.

[HK90] Z. Harel and R.P. Kurshan. Software for analytica development of communications protocols. *AT&T Bell Laboratories Journal*, 69(1):45–59, Enero-Febrero 1990.

[HK91] I. Houston and S. King. CICS project report: Experiences and results from using Z. In *Proc. of VDM'91: Formal Development Methods*, volume 551 of *Lecture Notes in Computer Science*. Springer-Verlag, 1991.

[HL95a] M. Hennessy and H. Lin. Symbolic bisimulations. *Theoretical Computer Science*, 138:353–389, 1995.

[HL95b] M. Hennessy and X. Liu. A modal logic for message passing processes. *Acta Informática*, 32:375–393, 1995.

[HM80] M. Hennessy and R. Milner. *On observing nondeterminism and concurrency*. Number 85 in Lecture Notes in Computer Science. Springer-Verlag, Berlín, 1980.

[HM85] M. Hennessy and R. Milner. Algebraic laws for nondeterminism and concurrency. *J. Ass. of Computer Mach.*, 1985.

[Hoa69] C.A.R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, Octubre 1969.

[Hoa85] C.A.R. Hoare. *Communicating Sequential Processes*. Prentice-Hall International, 1985.

[Hol91] G. Holzmann. *Design and validation of computer protocols*. Prentice-Hall, 1991.

[Hol94] G.J. Holzmann. The theory and practice of a formal method: NewCore. In *IFIP 94*, Hamburgo, 1994.

[HP95] U. Harner and J. Peleska. The Airbus A 330/340 Cabin Communication Systema. In M. Hinchey and J. Bowen, editors, *Applications of Formal Methods*. Prentice-Hall International, 1995.

[HR97] M. Hennessy and J. Rathke. Local model checking for a value-based modal μ-calculus. In *International Simposium on Theoretical Aspects of Computer Software, TACS'97*, Sendai, 1997.

[ISO89a] ISO. *Information Processing Systems - Open Systems Interconnection - ESTELLE - A Formal Description Tecnhique based on an Extended State Transition Model.* ISO/IEC/9074, Geneva, 1989.

[ISO89b] ISO. *Information Processing Systems - Open Systems Interconnection - LOTOS - A Formal Description Tecnhique based on the Temporal Ordering of Observational Behaviour.* ISO/IEC/8807, Geneva, 1989.

[ISO93] ISO. *Information Technology Programming Languages -VDM-SL.* ISO/IEC/JTC1/SC22/WG19, n-20 edition, Noviembre 1993.

[ISO95] ISO. *Z Notation.* ISO Panel JTC1/SC22/WG19, version 1.2 edition, Septiembre 1995.

[ISO98] ISO. *Final Commite Draft on Enhancements to LOTOS.* ISO/IEC JTC1/SC21/WG7, Mayo 1998.

[ITU93] ITU. *SDL: Specification and Description Language.* CCITT, 1993.

[JB78] C.B. Jones and D. Bjørner. *The Vienna Development Method: The Metalanguage*, volume 61 of *Lecture Notes in Computer Science.* Springer Verlag, 1978.

[JB82] C.B. Jones and D. Bjørner. *Formal Specification and Software Development.* Prentice-Hall International, 1982.

[JL95] A. Jirachiefpattana and R. Lai. Verification results for the ISO ROSE protocol specified in ESTELLE. In S.T. Vuong and S.T. Chanson, editors, *14th IFIP/PSTV*, 1995.

[JMT90] D. T. Jordan, J.A. McDermid, and I. Tyn. CADiZ - Computer Aided Design in Z. In J. E. Nicholls, editor, *Workshops in Computing: Z User Workshop*, pages 93–104. Springer-Verlag, 1990.

[Jon90] C.B. Jones. *Systematic Software Development Using VDM.* International Series in Computer Science. Prentice-Hall International, New York, second edition, 1990.

