UNIVERSIDAD DE VIGO

Dpto. de Tecnologías de las Comunicaciones
ETSI de Telecomunicación

TESIS DOCTORAL

ESPECIFICACIÓN, VERIFICACIÓN Y MANTENIMIENTO DE REQUISITOS FUNCIONALES CON TÉCNICAS DE DESCRIPCIÓN FORMAL

Autor: Jorge García Duque

Director: Dr. José J. Pazos Arias

2000

A los míos

Agradecimientos[1]

Durante el camino recorrido para realizar un trabajo largo y costoso siempre se encuentran obstáculos, momentos en los que uno se siente sólo ante el peligro. Sin embargo, al final del trayecto, cuando se echa la vista atrás, es imposible evitar esbozar esa sonrisa de satisfacción. Los problemas pasados, aquellos momentos de soledad, se ven ahora desde otra perspectiva; y aparecen, casi sin querer, un montón de manos tendidas a lo largo de todo el camino. Y uno piensa, cómo pude encontrar algún inconveniente teniendo tanta gente, tanta ayuda a mi lado. Los problemas surgidos siempre se solventaron gracias a vuestros ánimos, vuestros consejos, vuestra compañía; en fin, gracias a vosotros. Afortunadamente, el final del camino te permite ver que siempre estuvisteis ahí, sin hacer ruido, sin exigir nada. Otra vez tengo que sonreír, recordando los momentos pasados.

Por todo ello, quiero agradecer a cada uno de vosotros todas y cada una de las veces que estuvisteis a mi lado. De tantas maneras distintas. Todas me abrieron las puertas. Cada uno abrió aquéllas que podía abrir. Gracias.

Perdonad si durante el camino no supe valorar vuestro apoyo. Hay veces que sólo es posible ver mirando hacia atrás. Perdonad también si no os pongo nombre, pero vosotros me habéis enseñado que el anonimato de la ayuda desinteresada engrandece aún más el apoyo prestado. Por eso, a todos os tiendo mi mano agradecida, esperando que la recibáis con la misma satisfacción y orgullo con que yo recibí la vuestra. De nuevo, Gracias.

[1]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 enmarca dentro de la *Ingeniería del Software*, disciplina que tiene como objetivo proporcionar teorías, métodos y herramientas para el desarrollo de software de calidad.

La aplicación de las metodologías de desarrollo software tradicionales a sistemas software complejos (sistemas distribuidos) supuso un detrimento en la calidad de los productos software desarrollados, y un incremento en los costes de los mismos; debido, en gran medida, a la imposibilidad de probar el sistema hasta que se disponía de una implementación del mismo.

Las técnicas formales, basadas en la utilización de las matemáticas como vehículo para crear especificaciones de sistemas con una sintaxis y semántica formalmente definida, permiten verificar y validar el sistema en todas las fases del proceso de desarrollo, reduciendo el riesgo de propagar errores a lo largo de dicho proceso. Sin embargo, y a pesar de las ventajas potenciales de los métodos formales, es escasa su integración en la industria del software. Este hecho está motivado principalmente por: el desconocimiento (tanto de clientes como de diseñadores) de estas técnicas; la inexistencia de herramientas apropiadas que permitan su aplicación práctica; así como su deficiente integración en el proceso de desarrollo software. Además, la principal ventaja de estas técnicas (posibilidad de verificar y validar en cualquier fase del proceso de desarrollo) se ve notablemente reducida, debido al alto coste computacional que requiere la verificación formal, cuando se aplican a sistemas de medio y gran tamaño. Una alternativa para solucionar este problema es abordar la verificación mediante refinamientos sucesivos del sistema, siendo la verificación de cada refinamiento mucho más simple.

En esta tesis se presenta un modelo de proceso de desarrollo software incremental totalmente formalizado. El trabajo se centra en las primeras etapas del proceso de desarrollo software (especificación, análisis y verificación de requisitos), abordando también la fase de diseño de la arquitectura del sistema y la fase de mantenimiento. El proceso incremental definido permite tratar cada refinamiento del sistema (incremento) de diferente forma, en función de la etapa de desarrollo en que se produzca. Esta idea se basa en dos aspectos fundamentales: por una parte, aprovechar las características complementarias de los distintos tipos de técnicas de descripción formal (orientadas a propiedades para las primeras fases, y constructivas para fases posteriores), combinándolas en función de la etapa de desarrollo; y por otra, adecuar el tratamiento de dichos refinamientos a su naturaleza.

A continuación, se resumen las principales características y aportaciones de la metodología formal desarrollada, clasificándolas según las diferentes etapas del proceso de desarrollo propuesto:

- Especificación y análisis de requisitos: Se define una lógica temporal (técnica formal orientada a propiedades) con una semántica cercana al lenguaje natural. La expresividad de dicha lógica se enriquece con la introducción de un tercer grado de especificación denominado *subespecificado*. Las partes subespecificadas del sistema pueden ser especificadas en refinamientos posteriores del sistema. Además, dada la importancia de establecer con claridad

los requisitos iniciales del sistema, se proporciona una representación gráfica de los mismos a través de un formalismo basado en grafos con arcos en tres estados: posibles, no posibles y subespecificados. Los arcos subespecificados proporcionan grados de libertad y permiten automatizar tomas de decisión en las siguientes fases de desarrollo.

- Verificación: Se proporciona una técnica de verificación basada en *model checking*, utilizando la lógica temporal definida para la especificación de requisitos y el grafo de representación para realizar la verificación propiamente dicha. Para aumentar la expresividad de las técnicas utilizadas, proporcionando la mayor información posible al usuario, se define una relación de satisfacción con seis grados de verdad; lo que permite expresar el grado de satisfacción de una propiedad y su posible evolución en los refinamientos del sistema.

- Síntesis: A partir de un conjunto de requisitos, se sintetiza automáticamente un grafo de estados según el formalismo definido, proporcionando un prototipo del sistema. El proceso de síntesis se realiza de manera incremental, a medida que el usuario especifica los requisitos del sistema. Dicho proceso automatiza las posibles tomas de decisión, pudiendo sintetizar familias de sistemas que satisfacen el conjunto de requisitos especificados. Además, el proceso de síntesis definido permite reutilizar parte de la síntesis de sistemas con conjuntos de requisitos comunes.

- Diseño de la arquitectura: La definición de requisitos de sincronización y de operadores arquitectónicos permite construir una especificación con estructura del sistema. La elección de los operadores arquitectónicos de E-LOTOS hace que el sistema obtenido (arquitectura inicial del sistema) sirva como entrada en el entorno transformacional LIRA [PA95, GS99], donde puede completarse, en una segunda etapa más orientada hacia la implementación, el proceso de refinamiento.

- Mantenimiento: La definición incremental del proceso de desarrollo permite definir una fase de mantenimiento a nivel de requisitos, ya que la especificación de cada requisito supone un cambio en el sistema, creando versiones del mismo. Cada versión del sistema puede recuperarse a partir de los requisitos que satisface y de las tomas de decisión llevadas a cabo en el proceso de síntesis.

Como complemento al trabajo teórico descrito se ha implementado una herramienta software que integra la totalidad de algoritmos desarrollados. Dicha herramienta incluye una base de datos y una interfaz gráfica que facilita su acceso en un entorno Web.

Abstract

This PhD. thesis is concerned with the discipline called Software Engineering. The main goal of Software Engineering is to improve the quality of software systems by providing theories, methods and tools for their development.

The quality of software systems was reduced when traditional methods of software development were applied to complex software systems (distributed systems). It was mainly due to the impossibility of proving the system until its implementation was developed, increasing the costs of the product.

Formal techniques, which are based on using mathematics as a vehicle to make system specifications with formally defined syntax and semantics, allow proving and validating systems at all stages of the development process. Therefore, using formal techniques reduces the risk of propagating errors during the software development process. However, in the software industry there is still great reluctance to accept mathematically based software engineering methods. Some of the possible reasons why formal methods have not yet found widespread acceptance in software engineering are: the users' difficulty to understand the formal specifications; the lack of suitable tools (most of which are still prototypes, contain many bugs and are difficult to use); and the deficient integration of formal methods in the software development process. Moreover, the main advantage of these techniques (proving system at all stages of the software development process) is notably limited when they are applied to develop complex systems, due to the high computational load of the formal verification. An alternative to solve this problem is making simple verifications by consecutive refinements of the system. This process ends when the expected product is obtained.

This thesis introduces an incremental model of software development process which is totally formalized. It is focused on the first stages of the software development process (specification, requirements analysis and verification) but it also supports the phases of architectural design and maintenance. The proposed software development model is based on its incremental nature, since the system is successively refined until the final product is obtained. Each refinement is deal with in a different way, according to the development stage. This approach is based on two main aspects: on the one hand, to take advantage of the different kinds of formal description techniques; on the other hand, to deal with the refinements according to their context.

Next, the main features and contributions of the proposed methodology are summarized. They are classified according to the stage of the defined software development process:

- Requirements specification: A temporal logic (a property-oriented formal technique) with semantics close to natural language is defined for specifying system requirements. A new degree of specification is introduced (referred to as *unspecified*) to enrich the expressivity of the logic. Therefore, unspecified parts of system can be specified in subsequent system refinements. Besides, a graphical representation of the logic is provided, due to the relevance of establishing the initial system requirements accurately. To achieve it, a formalism based on graphs with three-valued (possible, non-possible and unspecified) arcs is defined. Unspe-

cified arcs provide different ways to the system evolution and allow automatically carrying out the decision making of the system refinement process.

- Verification: A verification technique based on *model checking* is provided. It uses the defined temporal logic to specify system properties, and the three-valued graphs to make the verification. A satisfaction relation with six degrees of truth is defined. It allows enriching the expressivity of the defined formalisms and providing a detailed verification result, including the current satisfaction degree of the property and its possible evolution during the subsequent system refinements.

- Synthesis: A system prototype is automatically synthesized from the requirements specification by using the formalism based on graphs. The synthesis process is carried out in an incremental way, adding the requirements as the user specifies them. Decision making is automated into this process, being possible synthesizing system families which satisfy the specified requirements. Moreover, the defined synthesis process allows reusing part of systems with subsets of common requirements.

- Architectural Design: System architecture is provided by defining requirements of synchronization and architectural operators. The choice of the E-LOTOS architectural operators makes possible using the obtained system (the initial architecture of the system) as the input of the transformational environment called LIRA [PA95, GS99] in which the system can be refined by specifying E-LOTOS transformations.

- Maintenance: Each specified requirement causes a system refinement. It allows defining system versions in which the maintenance is made at the requirements specification level. A system version is identified by a set of requirements and the decision making carried out during the synthesis process.

A software tool has been implemented as a complement of the theoretical work described above. It integrates all the developed algorithms, including a relational database and a WWW interface.

Índice General

Índice de Figuras

Índice de Tablas

Índice de Algoritmos

Índice de Ejemplos

Índice de Definiciones

Parte I

Introducción

Capítulo 1

Ámbito y Objetivos de la Tesis

1.1 Introducción

La rápida evolución de los ordenadores ha hecho que su aplicación haya sufrido un proceso de continua adaptación durante los últimos 30 años. Inicialmente, las computadoras surgieron como una herramienta de apoyo al cálculo científico, desarrollándose hardware que permitía automatizar dichos cálculos. La aparición del software dotó de una mayor flexibilidad a las computadoras, permitiendo el desarrollo individual de programas o algoritmos que luego se introducían y ejecutaban en las computadoras. El rápido avance en el desarrollo hardware, tanto en el descenso de los costes como en el aumento de la potencia de cálculo de las computadoras, permitió abrir el campo de acción de las mismas, reducido hasta entonces al cálculo matemático.

Se crean, por tanto, nuevas expectativas, lo que conlleva el desarrollo de programas –software– cada vez más complejos. Esta complejidad aumenta con la transformación del ordenador en un nodo de comunicación, lo que permite el desarrollo de nuevos sistemas complejos –sistemas distribuidos– entre los que se encuentran los sistemas software de comunicaciones. Para el desarrollo de estos sistemas software complejos se aplican, inicialmente, los mismos métodos que se aplicaban para el desarrollo de los primeros sistemas software, orientados al cálculo científico. Esto da lugar a una crisis en el proceso de desarrollo de los sistemas software, debido al aumento de los costes y a la baja calidad del software desarrollado; en contraposición con el continuo progreso ya mencionado del hardware, dando origen a lo que se denominó *la crisis del software*.

El problema reside en aplicar métodos conocidos, y desarrollados para otros fines, a las nuevas expectativas que dan origen a los sistemas software complejos. Es necesario, por tanto, un proceso de adaptación, que permita identificar las características de los nuevos productos a desarrollar, así como definir los mecanismos necesarios para reducir los costes y aumentar la calidad de los productos desarrollados. Surge así una nueva área de investigación conocida como *Ingeniería del Software* cuyo objetivo principal es proporcionar teorías, métodos y herramientas para el desarrollo de sistemas software de calidad [NR69].

Son muchos los métodos de desarrollo software estudiados y aplicados en los últimos años,

gracias a los cuales se han logrado grandes progresos en los objetivos planteados –el desarrollo de software de calidad a costes reducidos–; pero lejos de obtener un método o técnica de aplicación general, existe una gran variedad y controversia en los mismos. La comparación con otras ingenierías más asentadas (ingeniería civil, mecánica, etc.) muestra el menor esfuerzo que se dedica en la ingeniería del software al proceso de especificación y análisis de requisitos, así como al proceso de diseño. La razón que explica este hecho, responsable en gran medida de las disfunciones del proceso software, es la escasa formalización de estas fases, lo que imposibilita la realización de tareas de verificación y validación a este nivel. Desde hace años se han dedicado grandes esfuerzos de investigación al empleo de métodos formales en la ingeniería del software. En este ámbito se sitúa el trabajo de esta tesis.

La aplicación práctica de dichos métodos formales en la industria del software encuentra una gran resistencia, debido fundamentalmente a dos deficiencias: la falta de herramientas prácticas que demuestren sus ventajas teóricas; y la necesidad de un nuevo proceso de adaptación en los usuarios y diseñadores, que no están acostumbrados a la utilización de este tipo de técnicas. En cualquier caso, existen argumentos en los dos sentidos, a favor y en contra, y sólo la aplicación real de dichas técnicas dentro de la industria podrá clarificar su situación.

En las siguientes secciones de este capítulo se describen los aspectos más relevantes de la ingeniería del software, definiendo conceptos relativos a la misma. A continuación, se hace una breve introducción a los sistemas distribuidos y a los métodos formales, apuntando sus ventajas e introduciendo la aplicación de las técnicas formales en el proceso de desarrollo de sistemas software. Finalmente, se muestran los objetivos de esta tesis y se hace una breve descripción de la organización de la memoria.

1.2 Ingeniería del Software

El objetivo del desarrollo software es crear sistemas que satisfagan las necesidades de los clientes y de los usuarios. Para ello se debe garantizar la calidad del sistema final, lo que hace necesario disponer de un proceso de desarrollo de calidad, definiendo sus actividades y dependencias, así como documentando sus características fundamentales.

Un **proceso de desarrollo software** se define [Som95] como el conjunto de actividades necesarias para el desarrollo de un sistema software. En la actualidad existen varios **modelos del proceso de desarrollo software** que organizan estas actividades de manera diferente, agrupándolas en un conjunto de fases claramente definidas. Cada fase parte de una entrada y produce unos resultados que sirven de entrada a la fase siguiente. El secuenciamiento de dichas fases y los resultados de cada una de ellas varía de un modelo a otro.

Los modelos del proceso de desarrollo software, también denominados **modelos de ciclo de vida**, son en la actualidad uno de los principales campos de investigación dentro de la ingeniería del software. A continuación se describen, brevemente, cada una de las fases identificadas en la mayoría de los modelos de ciclo de vida desarrollados hasta el momento.

- **Especificación y análisis de requisitos**. En esta fase se establecen los objetivos del sistema, las necesidades de los usuarios o clientes y el dominio de aplicación. Esta fase concluye cuando se dispone de una especificación del sistema software.

  En la actualidad, las actividades de esta fase se engloban, dentro de la ingeniería del software, en una disciplina denominada **Ingeniería de Requisitos**, en adelante **RE** (**R**equirements **E**ngineering). En [Std84] se define RE como el proceso de adquisición, refinamiento y consulta de las necesidades de un cliente para el diseño y desarrollo de un sistema software.

  En [Std90] se define el término **requisito** como: (a) Una condición o capacidad que un usuario necesita para resolver un problema o alcanzar un objetivo. (b) Una condición o capacidad que debe poseer el sistema para satisfacer un contrato, norma, especificación, u otros documentos formales. (c) Una representación documentada de una condición o capacidad satisfaciendo (a) o (b).

  Los requisitos se dividen en **funcionales**,[1] que especifican los servicios que debe proporcionar el sistema; y en **no funcionales**, que especifican las restricciones bajo las cuales deberá operar el sistema (tiempo de respuesta, consumo de recursos, etc.).

- **Diseño**. Tiene como objetivo la determinación de la arquitectura del sistema, descomponiendo el sistema en módulos o componentes. Toma como entrada la especificación del sistema obtenida en la fase anterior, y concluye cuando se dispone de la documentación de cada uno de los módulos, el mecanismo de comunicación entre los componentes del sistema, los algoritmos y las estructuras de datos.

  Esta fase suele dividirse en dos partes: en la primera se obtiene un diseño de alto nivel de cada uno de los componentes o módulos del sistema; y en una segunda parte, se refinan dichos módulos hasta obtener un diseño con el nivel de detalle suficiente para pasar a la siguiente fase de implementación.

- **Implementación**. A partir de la arquitectura software definida se construyen los componentes del sistema, de forma que puedan ejecutarse en el hardware seleccionado.

- **Prueba**. El objetivo de esta fase es asegurar que el sistema implementado satisface los requisitos especificados (verificación), y las expectativas del cliente (validación). A este proceso se le denomina [Som95] proceso de **V**erificación y **V**alidación (**V**&**V**). En primer lugar, se realizan pruebas unitarias de cada uno de los módulos del sistema. Una vez realizadas las pruebas individuales, se añaden gradualmente cada uno de los módulos del sistema, realizando pruebas de integración.

  Existen dos tipos de técnicas aplicables a esta fase: las **técnicas estáticas**, que realizan el análisis sobre representaciones del sistema, tales como la especificación de requisitos, diagramas del diseño o código fuente de los programas; y las **técnicas dinámicas**, que necesitan de una implementación o prototipo sobre el que evaluar el sistema mediante un conjunto de pruebas de ejecución o tests.

[1] En este trabajo nos centraremos en los requisitos funcionales.

Según lo expuesto anteriormente, las técnicas estáticas sólo pueden realizar tareas de verificación (comprobaciones con respecto a especificaciones del sistema), mientras que las técnicas dinámicas sí pueden poner de manifiesto que el sistema satisface las expectativas del usuario, a través de tests de comprobación del funcionamiento del mismo.

- **Operación y mantenimiento**. Una vez que el software ha sido aceptado por el cliente, se instala y se pone en funcionamiento. A partir de aquí comienza una fase de mantenimiento en la que se subsanan errores no detectados en fases anteriores y se realizan modificaciones en el sistema a medida que van surgiendo nuevos requisitos. Existen tres tipos de mantenimiento de software, con características claramente diferenciadas:

  - **De Corrección**: Se centra en la localización de errores. Los de codificación son los más sencillos de corregir; seguidos de los de errores de diseño, ya que conllevan reescribir componentes del programa; mientras que los errores procedentes de la fase de captura y análisis de requisitos son los más costosos, ya que suponen un rediseño del sistema desde su fase inicial.
  - **De Adaptación**: No implica un cambio en la funcionalidad del sistema. Se refiere a cambios en el entorno del sistema, como el sistema operativo o la plataforma hardware sobre la que se ha desarrollado.
  - **De Mejora**: Conlleva añadir nuevos requisitos funcionales y/o no funcionales, debido a cambios en las expectativas de los usuarios.

  En la actualidad, la fase de mantenimiento constituye un ciclo de vida software completo [Dav93], ya que, si se produce un cambio en los requisitos del sistema, es necesario volver a realizar, total o parcialmente (dependiendo del grado de reutilización que permita el proceso de desarrollo software utilizado) cada una de las fases anteriormente descritas. Esta es la razón por la que esta fase es la más costosa del ciclo de vida, y de ahí la importancia de realizar procesos de desarrollo software que permitan cambios en el sistema con un coste reducido.

1.3 Sistemas Distribuidos

El rápido progreso de las aplicaciones de los ordenadores, junto con su cada vez menor precio y mayor potencia de cálculo, ha dado lugar a la aparición de los sistemas distribuidos. Un sistema distribuido se compone de un conjunto de ordenadores autónomos que trabajan de forma coordinada para la realización de una determinada tarea. Estos sistemas suelen estar compuestos por un conjunto de procesos concurrentes que interactúan entre sí. Éste es el entorno típico de las aplicaciones telemáticas, en cuyo ámbito se centra este trabajo.

Una característica común de muchos sistemas distribuidos (protocolos de comunicación, servicios telemáticos, etc.) es que su comportamiento observable no termina nunca, es decir, su

comportamiento se caracteriza más por reaccionar a estímulos del entorno que por terminar produciendo algún tipo de resultado final. Pnueli [Pnu85] denominó **reactivos** a este tipo de sistemas software.

Un **sistema reactivo** es aquél que no se puede caracterizar adecuadamente mediante una semántica relacional, es decir, mediante una relación entre un conjunto de entradas y sus correspondientes salidas. Es preferible describir este tipo de sistemas a través de la interacción con su entorno. Un sistema reactivo se puede modelar como un generador de trazas, siendo una **traza** una secuencia finita o infinita de estados o eventos. Cada una de estas trazas puede verse como una secuencia de ejecución. El conjunto total de trazas constituye el comportamiento observable del sistema.

Existen dos posibilidades para expresar el comportamiento observable de un sistema: especificar **explícitamente** el orden de los eventos que pueden suceder en el sistema; o definir las propiedades del sistema, restringiendo **implícitamente** el orden posible de eventos. Una **propiedad** es un conjunto de trazas, siendo una traza una sucesión ordenada de eventos. Un programa satisface una propiedad si todas sus posibles trazas pertenecen al conjunto de trazas de la propiedad.

1.4 Los Métodos Formales en el Proceso de Desarrollo Software

1.4.1 Introducción

Los modelos tradicionales de ciclo de vida de un sistema software (modelo de ciclo de vida en cascada) muestran numerosas deficiencias cuando se aplican a sistemas complejos. Estas deficiencias son motivadas, en gran medida, por la imposibilidad de realizar pruebas hasta la fase de implementación. Esta problemática se acentúa en los sistemas distribuidos, ya que estos son intrínsecamente más complejos que los sistemas secuenciales clásicos. Esta complejidad se deriva de la necesidad de especificar un conjunto de sistemas secuenciales que deben cooperar y evolucionar en paralelo; siendo la complejidad del sistema final mucho mayor que la suma de las complejidades de los sistemas secuenciales componentes.