[KAW96] D.J. King, R.D. Arthan, and I.C.L. Winnersh. Development of practical verification tools. *ICL Systems Journal*, 11, Mayo 1996.

[KL93] R. Kurshan and L. Lamport. Verification of a multiplier: 64 bits and beyond. In C. CouCourbetis, editor, *Computer Aided Verification*, volume 697 of *Lecture Notes in Computer Science*. Springer-Verlag, 1993.

[Klo87] J.W. Klop. Term Rewriting Systems, a tutorial. *Bulletin of the EACTS, 32*, Marzo 1987.

[Klo92] J.W. Klop. Term Rewriting Systems. In *HandBook of Logic in Computer Science*, volume 2. Oxford Science Publications, 1992.

[KM87] D. Kapur and D. Musser. Proof by consistency. *Artificial Intelligence*, 31:125–157, 1987.

[KM95] M. Kaufmann and J.S. Moore. *ACL2: A Computational Logic for Applicative Common Lisp. The user's manual*. http://www.utexas.edu/users/moore/acl2/v2-3/acl2-doc.html#User's-Manual, 1995.

[Koz83] D. Kozen. Results on the propositional mu-calculus. *Theoretical Computer Science*, 27:333–354, Diciembre 1983.

[KS97] Gerald Kotonya and Ian Sommerville. *Requirements Engineering. Processes and Techniques*. John Wiley & Sons, 1997.

[KTW96] Kolyang, T.Santen, and B. Wolff. *A Structure Preserving Encoding of Z in Isabelle/HOL*, volume 1125 of *Lecture Notes in Computer Science*. Springer-Verlag, 1996.

[Kur94a] R.P. Kurshan. The complexity of verification. In *Proc. 26th ACM Symposium on Theory of Computing (STOC)*, pages 365–371, Montreal, 1994.

[Kur94b] R.P. Kurshan. *Computer-Aided Verification of Coordinating Processes*. Princenton University Press, 1994.

[Lam80] L. Lamport. 'Sometimesís sometimes 'Not never'. In *7th ACM Simposium on principles of programming languages*, 1980.

[Lam83] L. Lamport. What good is temporal logic. In *Proc. IFIP Congress*. North-Holland, 1983.

[Lam89] L. Lamport. A simple approach to specifying concurrent systems. In *CACM*, volume 32, Enero 1989.

[Lar90] K. Larsen. Proof systems for satisfiability in Hennesy-Milner logic with recursion. *Theoretical Computer Science*, 72, 1990.

[Leb97] P. Leblanc. Modelling and simulation with SDL and MSC by and example. In *SDL Forum 97*, France, Septiembre 1997.

[Les83] P. Lescanne. Computer experiments with the REVE term rewriting system generator. In *Proc. of the 10th Symposium on Principles of Programming Languages*, pages 99–108, Austin-Texas, Enero 1983.

[LH95] K. C. Lano and H. P. Haughton. Formal development in B Abstract Machine Notation. *Information and Software Technology*, 37(5–6):303–316, Mayo–Junio 1995.

[LK95] Pericles Loucopoulos and Vassilios Karakostas. *System Requirements Engineering*. McGraw-Hill, 1995.

[Low98] G. Lowe. Casper: A Compiler for the Analysis of Security Protocols. *Journal of Computer Security*, 6:53–84, 1998.

[LP92] Z. Luo and R. Pollack. LEGO proof development system: User's manual. Technical Report ECS-LFCS-92-211, Computer Science Dept., University of Edinburgh, Mayo 1992.

[MC98] M. Mauny and G. Cousineau. *The functional approach to programming*. Cambridge University Press, 1998.

[McM93] K.L. McMillan. *Symbolic model checking: an approach to the state explosion problem*. Kluwer Academic Publishers, 1993.

[MdM88] J. Mañas and T. de Miguel. From LOTOS to C. In *Formal Descriptions Techniques. FORTE 88*, Stirling, Escocia, Septiembre 1988.