Las soluciones a este problema conceden al ordenador un papel más importante en el proceso de diseño y desarrollo de estos sistemas, de forma que el papel del diseñador quede restringido a la toma de decisiones (parte creativa del proceso), dejando al ordenador las tareas de manipulación, análisis y documentación (parte menos creativa).

El uso de una notación formal es decisiva para la solución de los problemas planteados. Por una parte, para capturar en el ordenador toda la información importante de las fases de especificación, diseño, implementación, prueba y mantenimiento, automatizando en la medida de lo posible el proceso de desarrollo software; y por otra, para obtener una especificación precisa, completa y consistente de los requisitos del sistema.

La especificación formal utiliza herramientas matemáticas para construir modelos de sistemas, no ambiguos y susceptibles de verificación. Las matemáticas usadas para la especificación formal

de sistemas son la teoría de conjuntos, la lógica y el álgebra. El objetivo es escribir especificaciones de sistemas con una sola interpretación posible, consistentes y completas.

En otras ingenierías, las matemáticas constituyen una herramienta fundamental de estudio y análisis. Sin embargo, a pesar de las ventajas potenciales que ofrece la aplicación de técnicas formales en la ingeniería del software, están escasamente implantadas a nivel industrial, constituyendo, actualmente, una de las líneas de investigación más importantes dentro de la ingeniería del software. En la sección 1.4.4 se exploran las razones de esta situación.

1.4.2 Las Técnicas de Descripción Formal

Es conveniente distinguir entre **métodos formales** y **métodos de especificación formal**. Los primeros se refieren a cualquier técnica, basada en el uso de las matemáticas, aplicable al desarrollo de sistemas software. Los métodos de especificación formal, en los que se centra este trabajo, son métodos formales dedicados a escribir especificaciones de sistemas. La mayor parte de los métodos de especificación formal son notaciones que permiten especificar y razonar sobre el comportamiento de sistemas. Esta característica hace necesaria la experiencia y adecuada formación del equipo de desarrollo para trabajar con abstracciones y modelos de sistemas.

Una **especificación formal** es una expresión matemática que contiene la descripción de un sistema. Una característica deseable, y en algunos casos –en la especificación de normas– necesaria, es que la especificación sea independiente de la implementación; es decir, los detalles de implementación deben incorporarse en fases posteriores.

En la actualidad, existe un amplio consenso en considerar las técnicas de descripción formal, en adelante **FDT**s (**F**ormal **D**escription **T**echniques), como una forma adecuada de describir el comportamiento de sistemas reactivos de procesamiento de la información. Aunque el término FDT se acuñó para denominar los métodos y lenguajes formales normalizados [Tur93], actualmente se utiliza para denominar cualquier técnica o método que permite definir completamente el comportamiento de un sistema (hardware o software) mediante un lenguaje con sintaxis y semántica formales[2].

Las principales mejoras que se persiguen con el empleo de las FDTs son: corrección del sistema construido respecto a una especificación formal; mejora de la calidad del sistema (entendiendo calidad como el grado de cumplimiento de los requisitos y expectativas planteadas); y aumento de la productividad. En la literatura se pueden encontrar muchos tipos distintos de FDTs.

Las características deseables de una FDT dependen de la fase del proceso de desarrollo en la que ésta se aplique [Bro96]:

- En la fase inicial, de captura y especificación de requisitos, es preferible obtener una especificación flexible del sistema en función de sus propiedades más importantes.
- En la fase de diseño de la arquitectura es necesario poder describir los componentes del

[2] A partir de ahora utilizaremos el término FDT con este significado.

sistema, su interfaz y cómo interactúan entre ellos. Es decir, es necesario poder expresar la estructura del sistema especificado.

- En la fase de implementación los componentes tienen que describirse de forma que su comportamiento pueda ser generado por máquinas, y por tanto, es necesario utilizar un lenguaje de programación. El proceso de traducción de un lenguaje de especificación a un lenguaje de implementación se puede automatizar en gran medida [MdM88]. Por tanto, si se emplean FDTs, podemos decir que esta fase es semiautomática.

En la actualidad, no existe homogeneidad en las técnicas de descripción formal sino que existe un amplio conjunto de técnicas. A continuación, se resumen los tipos de técnicas existentes, sus ventajas e inconvenientes.

La clasificación de las FDTs más difundida distingue tres categorías[3]:

1. **Técnicas algebraicas**. Un sistema se modela mediante álgebras *multi-sorted* como un conjunto de tipos y de operaciones sobre esos tipos. Cada tipo (*sort*) tiene un conjunto de valores y un conjunto 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 componentes del sistema, considerando en este nivel de abstracción el sistema como un conjunto de tipos abstractos de datos.

   Las técnicas algebraicas presentan problemas para la verificación formal de la completitud de las especificaciones.

   Ejemplos de métodos de especificación formal algebraicos son: Larch y OBJ.

2. **Técnicas basadas en modelos matemáticos**. Basadas en teoría de conjuntos y lógica de primer orden, las más conocidos son Z, VDM y B.

   El espacio de estados del sistema se modela a través de sus componentes que son a su vez modelados mediante conjuntos, funciones, etc. Se pueden formular condiciones invariantes sobre el espacio de estados mediante predicados sobre sus componentes. Las operaciones que manipulan el espacio de estados se especifican mediante predicados que relacionan diferentes estados de los componentes.

   La especificación explícita del espacio de estados y de operaciones sobre estados constituye la principal diferencia con las técnicas algebraicas, que no utilizan los estados para modelar un sistema sino tipos abstractos de datos y operaciones sobre tipos.

3. **Técnicas basadas en álgebra de procesos**. Las técnicas citadas anteriormente modelan principalmente propiedades funcionales y comportamiento secuencial. Este hecho limita su aplicación a sistemas secuenciales (sistemas no concurrentes).

[3] En el capítulo 2 se detallan las características más significativas de cada una de las técnicas de descripción formal citadas.

Las álgebras de procesos se especializan en modelar las interacciones entre procesos concurrentes. En este tipo de sistemas, propiedades como: seguridad (*el sistema no hace nada malo*) y viveza (*finalmente algo bueno ocurrirá*) son muy importantes.

Estas técnicas son especialmente útiles en la especificación de sistemas distribuidos y concurrentes, como son los protocolos y servicios de telecomunicaciones.

Ejemplos de FDTs basadas en álgebra de procesos son CCS, CSP y LOTOS.

4. **Técnicas basadas en lógica.** Las diferentes versiones de lógica temporal pueden usarse para especificar sistemas.

   Estas técnicas permiten especificar explícitamente las propiedades de seguridad y viveza de un sistema. Un sistema se especifica a través de un conjunto de fórmulas de la lógica que definen relaciones y sucesos que ocurren en el tiempo. Permiten trabajar con especificaciones parciales de un sistema.

   Estas técnicas son muy apropiadas para la especificación de sistemas de tiempo real.

   Su principal inconveniente está en su incapacidad para expresar la estructura –arquitectura– de un sistema, de ahí que se utilicen con más frecuencia en las primeras fases del diseño.

De la clasificación anterior se puede extraer como conclusión que ningún tipo de técnica se adapta perfectamente a todas las fases del diseño y desarrollo de un sistema concreto.

A continuación, se muestra una clasificación de las FDTs en función de los aspectos del sistema que modelan:

- **Funcionalidad del sistema** (son la mayor parte).
- **Propiedades concurrentes**. De especial interés para la especificación de sistemas distribuidos y de comunicaciones.
- **Tiempo**. De especial interés para la especificación de sistemas en tiempo real. Están menos desarrolladas que las dos anteriores.

Otra clasificación muy extendida de las FDTs define dos grandes grupos [Got92]:

- **Constructivas**. Las especificaciones escritas con estas técnicas son ejecutables. Su principal ventaja es su adecuación para realizar un prototipado rápido del sistema final. Su principal inconveniente es la imposibilidad de especificar de forma explícita las propiedades de un sistema, lo que impide la verificación explícita de propiedades. Ejemplos de FDTs constructivas son: LOTOS, CSP, etc.
- **No constructivas u orientadas a propiedades**. Las propiedades del sistema se expresan de forma explícita. Permiten tratar cada propiedad de forma separada lo que ayuda a decidir si es o no esencial. Su principal inconveniente reside en la dificultad para decidir si la especificación de un sistema es o no completa, es decir, si define únicamente comportamientos

correctos. Este tipo de FDTs son formalismos basados en lógica, como la lógica modal y la lógica temporal.

Por tanto, las técnicas de descripción formal constructivas y no constructivas tienen características complementarias que sugieren combinar su empleo dentro del proceso de desarrollo: FDTs orientadas a propiedades (no constructivas) en la fase inicial de especificación de requisitos y FDTs constructivas en la fase de diseño.

1.4.3 Verificación Formal

Las FDTs proporcionan una marco adecuado para reducir el riesgo inherente a la fase de captura y especificación de requisitos del sistema. No obstante, es importante ser consciente de que una especificación de un sistema escrita en una notación formal puede contener también errores. Para aumentar la confianza del diseñador en la corrección del método es preciso realizar operaciones de verificación sobre la especificación. Muchas veces esta verificación se realiza de manera informal a través de inspecciones y reuniones de revisión. Sin embargo, la base matemática de los métodos de especificación formal posibilita la **verificación formal** en dos aspectos: verificación formal de propiedades formuladas sobre la especificación, y verificación formal de que la implementación satisface la especificación.

Una de las más importantes motivaciones que impulsó el desarrollo de las FDTs fue la posibilidad de realizar una verificación formal del sistema obtenido. No obstante, es en este último punto donde surgen más controversias sobre la utilidad de las mismas. Por una parte, la experiencia ha demostrado que la verificación formal de grandes especificaciones es, en muchos casos, inviable por el alto coste de recursos que es necesario invertir. Parece, entonces, necesario limitar o concentrar las verificaciones a partes del sistema que por su naturaleza se consideren más críticas. En este trabajo avanzaremos por esta dirección. Además, a pesar de los numerosos trabajos de investigación realizados, hasta la fecha sólo se ha demostrado su aplicación en sistemas de tamaño pequeño o medio, lo que motiva su escasa utilización en la industria y que su uso esté prácticamente restringido a entornos académicos o de investigación.

Un método alternativo a la verificación formal citada –verificación formal de que la implementación satisface la especificación–, lo constituye un proceso de refinamiento iterativo donde la especificación formal de un sistema se obtiene tras la aplicación sucesiva de un conjunto de simples refinamientos o transformaciones aplicadas a una especificación inicial del sistema con un elevado nivel de abstracción. Cada una de las fases de este proceso toma como entrada la especificación resultante del refinamiento anterior y produce como salida otra especificación con mayor cantidad de detalles de implementación. Este proceso continúa hasta que se dispone de una especificación con el nivel de detalle suficiente como para poder traducirla a un lenguaje de programación convencional. Las especificaciones de entrada y salida de un refinamiento se pueden relacionar mediante relaciones de equivalencia o de orden. Las ventajas, a priori, de este método es que las verificaciones que se hacen son mucho más simples.

En la actualidad existen dos grandes técnicas para la realización de la verificación formal:

- **Model Checking**: El *model checking* es una técnica basada en la definición de un modelo de estados finito del sistema y la demostración de que una propiedad dada se satisface en dicho modelo.

  Básicamente, la demostración se realiza mediante una búsqueda exhaustiva en el espacio de estados, 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 un espacio de estados más grande.

  Las principales ventajas del *model checking* son: la posibilidad de automatizar totalmente las demostraciones; su rapidez; su capacidad para trabajar con especificaciones parciales; y su capacidad para proporcionar información útil aún cuando el sistema no se encuentre totalmente especificado. Estas dos últimas características hacen que el *model checking* sea una técnica especialmente útil en las primeras fases del desarrollo software, ya que, por una parte el sistema no está totalmente especificado; y por otra, el espacio de estados será más reducido que en las fases posteriores.

  Por contra, su principal desventaja es el problema de la explosión de estados.

- **Demostradores de teoremas**: 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 es un sistema formal que define un conjunto de axiomas y de reglas de inferencia.

  La demostración de una propiedad se realiza a través de la aplicación sucesiva de los axiomas, propiedades y reglas definidas en la lógica utilizada. El proceso finaliza cuando se obtiene una relación, que suele ser de equivalencia o de orden, entre la especificación y la propiedad formulada.

  Al contrario que la técnica de *model checking*, los demostradores de teoremas pueden manejar espacios de estados infinitos. Su principal desventaja reside en la dificultad de automatizar la aplicación de las reglas de inferencia para obtener un resultado sin entrar en procesos de reescritura sin fin.

1.4.4 Los Métodos Formales en la Industria del Software

Al contrario de lo que ocurre en otras ingenierías, la utilización de las FDTs es muy escasa. En la industria del software existe cierta resistencia al empleo de métodos matemáticos en el desarrollo software. Los principales argumentos para no usar técnicas formales son:

- Es difícil demostrar, a priori, que el esfuerzo invertido en el desarrollo de especificaciones formales se verá recompensado en las otras fases del proceso de desarrollo software. Los gestores de proyectos software prefieren adoptar posturas conservativas y no emplear nuevas técnicas hasta que sus beneficios no sean más claros.

- Gran parte de los ingenieros software, especialmente los que cuentan con mayor experiencia, no han sido entrenados en el uso de estas técnicas que requieren conocer conceptos de matemática discreta y de lógica.

- Los clientes no suelen conocer las técnicas de descripción formal, por lo que, en principio, son reacios a invertir en técnicas que desconocen.

- Las técnicas de descripción formal no son adecuadas para la especificación de algunos sistemas software, especialmente interfaces de usuario.

- Existe un gran desconocimiento de la utilidad de las técnicas de descripción formal existentes en la actualidad, así como de las ventajas observadas en algunos proyectos de desarrollo software.

- Una gran parte del esfuerzo de investigación en FDTs se ha centrado en el desarrollo de lenguajes de especificación y de su base teórica. Por el contrario, se ha dedicado muy poco esfuerzo al desarrollo de herramientas de soporte del proceso de desarrollo software, así como a los métodos más adecuados de uso de las mismas.

La mayor parte de los argumentos que se dan para no utilizar FDTs se basan en la dificultad de introducir su uso en el proceso de diseño, así como en la dificultad de encontrar herramientas adecuadas de soporte. Por contra, las razones que aconsejan su uso tienen como principal argumento las ventajas técnicas que aportan. Las principales razones a favor del empleo de estas técnicas son:

- El desarrollo de especificaciones formales permite una mejor comprensión de los requisitos del sistema, reduciendo errores y omisiones en la definición de los mismos. Además, proporcionan la base para un diseño elegante.

- Las especificaciones formales son entidades matemáticas que pueden analizarse empleando métodos matemáticos. Es posible demostrar la consistencia y completitud de una especificación, la adecuación entre una especificación y una implementación, y la ausencia de ciertos tipos de errores. Sin embargo, la verificación completa consume muchos recursos, de ahí que para sistemas de complejidad media o alta sea preciso recurrir a verificaciones parciales.

- El procesamiento de especificaciones formales puede automatizarse. Es posible construir herramientas software de ayuda al desarrollo, comprensión y depuración de especificaciones formales. Además una especificación formal puede ejecutarse (*animarse*) proporcionando así la posibilidad de un prototipado rápido.

- Las especificaciones formales pueden usarse para la definición de casos de pruebas.

En 1990, Hall [Hal90] argumenta en contra de las principales razones dadas para no usar los métodos formales, formulando siete mitos de los métodos formales:

1. *El uso de métodos formales produce software perfecto.* Una especificación formal es un modelo del mundo real y puede contener errores, malas interpretaciones y omisiones. Su traducción a un programa ejecutable está limitada por el hardware del ordenador, el sistema operativo y los compiladores utilizados. Sin embargo, una especificación formal permite que los errores sean más fáciles de detectar, y proporciona una base no ambigua para el diseño.

2. *Los métodos formales sirven para verificación.* La verificación es sólo una de las ventajas que proporcionan los métodos formales.

3. *El uso de métodos formales es caro por lo que sólo se justifica para sistemas con fuertes restricciones de seguridad.* Hall argumenta que el empleo de métodos formales puede suponer reducción de costes de desarrollo para cualquier tipo de sistemas.

4. *El empleo de métodos formales requiere una fuerte base matemática.* Sólo son necesarios conocimientos básicos de matemáticas.

5. *Los métodos formales incrementan los costes de desarrollo.* El reparto de costes entre las distintas fases del proceso de desarrollo software se ve modificado, dedicando mayores esfuerzos a las primeras fases, pero sin que lleve consigo un aumento del presupuesto global del proyecto.

6. *Los clientes no entienden las especificaciones formales.* Una especificación formal, con comentarios en lenguaje natural y con técnicas de animación, permite entender mejor los requisitos de un sistema.

7. *Los métodos formales sólo se han usado en el desarrollo de sistemas de escasa complejidad.* En la literatura se pueden encontrar ejemplos de sistemas complejos desarrollados con FDTs.

Más recientemente, en 1995 Bowen y Hyntchen [BH95a] formulan otros siete argumentos a favor del empleo de métodos formales en la ingeniería del software:

8. *Los métodos formales retrasan el proceso de desarrollo.* Uno de los principales problemas de la ingeniería de software, que constituye una de las principales preocupaciones de los equipos de desarrollo, es la estimación de costes del proceso software.

   El modelo de estimación de costes más usado es el COCOMO, propuesto por Boehm [Boe81]. Está basado en la ponderación de varios factores (nivel de experiencia, conocimiento del dominio de aplicación, etc.) medidos a partir de la experiencia previa en el desarrollo de otros sistemas software.

   Todavía no se dispone de esta experiencia cuando el desarrollo se acomete con técnicas de descripción formal, lo que complica todavía más la estimación de costes.

   No obstante, en la literatura se pueden encontrar numerosas experiencias de desarrollo de grandes sistemas software en los que el empleo de FDTs ha supuesto una reducción del tiempo estimado de desarrollo y, por tanto, de los costes previstos [HB95].

9. *No se dispone de herramientas de soporte adecuadas*. Muchas de las técnicas de descripción formal actuales tienen distintos tipos de herramientas de soporte asociadas, algunas de ellas de libre distribución.

   Sin embargo, muchas de estas herramientas no han alcanzado un estado estable, ya que no están suficientemente probadas.

   En este campo hay mucho trabajo por hacer. Es necesario disponer de herramientas más robustas y de más amplio espectro, de manera que den soporte a las principales actividades del proceso de desarrollo mediante FDTs: captura y análisis de requisitos; especificaciones y refinamientos de diseños; animación o ejecución simbólica de especificaciones; demostración de propiedades, etc.

   Estas nuevas herramientas, denominadas **IFDSEs** (**I**ntegrated **F**ormal **D**evelopment **S**upport **E**nvironment), deberían facilitar también mecanismos de control de configuración, de control de versión, además de otras facilidades dirigidas a facilitar las actividades de todo el proceso de desarrollo, teniendo en mente la posibilidad de que el equipo de desarrollo esté compuesto por un número grande de miembros.

10. *Los métodos formales suponen el abandono de los métodos tradicionales de la ingeniería de diseño*. Una de las principales críticas que se le hacen a los métodos formales es que no son en realidad métodos formales sino más bien lenguajes formales, es decir, proporcionan un lenguaje de especificación formal y un sistema de verificación, pero no está claro su papel dentro de los métodos estructurados de desarrollo software.

    Los métodos formales no sustituyen a los métodos estructurados de diseño, sino que ofrecen características complementarias a los mismos. No obstante, muchos autores reconocen la necesidad de seguir investigando en la integración de métodos estructurados de diseño y métodos formales.

11. *Los métodos formales sólo se aplican al software*. Los métodos formales pueden aplicarse tanto al desarrollo software como al diseño hardware. En la literatura se pueden encontrar numerosas referencias a la aplicación con éxito de demostradores de teoremas (HOL, Nuprl, etc.) y técnicas de *model checking* al diseño hardware.

    Una línea de trabajo más reciente en el desarrollo hardware es la llamada *compilación hardware*. Este procedimiento permite compilar un programa de alto nivel en una lista de componentes hardware simples (*netlist*) junto con sus interconexiones. La tecnología de **FPGAs** (**F**ield **P**rogrammable **G**ate **A**rrays) permite que este proceso sea por completo equiparable a un proceso de desarrollo software.

    Asimismo, esta tecnología posibilita la aplicación de métodos formales al diseño de sistemas hardware-software.

12. *Los métodos formales no son necesarios*. El uso de métodos formales es recomendable en cualquier sistema donde sea importante garantizar su corrección.

13. *Los métodos formales no tienen un soporte lo suficientemente amplio*. En la actualidad, existen notaciones formales con distintos tipos de expresividad y nivel de abstracción, así como libros, simposios y foros de debate sobre los métodos formales. Además, los métodos formales más populares están normalizados por organismos de normalización internacionales.

14. *Las personas especializadas en métodos formales los usan incluso en situaciones donde no son necesarios*. Como ya se comentó los métodos formales no son una panacea, y hay sistemas –interfaces de usuario– en los que su aplicación no se aconseja.

Las conclusiones que se pueden extraer de la discusión anterior es que el empleo de los métodos formales es más adecuado para el desarrollo de ciertos tipos de sistemas software, y que un uso más generalizado de estos métodos exige definir claramente su papel dentro del ciclo de vida así como desarrollar herramientas de soporte adecuadas.

1.5 Objetivos de la Tesis

El principal objetivo de la presente tesis es el estudio de los problemas citados en torno al desarrollo de software mediante métodos formales, así como la propuesta de soluciones que contribuyan a una mayor utilización de las FDTs en la industria del software. Para la consecución de este objetivo, el trabajo de investigación de esta tesis se centrará en tres líneas principales:

1. **Facilitar a los clientes la comprensión de las especificaciones formales:**

   Los clientes expresan los requisitos del sistema de una manera informal, y generalmente, a través del lenguaje natural. Para formalizar la primera fase de especificación de requisitos, de manera que el cliente comprenda y se familiarice con la especificación formal, es necesario mantener, en la medida de lo posible, el carácter informal de dicho proceso de especificación. El primer objetivo planteado es, por tanto, definir una técnica de descripción formal para la fase inicial de especificación de requisitos con las siguientes características:

   (a) Orientada a propiedades.

   En las primeras fases el usuario especifica el sistema como un conjunto de requisitos (propiedades) que éste debe satisfacer.