[Mil80] R. Milner. *A Calculus of Communicating Systems*, volume 92 of *Lecture Notes in Computer Science*. Springer Verlag, 1980.

[Mil85] G. Milne. CIRCAL and the representation of communication, concurrency and time. *ACM TOPLAS*, pages 270–298, Abril 1985.

[Mil89] R. Milner. *Communication and Concurrency*. Series in Computing Sciencie. Prentice-Hall, 1989.

[MP89] Z. Manna and A. Pnueli. *The anchored version of the temporal framework*. Number 354 in Lecture Notes in Computer Science. Springer-Verlag, 1989.

[MP90] Z. Manna and A. Pnueli. A hierarchy of temporal properties. In *Proc. 9th ACM Symp. Princ. of Distr. Comp.*, pages 377–408, Agosto 1990.

[MP92a] Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems*. Springer-Verlag, New York, 1992.

[MP92b] Z. Manna and A. Pnueli. *Time for concurrence*. Lecture Notes in Computer Science. Springer-Verlag, 1992. INRIA 25th Anniversary.

[MP92c] Z. Manna and A. Pnueli. *Verifying hybrid systems*. Lecture Notes in Computer Science. Springer-Verlag, 1992.

[MP93] Z. Manna and A. Pnueli. Models for reactivity. *Acta Informática*, 30, 1993.

[MS93] P. Mukherjee and V. Stavridou. The formal specification of safety requirements for storing explosives. *Formal Aspects of Computing*, 5:299–336, 1993.

[MS95] S.P. Miller and M. Srivas. Formal verification of the AAMP5 microprocessor. a case study in the industrial use of formal

methods. In *Workshop on Industrial Stregth Formal Specification Techniques*, pages 2–16, Boca Raton, Florida, 1995.

[NH84] R. Nicola and M. Hennessy. Testing equivalences for processes. *Theoretical Computer Science*, Noviembre 1984.

[NR69] P. Naur and B. Randell. *Software Engineering: A Report on a Conference sponsored by the NATO science committee.* NATO, 1969.

[ORS92] S. Owre, J. Rushby, and N. Shankar. PVS: A prototype verification system. In D. Kapur, editor, *11th International Conference on Automated Deduction (CADE)*, volume 607 of *Lectures Notes in Artificial Intelligence*, pages 748–752. Springer-Verlag, Junio 1992.

[PA95] J.J. Pazos-Arias. *Transformación y verificación con LOTOS*. PhD thesis, Departamento de Ingeniería de Sistemas Telemáticos - Universidad Politécnica de Madrid, 1995.

[PAGDGS+98] J.J. Pazos-Arias, J. García-Duque, A. Gil-Solla, R. Díaz-Redondo, and A. Fernández-Vilas. LTCS: Una lógica temporal causal para la especificación y verificación de requisitos funcionales de un sistema distribuido. In *VI Jornadas de Concurrencia*, pages 105–116, 1998.

[Pau84] Lawrence C. Paulson. Verifying the unification algorithm. Technical Report 50, University of Cambridge, Computer Laboratory, 1984.

[Pau94] Lawrence C. Paulson. *Isabelle: A Generic Theorem Prover*, volume 828 of *Lecture Notes in Computer Science.* Springer-Verlag, 1994.

[Pau98] Lawrence C. Paulson. The inductive approach to verifying cryptographic protocols. *Journal of Computer Security*, 6:85–128, 1998.

[Pel96] D. Peled. Combining partial order reductions with on-the-fly model-checking. *Journal of Formal Methods in Systems Design*, 8(1):39–64, 1996.

[PJW+95] J. Peeters, M. Jadoul, M. Wasosky, D. Witaszek, and J.P. Delpiroux. HW-SW Co-Design and Simulation of a Multimedia Application. In *7th European Simulation Symposium*. Society for Computer Simulation, 1995.

[PM87] D. Pountain and D. May. *A tutorial introduction to Occam programming*. Blackwell Scientific Publications, 1987.