   (b) Cercana al lenguaje natural.

   El lenguaje causal se acerca al lenguaje natural utilizado por el cliente, ya que, permite expresar proposiciones donde una premisa causa un consecuencia [JMM95].

   Un formalismo basado en una lógica temporal y causal reúne las características anteriores.

   (c) Formalizar el carácter incompleto del proceso de especificación incremental de requisitos.

   Una de las características del proceso informal de especificación de requisitos es la incompletitud de dichas especificaciones. Esto es debido a que el usuario no especifica el sistema completamente, sino que lo hace de manera incremental, a medida

que identifica los requisitos del sistema deseado. Sin embargo, para poder obtener una especificación del sistema capaz de producir una implementación o un prototipo del mismo, es necesario obtener especificaciones completas. Esto supone considerar *verdadero* aquello que el usuario especifica y *falso* aquello que no ha especificado, perdiendo así información respecto al sistema (incompletamente) especificado.

La solución propuesta en esta tesis está basada en la formalización de las partes incompletas (no especificadas por el usuario), dotando así de una mayor expresividad al formalismo definido. El objetivo es, por tanto, permitir al usuario crear especificaciones formales sin especificar totalmente los elementos del sistema. Para ello, se introduce el concepto de **subespecificación**, que permite especificar los elementos del sistema en tres estados: *verdadero*, *falso* y *subespecificado*. Un elemento subespecificado se corresponde con un elemento que no ha sido especificado, y puede evolucionar, en futuras especificaciones del sistema, hacia cualquiera de los otros dos valores (*verdadero* o *falso*).

Para la consecución del primer objetivo se define una lógica multivalente, denominada **SCTL** [PAGDGS+97] (**S**imple **C**ausal **T**emporal **L**ogic), como enlace entre el usuario y la especificación formal.

2. **Definir e integrar una metodología formal en el proceso de desarrollo software:**

   (a) Para la definición de una metodología formal es necesario abordar, previamente, la definición de herramientas de ayuda que permitan su integración en el proceso de desarrollo software. En [CW96a] se establece, como determinante para la expansión e integración de los métodos formales, el desarrollo de herramientas de soporte; y el **análisis parcial** de un sistema como la fórmula más rentable de usar los métodos formales en un entorno industrial. El análisis parcial de un sistema se centra en el estudio de aquellas propiedades consideradas más relevantes, sin plantearse una verificación total del sistema. Es, por tanto, menos ambicioso en su planteamiento, si bien, es mucho más eficiente en lo que a coste computacional se refiere, y, en muchos casos, estos objetivos son suficientes para garantizar una calidad aceptable del sistema desarrollado.

   En este sentido, se plantea como objetivo de esta tesis explorar los distintos métodos de verificación formal existentes, proporcionando una metodología formal que permita combinar e integrar dichos métodos. Para ello, y con la idea de mantener la especificación formal cercana al usuario, se propone enriquecer el grado de satisfacción de una propiedad (*verdadero* o *falso*) mediante la definición de grados de satisfacción intermedios [PAGDGS+98]. El aumento de expresividad en los métodos de verificación permitirá al usuario reducir los errores introducidos en la fase de especificación de requisitos[4]. Para la consecución de este objetivo es necesario definir el conjunto

[4] Un error introducido en dicha fase y propagado a lo largo del desarrollo, puede hacer que el coste de su corrección se incremente en cientos o miles de veces, incluso detectándose en la siguiente fase de desarrollo [Boe76, Boe88].

de grados de satisfacción, así como dotar a éste de la estructura matemática necesaria para poder razonar, formalmente, sobre ellos.

(b) El proceso de desarrollo software consta de unas fases claramente diferenciadas que han sido comentadas brevemente en la sección 1.2. En la presente tesis se propone combinar en el proceso de diseño software, la utilización de distintos tipos de FDTs. La utilización de una FDT orientada a propiedades en las primeras fases de diseño debe ser complementada con una FDT constructiva en las fases posteriores. Para ello, se propone definir un nuevo formalismo como paso intermedio entre ambas. Dicho formalismo, denominado **MUS** (**M**odel of **U**nspecified **S**tates), está basado en grafos dirigidos en los que se introduce el concepto de subespecificación como un tercera posibilidad de etiquetar los arcos que unen dos estados del grafo[5].

(c) Para completar la metodología formal definida, se plantea como objetivo de esta tesis, realizar la fase de mantenimiento a nivel de especificación de requisitos (la fase de implementación es semiautomática). Para ello, se propone enriquecer las especificaciones en SCTL con **tomas de decisión** almacenadas mediante el formalismo MUS; así como desarrollar mecanismos de almacenamiento que permitan la reutilización de sistemas software que compartan subconjuntos comunes de requisitos. Para lograr este último objetivo, se propone: la definición de requisitos atómicos, a partir de los cuales se obtengan, recursivamente, los requisitos más complejos; y la utilización como modelo de desarrollo software, de un modelo iterativo [LG97] orientado a recoger los cambios continuos que se producen en el desarrollo de un producto software [Dav93]. En este sentido, se propone un modelo de ciclo de vida orientado a incrementos. Cada uno de estos incrementos se corresponde con la especificación de un nuevo conjunto de requisitos que produce una evolución en el sistema. Este proceso se ha denominado **síntesis incremental**.

Por tanto, este segundo objetivo se centra en la definición de una metodología formal, denominada **metodología SCTL-MUS**, que partiendo de la primera fase del ciclo de vida software, obtenga una especificación formal de la arquitectura inicial del sistema especificado. Dicha metodología involucra: la captura y especificación de requisitos, interaccionando directamente con el cliente a través de una FDT orientada a propiedades denominada SCTL; el análisis de los mismos, dotando a los métodos de verificación formal de mayor expresividad mediante la definición de grados de satisfacción intermedios; y la síntesis de una especificación formal con estructura, utilizando una FDT constructiva, a partir de una especificación SCTL y la definición de un formalismo intermedio denominado MUS.

3. **Diseñar e implementar un conjunto de herramientas adecuadas a la metodología formal definida:**

   Finalmente, el último objetivo planteado consiste en el diseño e implementación de un conjunto de herramientas de apoyo que permitan, por una parte, poner de manifiesto los pro-

[5] En contraposición con los modelos de estados clásicos, en los que los arcos se consideran *verdaderos* –si existen– o *falsos* –en caso contrario–.

blemas y soluciones planteados; y por otra, acercar a los usuarios finales la utilización y comprensión de la metodología formal definida. A continuación, se describen las funcionalidades más importantes que deberá proporcionar el conjunto de herramientas diseñadas:

(a) Interfaz para la captura, especificación y almacenamiento de requisitos SCTL.

(b) Traducción de un requisito SCTL a un modelo de estados según MUS.

(c) Representación gráfica de los requisitos especificados mediante la visualización de grafos MUS.

(d) Verificación formal.

(e) Síntesis incremental a partir de un conjunto de requisitos SCTL.

(f) Mantenimiento de los sistemas almacenados, permitiendo modificar los requisitos especificados en cada uno de ellos.

(g) Almacenamiento de requisitos SCTL a través de librerías de requisitos que permitan la reutilización de requisitos SCTL previamente especificados.

1.6 Organización de la Memoria

La presente memoria está organizada en ocho partes:

- La parte I está dedicada a la revisión del estado del arte y al planteamiento de los objetivos de la Tesis. El primer capítulo es el actual y contiene una breve introducción a la ingeniería del software, al empleo de técnicas de descripción formal en el proceso software, así como los objetivos de la Tesis. En el capítulo dos se realiza una revisión del actual estado del arte en el diseño y desarrollo de sistemas distribuidos con técnicas de descripción formal.

- La parte II se dedica por completo a la definición de los formalismos utilizados por la metodología SCTL-MUS. El capítulo tres introduce y define el modelo de desarrollo software propuesto. El capítulo cuatro se dedica a la presentación del formalismo MUS, mientras que la lógica SCTL se presenta en el capítulo cinco. En el capítulo seis se describe el algoritmo de traducción SCTL-MUS, que permite obtener la especificación según MUS de un requisito SCTL.

- En la parte III se presentan las posibilidades de verificación de sistemas especificados mediante los formalismos SCTL y MUS. Para ello, el capítulo siete define los grados de satisfacción utilizados, dotando a dicho conjunto de una estructura matemática sobre la que poder razonar formalmente. El capítulo ocho presenta el algoritmo de verificación propuesto mediante la combinación de SCTL y MUS.

- En la parte IV se describe el proceso de síntesis incremental que tiene lugar en la metodología propuesta. En el capítulo nueve se propone un algoritmo de síntesis basado en el algoritmo de traducción. En el capítulo diez se desarrollan una serie de algoritmos de síntesis cuyo objetivo es la reutilización en el proceso de síntesis. Por último, en el capítulo

once se aborda el diseño de la arquitectura, proponiendo un algoritmo que permite expresar los sistemas desarrollados como un conjunto de procesos sincronizados por un proceso denominado sincronizador.

- La parte V se dedica a la fase de mantenimiento. En dicha parte se exploran diferentes mecanismos de almacenamiento para los requisitos SCTL y se propone un sistema de reutilización de sistemas con subconjuntos de requisitos comunes.

- En la parte VI se describe la herramienta software implementada, su arquitectura y su funcionalidad. Esta parte contiene también un capítulo en el que se muestra un ejemplo de aplicación general. En dicho ejemplo se aborda la síntesis de una estación emisora según la técnica de acceso al medio CSMA/CD.

- La parte VII contiene un capítulo dedicado a la exposición de las conclusiones del trabajo, al análisis de los resultados obtenidos y a perfilar posibles líneas de trabajo futuro.

- Finalmente, la parte VIII de la memoria contiene unos apéndices donde se describe un ejemplo del algoritmo de reducción de estados y una serie de algoritmos auxiliares utilizados en las partes III y IV. Estos apéndices junto con una lista de las referencias bibliográficas citadas cierran el contenido de la memoria.

Capítulo 2

Estado del Arte

En los últimos años se han dedicado, dentro de la ingeniería del software, numerosos proyectos de investigación a los métodos formales. El resultado de estos proyectos ha sido la definición de nuevas metodologías de diseño con métodos formales, nuevas herramientas de soporte de los procesos de diseño y verificación, así como nuevos lenguajes formales. En la *Virtual Library on Formal Methods*[1] se puede encontrar una buena lista de lenguajes y métodos formales, así como muchos enlaces a otros servidores de información.

En este capítulo se describen: los modelos de proceso de desarrollo software más relevantes; el proceso de desarrollo con técnicas formales; los lenguajes de especificación formal, detallando las características de los más relevantes; algunas herramientas de verificación; así como los proyectos y resultados más significativos.

2.1 Modelos del Proceso de Desarrollo Software

No existe consenso sobre cuál es el mejor modelo del proceso software. Distintos equipos de desarrollo pueden utilizar diferentes modelos de proceso software para producir el mismo tipo de sistema software. Sin embargo, algunos modelos son más apropiados para producir ciertos tipos de sistemas, de forma que si no se utiliza un modelo adecuado puede ocurrir que el sistema software resultante sea de menor calidad.

El reparto de costes entre las distintas fases del proceso de desarrollo es difícil de determinar, habida cuenta de los distintos modelos de proceso existentes. Sin embargo, con independencia del modelo que se adopte, al menos el 60% del coste total se emplea en la actividad de evolución del sistema. La estimación de este porcentaje es pesimista, ya que la tasa de crecimiento de nuevos productos software es mucho mayor que la tasa de productos software que quedan en desuso (no tienen que ser mantenidos), por lo que el número de operaciones de mantenimiento que se realizan sigue aumentando. El proceso de diseño software deberá, por tanto, tener en cuenta la posterior evolución del sistema.

[1] `http://www.comlab.ox.ac.uk/archive/formal-methods.html`

Las características deseables de un proceso de desarrollo software son:

- **Claridad**. El proceso de desarrollo es claro cuando se entiende con facilidad.
- **Visibilidad**. Un proceso de desarrollo es visible cuando sus actividades producen resultados claros identificables externamente.
- **Facilidad de soporte**. Exige disponer de herramientas **CASE** (**C**omputer-**A**ided **S**oftware **E**ngineering) que den soporte a todas o alguna de las actividades del proceso de desarrollo.
- **Fiabilidad**. Un proceso de desarrollo es fiable cuando es capaz de detectar posibles errores.
- **Facilidad de mantenimiento**. Requiere capacidad para incorporar nuevos requisitos o modificar alguno o algunos de los existentes.
- **Rapidez**. Un proceso software es rápido cuando se puede obtener, a partir de la especificación, una implementación del sistema en un tiempo reducido.

En la literatura se pueden encontrar distintos modelos de desarrollo entre los que cabe citar por su mayor generalidad:

- **Modelo en cascada o convencional**. Tomado de otras ingenierías, es el primer modelo de desarrollo software propuesto. Ampliamente usado en la industria por su facilidad de gestión y visibilidad.

  En la figura 2.1 se representa el secuenciamiento de las actividades de este modelo de desarrollo.

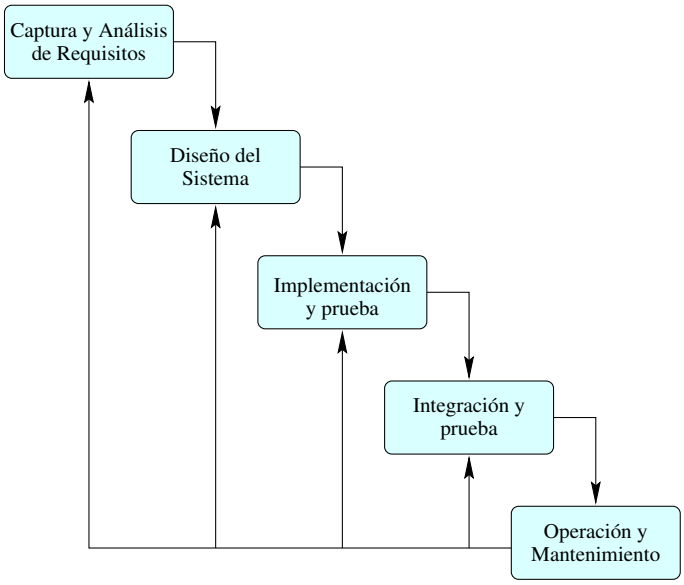


Figura 2.1: Modelo en cascada o convencional.

Sin embargo, su principal problema reside en su poca flexibilidad al separar el proceso de desarrollo en etapas totalmente distintas. En la práctica, estas etapas no tienen fronteras tan

bien definidas, lo que hace que, en no pocas ocasiones, se solapen y compartan información. En [LK95] puede encontrarse una descripción más extensa de los principales problemas de este modelo: dificultad para realizar prototipos, reutilizar software y realizar pruebas sin disponer de una implementación del sistema.

- **Modelo evolutivo**. En este modelo se entrelazan las actividades de especificación, desarrollo y validación.

  Inicialmente, se desarrolla rápidamente un sistema inicial a partir de una especificación muy abstracta. El sistema se va refinando con la información que van suministrando los clientes y/o usuarios hasta que se obtiene un sistema final que satisfaga todas las necesidades previstas.

  El sistema final obtenido puede rediseñarse para producir otro más robusto y más fácil de mantener. En la figura 2.2 se esquematiza este modelo.

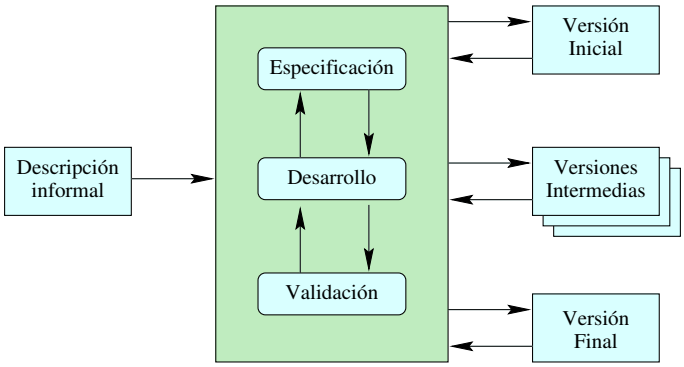


Figura 2.2: Modelo evolutivo.

  Existen dos tipos de procesos de desarrollo evolutivos:

  - **Exploratorio**. Su objetivo es trabajar con el cliente para identificar y construir el sistema final a partir de una especificación informal. El resultado del proceso es el sistema final.
  - **Prototipado desechable**. Su objetivo es entender los requisitos del cliente. El resultado del proceso es la especificación del sistema (el prototipo se deshecha).

  Los principales problemas de este modelo son [Boe88, Som95]: escasa visibilidad; los continuos cambios que hacen que los sistemas desarrollados estén deficientemente estructurados; y la necesidad de disponer, en muchos casos, de un equipo de desarrollo altamente cualificado. Estos problemas hacen que la aplicación de este modelo se suela limitar a sistemas interactivos de tamaño pequeño o mediano. La deficiente estructura dificulta las tareas de mantenimiento de ahí que se suela aplicar a sistemas con una vida corta, y a partes de grandes sistemas, especialmente a sistemas de inteligencia artificial y a interfaces de usuario.

- **Modelo transformacional**. Se basa en disponer de una especificación formal del sistema y en transformar, con métodos matemáticos, esta especificación en una implementación.

Si las transformaciones que se aplican son correctas, es posible asegurar que el sistema construido satisface la especificación, es decir, es posible obtener programas correctos por construcción. La figura 2.3 esquematiza este modelo.

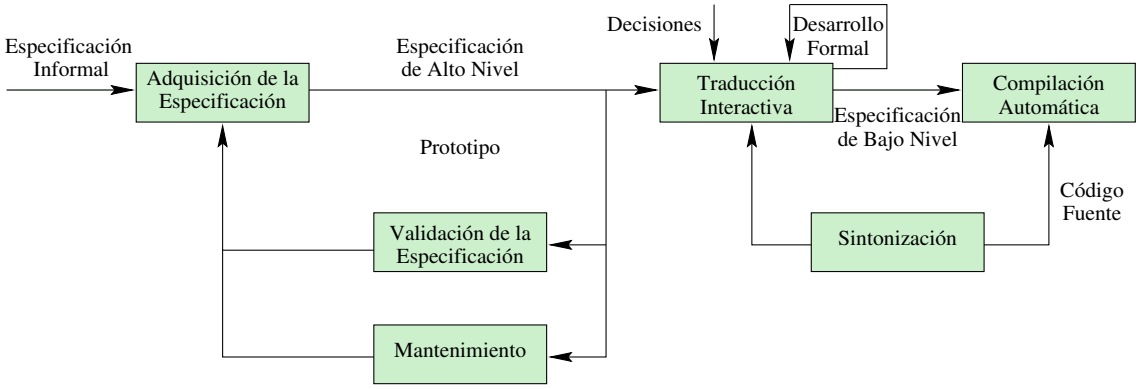


Figura 2.3: Modelo transformacional.

Otra de sus ventajas es la posibilidad de realizar el mantenimiento a nivel de especificación. Por contra, hay que destacar la necesidad de disponer de una especificación inicial correcta y de diseñadores altamente cualificados. Además, no existe apenas experiencia en la aplicación de este modelo a grandes proyectos.

- **Modelo basado en reutilización**. En este modelo se supone que alguno de los componentes del sistema final ya existe. El proceso de desarrollo se centra en integrar las partes ya existentes más que en construir todo el sistema desde el principio.

  Las ventajas que desde un punto de vista económico puede producir este modelo empiezan ahora a ser estudiadas en profundidad. Prácticamente no existe experiencia sobre el empleo de este modelo, si bien, se están haciendo numerosos estudios e investigaciones para posibilitar su uso.

- **Modelo en espiral**. Desarrollado por Boehm [Boe88] en el año 1988 con el objeto de reunir las ventajas de los modelos de proceso software en cascada y de prototipado. Se incluye el análisis de riesgo como una parte importante del proceso de desarrollo software.

  El modelo tiene la forma de una espiral (ver figura 2.4) en la que cada vuelta representa cada una de las fases en las que se estructura el proceso software, y está organizada en cuatro sectores:

  1. Definición de objetivos, alternativas y restricciones de cada fase del proyecto.
  2. Evaluación de alternativas y análisis de riesgos.
  3. Desarrollo y validación. Se elige el modelo de proceso de desarrollo que se considere más adecuado.
  4. Planificación de las siguientes fases del proyecto.

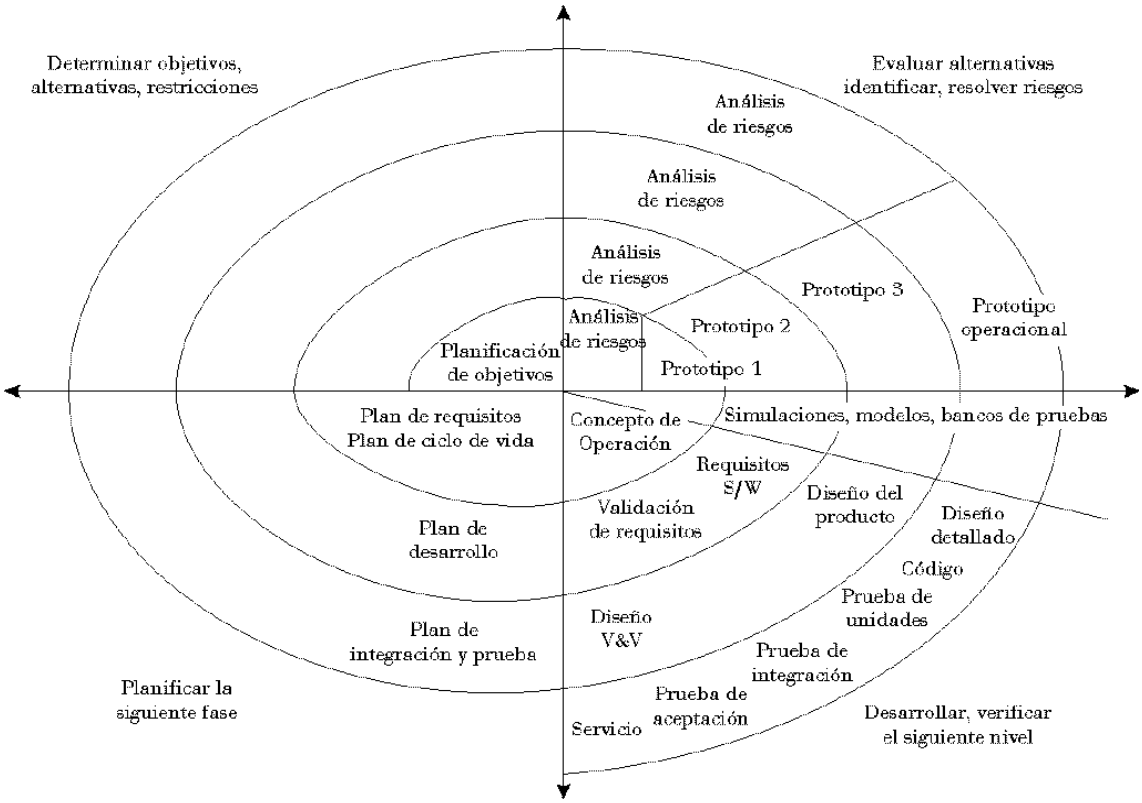


Figura 2.4: Modelo en espiral.