[Pnu77] A. Pnueli. The temporal logic of programs. In *Proc. 18 th FOCS*, pages 46–57, Octubre 1977.

[Pnu79] A. Pnueli. The temporal semantics of concurrent programs. In *Proc. International Symposium on Semantics of Concurrent Programs*, Evian, France, Julio 1979.

[Pnu81] A. Pnueli. A temporal logic of concurrent programs. *Theoretical Computer Science*, 13:45–60, 1981.

[Pnu85] A. Pnueli. Linear and branching structures in the semantics and logics of reactive systems. In *12th ICALP*, volume 194 of *Lectures Notes in Computer Science*, 1985.

[QPF89] J. Quemada, S. Pavón, and A. Fernández. State Exploration by Transformation with LOLA. In *Workshop on Automation Verification Methods for Finite State Systems*, Grenoble, Junio 1989.

[QS82] J. Queille and J. Sifakis. Specification y verification of concurrent systems with CAESAR. In *Proc. of fith ISP*, 1982.

[Ros94] A. Roscoe, editor. *Model Checking CSP*. Prentice-Hall, a classical mind: essays in honour of C.A.R. Hoare edition, 1994.

[Ros95] A.W. Roscoe. Modelling and verifying key-exchange protocols using CSP and FDR. In *8th IEEE Computer Security Foundations Workshop*, 1995.

[Ros97] A.W. Roscoe. *The Theory and Practice of Concurrency*. Prentice-Hall, 1997.

[RS90] V. Roy and R. Simone. Auto/autograph. In E. Clarke and R. Kurshan, editors, *Computer-Aided Verification '90*, volume 3 of *DIMACS Series on Discrete Mathematics and Theoretical Computer Science*, pages 235–250, Piscataway NJ, Junio 1990. American Mathematical Society.

[Sha94] N. Shankar. *Mathematics, Machines and Gödel's Proof.* Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, Cambridge, UK, 1994.

[Som95] Ian Sommerville. *Software Engineering.* Addison-Wesley, 1995.

[Spi88] J.M Spivey. *Introducing Z: a Specification Language and its Formal Semantics.* Cambridge University Press, 1988.

[SS97] Ian Sommerville and Pete Sawyer. *Requirements Engineering.* John Wiley & Sons, 1997.

[Sti96] C. Stirling. *Modal and Temporal Logics for Processes.* Number 1043 in Lecture Notes in Computer Science. Springer-Verlag, 1996.

[SW91] C. Stirling and D. Walker. Local model checking in the modal mu-calculus. *Theoretical Computer Science*, 89, 1991.

[Tar55] A. Tarski. A lattice-theoretical fixpoint theorem and its applications. *Pacific J. Math*, 5:285–309, 1955.

[TG98] J. Thees and R. Gotzhein. The eXperimental Estelle Compiler - automatic generation of implementations from formal specifications. In M. Ardis, editor, *2nd Workshop on Formal Methods in Software Practice*, Florida, 1998.

[Tho91] S. Thompson. *Type Theory and Functional Programming.* Addison-Wesley, 1991.

[vE88] P. van Eijk. *Software Tools for the Specification Language LOTOS.* PhD thesis, University of Twente, Enschede, The Netherlands, 1988.

[Vli93] H. Van Vliet. *Software Engineering: principles and methods.* John Wiley & Sons, 1993.

[VSSB89] C. Vissers, G. Scollo, M. Sinderen, and E. Brinksma. On the use of Specification Styles in the Design of Distributed Systems. Technical report, Fac. Informatics - University of Twente, Enschede, 1989.

[Win91] G. Winskel. A note on model checking the modal mu-calculus. *Theoretical Computer Science*, 83, 1991.

[Xia95] J. Xiaoping. An approach to animating Z specifications. In *Proc. Int'l Computer Software and Applications Conf*, Dallas, Texas, 1995.