La principal ventaja de este modelo es su flexibilidad, ya que puede utilizar en cada fase cualquier modelo anterior, lo que permite aprovechar las ventajas de cada uno y atenuar sus inconvenientes. Su principal inconveniente reside en la necesidad de disponer de experiencia para evaluar los riesgos potenciales.

2.2 Proceso de Desarrollo con FDTs

Hasta la introducción de las técnicas de descripción formal en el proceso de desarrollo software, las especificaciones detalladas se solían escribir utilizando diagramas de flujo o **PDL**s (**P**rogram **D**escription **L**anguage). El nivel de abstracción de este tipo de especificaciones no permite una realimentación de información con la especificación del sistema a nivel de requisitos.

Las técnicas de descripción formal permiten realizar especificaciones detalladas de forma abstracta lo que permite la interacción con la especificación a nivel de requisitos, permitiendo la participación de los usuarios o clientes en las primeras fases. La figura 2.5 esquematiza las fases del proceso de especificación y su relación con el proceso de diseño. En la figura 2.6 se muestra como con el empleo de FDTs se pueden realizar en paralelo actividades de especificación y diseño así como la influencia que ejercen unas actividades en las otras.

La metodología de diseño de arriba a abajo (*top-down*) por refinamientos sucesivos es muy

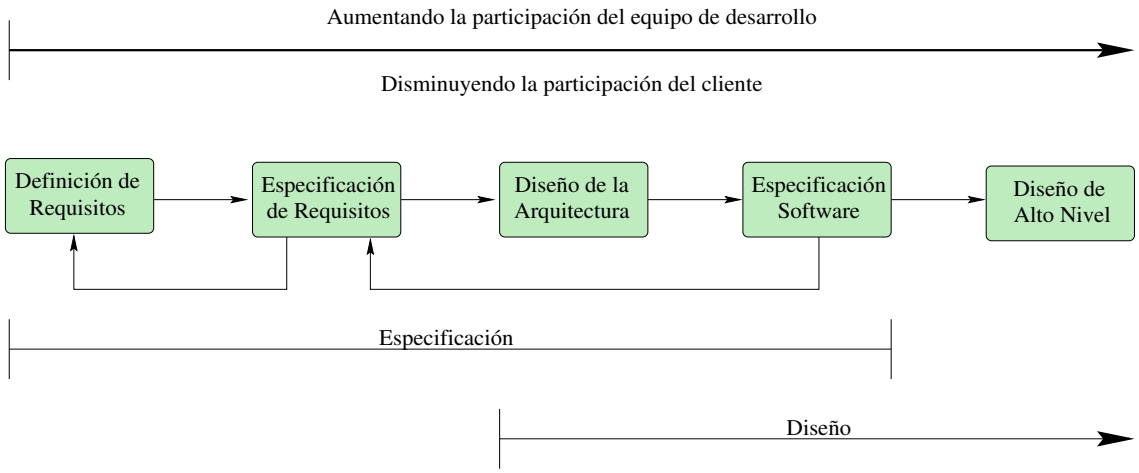


Figura 2.5: Especificación y diseño.

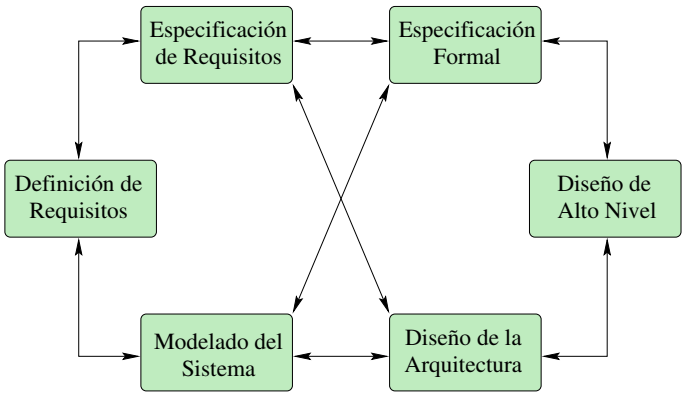


Figura 2.6: La especificación formal en el proceso software.

apropiada para el desarrollo de sistemas con técnicas de descripción formal. El proceso de desarrollo se compone de un conjunto de fases sucesivas, cada una de estas fases puede descomponerse, a su vez, en varias etapas. El proceso de desarrollo comienza con la descripción más abstracta del sistema y finaliza cuando se dispone de una versión ejecutable del mismo.

Cada etapa de desarrollo transforma la descripción del sistema en otra más refinada tras la inclusión de una decisión de diseño. Una decisión de diseño selecciona un diseño del conjunto de diseños posibles que satisfacen los requisitos considerados relevantes en el actual estado del proceso.

Para poder guiar este proceso de desarrollo es necesario poder disponer de una descripción precisa de cada etapa del desarrollo. Se seguirá un modelo de desarrollo en el que cada etapa del mismo puede descomponerse hasta en tres tareas diferentes:

1. **Diseño formal**. Esta tarea toma como entradas el diseño procedente de la fase anterior y el documento de requisitos. El objetivo de esta tarea es identificar requisitos relevantes de la etapa actual del desarrollo, así como tomar las decisiones de diseño adecuadas para obtener como salida un nuevo diseño del sistema.

2. **Evaluación**. Su objetivo es asegurar que la tarea anterior ha sido realizada correctamente, o bien detectar los errores cometidos. Para ello, se deben realizar dos evaluaciones: la primera para probar la consistencia entre el diseño actual y los requisitos del sistema, y la segunda para probar la consistencia entre el diseño actual y los procedentes de etapas anteriores.

3. **Implementación o prototipado**. Un prototipo es una implementación mínima de un sistema construida para validar la interpretación de los requisitos realizada por los diseñadores del sistema. Las distintas descripciones del sistema resultantes de cada una de las etapas de desarrollo pueden considerarse prototipos del sistema, únicamente el resultado de la última etapa implementará la funcionalidad completa del sistema.

Puede haber etapas de desarrollo en las que no se realicen todas las tareas antes citadas.

En el trabajo de esta tesis se propone una metodología de diseño de sistemas con técnicas de descripción formal basada en el modelo anterior. En la siguiente parte de la memoria se formaliza el modelo de desarrollo software propuesto, definiendo los formalismos utilizados en cada una de las fases del mismo.

2.3 Lenguajes de Especificación Formal

La especificación es el proceso mediante el que se describe un sistema y sus propiedades deseadas. 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 y ambigüedades. Con el empleo de FDTs se consigue, además, que la especificación pueda ser analizada formalmente, es decir, pueda ser probada su consistencia así como ciertas propiedades del sistema especificado. Por otra parte, una especificación es un buen vehículo de comunicación entre el cliente y el diseñador, entre el diseñador y el equipo de implementación, y entre los equipos de implementación y de pruebas. Asimismo, la especificación formal de un sistema sirve, con un nivel de abstracción mayor, como complemento a la documentación del sistema.

Las propiedades deseadas de un sistema pueden incluir tanto requisitos funcionales –funcionalidad del sistema y aspectos de estructura–, como no funcionales –restricciones en el modo de ejecución del sistema, principalmente restricciones en tiempos de ejecución y consumo de recursos–. La dificultad para encontrar un lenguaje con una potencia expresiva suficiente para poder expresar tanto requisitos funcionales como no funcionales hizo que los principales resultados se encuentren en la especificación de requisitos funcionales, y dentro de éstos en los requisitos de funcionalidad del sistema.

La necesidad de expresar formalmente todos los tipos de requisitos sugiere investigar la combinación de varios lenguajes formales, cada uno de ellos especializado en la especificación de

aspectos diferentes de un sistema.

A continuación, se describen brevemente algunos de los lenguajes formales más conocidos y usados, comentando su capacidad para expresar cada uno de los tipos de requisitos citados:

1. Lenguajes especializados en la especificación del comportamiento de sistemas secuenciales. El sistema se modela mediante conjuntos (tipos) y operaciones (funciones) sobre dichos conjuntos, a las que se les asocia pre y post-condiciones que definen la transición entre los estados del sistema.

   (a) **Z** [ISO95, Spi88, Spi92]. **Z** es una notación formal basada en lógica de predicados de primer orden y en teoría de conjuntos. Comenzó a desarrollarse en la Universidad de Oxford en el año 1980. Su nombre deriva del matemático Zermelo.

   Las especificaciones se construyen a partir de componentes denominados *schemas*, y suelen completarse con notaciones informales. Cada *schema* se compone de:

   - *Schema name*: A cada *schema* se le asigna un nombre para poder referirse a él en otras partes de la especificación.
   - *Schema signature*: En esta parte se declaran las variables de estado y el tipo de cada una de ellas.
   - *Schema predicate*: Define las relaciones entre las variables de estado mediante expresiones lógicas invariantes.

   Z permite definir funciones a partir de un conjunto de entrada (*domain*), un conjunto de salida (*range*) y unas reglas de asociación entre ambos conjuntos.

   Existen múltiples herramientas que la emplean, como **CADiZ** [JMT90], **ZANS** [Xia95] y **Cogito** [ABT95].

   A continuación, se destacan dos sistemas reales en los que se empleó Z:

   - **CICS** [HK91]. **CICS** (**C**ustomer **I**nformation **C**ontrol **S**ystem) es un proyecto desarrollado por IBM en colaboración con el Laboratorio de Computación de la Universidad de Oxford. En 1992 se utilizó Z para la nueva versión del sistema de procesamiento de transacciones de la versión CICS/ESA_V3.1.
   - **CIDS** [HP95]. Z se utilizó como especificación formal de parte del software del *Airbus A330/340*.

   (b) **VDM** [ISO93, Jon90]. **VDM** (**V**ienna **D**evelopment **M**ethod) es una colección de técnicas para la especificación formal y el desarrollo de sistemas software. Tiene su origen en un laboratorio de Viena de IBM entre 1960 y 1970. El modelo del sistema se describe mediante un conjunto de tipos de datos y una serie de operaciones sobre los mismos que se representan mediante funciones. A cada función se le asocia una *pre-condición* y una *post-condición*.

   Utiliza un lenguaje de especificación denominado **VDM-SL**. El lenguaje permite definir tipos complejos como conjuntos, registros, secuencias, etc., sobre los que se pueden especificar restricciones mediante invariantes. A diferencia de Z, en VDM-SL se

diferencian dos tipos de componentes: estados (sobre los que se pueden especificar invariantes); y operaciones (con pre y post-condiciones).

Existen varias herramientas que utilizan VDM, como por ejemplo: **IFAD VDML-SL Toolbox** [ELL94], **Mural** [BR91] y **SpecBox** [BFM89].

Dos de los proyectos más relevantes en los que se ha utilizado VDM son:

- **CDIS** [Hal96]. Parte del mecanismo de representación de información del sistema de control de tráfico aéreo de Londres fue verificado utilizando VDM. Este proyecto fue llevado a cabo en 1992 por la empresa *Praxis*.
- **ACS** [MS93]. En este caso, VDM se utilizó para especificar requisitos de seguridad de un sistema de almacenamiento de explosivos del ejército británico.

(c) **Larch** [GH93]. **Larch** es un proyecto de investigación cuyo objetivo era explorar métodos, lenguajes y herramientas para facilitar el uso de las especificaciones formales. El proyecto se inició a principios de los 80 en el Laboratorio de Ciencias de la Computación del MIT y en el Centro de Investigación de Sistemas de Digital en Palo Alto, California. Su principal característica es que contiene dos estilos de especificación.

Cada especificación se escribe en dos lenguajes:

- *Lenguaje de interfaz Larch*. Diseñado para un lenguaje de programación específico. Se utiliza principalmente para describir las interfaces entre los componentes del programa a través de tipos abstractos de datos.
- *Lenguaje compartido Larch* (**L**arch **S**hared **L**anguage). Independiente del lenguaje de programación. La unidad básica de una especificación LSL se denomina *trait* e incluye dos tipos símbolos: *operadores* y *conjuntos*. Es muy similar a otros lenguajes de especificación algebraicos.

El demostrador de teoremas **Larch Prover** [GG91] y **LCLint** [EGHT94] son dos de las herramientas desarrolladas en este proyecto.

2. Lenguajes especializados en la descripción de sistemas concurrentes. Los sistemas se modelan representando su comportamiento mediante secuencias de estados, árboles u órdenes parciales de eventos. A continuación, se citan los más relevantes:

(a) **CCS** [Mil80]. **CCS** (**C**alculus of **C**ommunicating **S**ystems), definido por Milner en 1980, fue el primer álgebra para describir concurrencia y comunicación entre procesos. Utiliza sistemas de transiciones etiquetadas, siendo estos sistemas los modelos naturales de las lógicas temporales y modales.

Concurrency Workbench [CPS93] de la Universidad de Edimburgo, **JACK** [BGL94] del IEI-CNR y **ATG FC-TOOLS** [BRRdS96] del INRIA/CMA-ENSMP son tres herramientas desarrolladas sobre CCS.

(b) **CSP** [Hoa85]. **CSP** (**C**alculus of **S**equential **P**rocesses) fue definido en un artículo de 1978 por C.A.R. Hoare, cuyo refinamiento dio origen a una notación más flexible

en 1985. El nombre se mantiene aunque la restricción de su aplicación a sistemas secuenciales se eliminó entre 1978 y 1985.

El sistema se describe mediante un conjunto de procesos independientes que se comunican a través de *canales* (esta idea es una de las bases del lenguaje de programación OCCAM [PM87, Lim88, JG88]).

Destacan las siguientes herramientas que utilizan CSP: **FDR** [Ros95], **ProBE** [Ros97] y **Caspe** [Low98].

(c) **SDL** [Z.188]. **SDL** (**S**pecification and **D**escription **L**anguange) es un lenguaje de descripción formal normalizado por **ITU** (**I**nternational **T**elecommunication **U**nion). Recomendación Z.100. Desde su primera versión (1980), SDL ha sufrido constantes revisiones. Las extensiones más recientes del lenguaje se han realizado en el área de diseño orientado a objetos.

El sistema se describe como un conjunto de máquinas de estados finitos y extendidas comunicándose mediante el intercambio de mensajes entre ellas y con el entorno. Dispone de un lenguaje gráfico (**G**raphical **R**epresentation), aunque algunos constructores sólo tienen una representación en modo texto (**P**hrase **R**epresentation).

A continuación, se enumeran las herramientas más destacadas que utilizan la técnica formal SDL:

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

Dos de los proyectos más relevantes en los que se utilizó la técnica formal SDL son:

- **INSYDE** [PJW$^+$95]. Proyecto *Sprit* dedicado al diseño híbrido hardware-software. SDL se utilizó para la especificación del software de un sistema de vídeo bajo demanda.
- **NewCore** [Hol94]. Es un sistema de telecomunicaciones en el que SDL se utilizó para su verificación.

(d) **ESTELLE** [BD87, DB89]. **ESTELLE** (**E**xtended **S**tate **T**ransition **L**anguage) es una técnica formal de especificación basada en máquinas de estados finitos y extendidas normalizada por ISO [ISO89a, ISO91]. Su objetivo inicial fue especificar sistemas distribuidos de información y se concibió particularmente para el desarrollo de protocolos de comunicaciones bajo el modelo de referencia OSI.

Su fundamento sintáctico es el Pascal y consta, al igual que este último, de una fuerte estructura de objetos tipados, lo que permite detectar las errores de la especificación en la fase de compilación.

Existen traductores de ESTELLE a C++ (**XEC** [TG98]) y a C (**EDT** [Bud92]), incluyendo simuladores y ayudas de depuración. Además, ESTELLE ha sido utilizado en

la especificación y verificación de diferentes protocolos ISO, como **ROSE** [JL95].

(e) **LOTOS** [BB89, PvED89, Tur93, BvLV94]. **LOTOS** (**L**anguage **O**f **T**emporal **O**rdering **S**pecification) es una técnica de descripción formal normalizada en 1989 por ISO [ISO89b, ISO91]. Su principal objetivo es expresar formalmente normas de servicios y protocolos OSI de una manera clara, no ambigua, precisa, completa, e independiente de la implementación.

LOTOS tiene dos partes claramente diferenciadas. La primera parte proporciona un modelo basado en álgebra de procesos para expresar el comportamiento e interacción entre procesos. Esta parte proviene fundamentalmente de CCS, de CSP y de CIRCAL [Mil85]. La segunda parte del lenguaje permite la especificación de tipos abstractos de datos y se basa en el lenguaje ACT-ONE [FEH83].

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

LOLA [QPF89], **ISLA** [GL93] y **CADP** [FGM$^+$94] son tres de las herramientas más destacadas que han sido desarrolladas para LOTOS.

(f) **Lógica Temporal** [BF96, Pnu81, Lam84, MP92]. La aplicación de la lógica modal [Sti96, Eme90] a la especificación y verificación de sistemas software ha sido de gran utilidad en el proceso de razonamiento sobre programas.

Los principios básicos de la lógica modal son compartidos también por la lógica dinámica, la lógica algorítmica y la lógica temporal. Estos principios se consideran el vehículo más adecuado para la expresión de las propiedades de un sistema [Pnu77].

La principal ventaja de la lógica dinámica (**DL**), introducida por Pratt [Pra78, Har79a], su equivalente proposicional (**PDL**) propuesta por Fischer y Ladner [FL79] y la lógica algorítmica [BKM$^+$77], es su capacidad para tratar con éxito el comportamiento antes-después (*before-after*) de los programas. Su principal inconveniente reside en su incapacidad para reflejar el comportamiento progresivo de los programas: *la variable x tomará el valor 0 en algún momento de la ejecución del sistema*.

Se han desarrollado algunas lógicas de procesos para subsanar esta dificultad, la mayor parte de ellas a nivel proposicional. Pratt [Pra79] sugirió usar dos nuevas conectivas: *during* y *preserves*. Para ello, presentó un conjunto de axiomas y reglas de aplicación en esas construcciones.

Parikh [Par78] definió el lenguaje SOAPL, demostrando que era, al menos, tan potente como el lenguaje propuesto por Pratt. Más tarde, Harel [Har79b] demostró que era estrictamente más potente. Sin embargo, la sintaxis de SOAPL es bastante compleja, lo que provoca que el conjunto de axiomas que lo define no sea muy claro.

De forma independiente, Pnueli [Pnu77, Pnu79] desarrolló la Lógica Temporal de Programas (TL) que permite realizar afirmaciones sobre el comportamiento progresivo de los programas. TL permite expresar conceptos tales como ausencia de bloqueo (*deadlock*), viveza o vivacidad de sistemas (*liveness*) y exclusión mutua. La principal limi-

tación de TL es que no proporciona mecanismos para darle nombre a programas, con lo que obliga a que la especificación se refiera a todo el programa dificultando así la especificación de grandes sistemas.

2.4 Verificación Formal

2.4.1 Model Checking

Inicialmente, el *model checking* [McM98] se usó en la verificación de sistemas hardware y de protocolos de comunicación. La tendencia actual es aplicar esta técnica al análisis y verificación de especificaciones de sistemas software. En la actualidad, existen dos caminos para realizar *model checking* [CW96b]:

- El *model checking temporal*, desarrollado de forma independiente por Clarke y Emerson [CE81] y por Queille y Sifakis [QS82] en la década de los ochenta. El sistema se modela mediante un sistema de transiciones de estados finitos, mientras que la especificación se realiza a través de la lógica temporal.

  Clarke y Emerson desarrollaron las herramientas **EMC** [CES86] y **SMV** [McM93]. Queille y Sifakis desarrollaron **CÆSAR** [QS82]. Todas estas herramientas utilizaban la lógica CTL [CE81, ES89].

- Utilizar autómatas para modelar el sistema y para su especificación. La verificación consiste en determinar si el comportamiento del sistema es el que define su especificación. Para ello, existen varios tipos de pruebas: inclusión de lenguajes [HK90, Kur94b], relaciones de orden [CPS93, Ros94] y equivalencia observacional [CPS93, FGK$^+$96, RS90].

  Las herramientas **SPIN** [Hol91] y **Murϕ** [Dil96] utilizan este enfoque.

En 1986, Vardi y Wolper [VW86] demostraron como se podían refundir estos dos caminos.

El principal problema de esta técnica de verificación es la explosión de estados. Las líneas de investigación que se están desarrollando para solucionar o reducir dicho problema se encaminan en las siguientes direcciones:

- Representación eficiente del espacio de estados. En esta línea cabe citar los trabajos de McMillan y Bryant [Bry86] con los Diagramas de Decisión Binaria (BDD) con los que consiguieron aumentar de forma considerable el tamaño del espacio de estados que se podía verificar.

- Otra línea de trabajo que parece va a producir resultados esperanzadores se dirige a utilizar la información de orden parcial disponible [Pel96], reducción de localización [Kur94a, Kur94b] y la minimización semántica [ECB96] para eliminar estados innecesarios del modelo del sistema.

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]. Mediante el empleo de técnicas de abstracción apropiadas se pueden realizar demostraciones sobre sistemas de prácticamente un número ilimitado de estados [CGL92].

A continuación, se citan algunos de los ámbitos de los proyectos en los que se ha utilizado esta técnica de verificación:

- 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 [ECB96] (verificado con Concurrency Workbench).

2.4.2 Demostradores de Teoremas

En los últimos años se han dedicado grandes esfuerzos de investigación en este tema, que han dado lugar a la aparición de numerosos demostradores de teoremas. Los demostradores de teoremas se están usando cada vez más en la verificación de sistemas con fuertes requisitos de seguridad (*safety-critical*), así como en diseños hardware y software.

Los demostradores de teoremas se suelen clasificar, en función del grado de automatización de su funcionamiento, en dos grupos:

- Muy automatizados. Resultan útiles como procedimientos generales de búsqueda, habiendo resuelto con éxito algunos problemas combinacionales.
- Interactivos. Se adaptan mejor al desarrollo formal sistemático de sistemas. Requieren la intervención humana y, por tanto, son lentos y no exentos de errores.

A continuación, se describen brevemente los demostradores de teoremas más representativos clasificándolos según el grado de automatización que proporcionan:

- Herramientas de deducción automáticas guiadas por el usuario. Tienen como característica común que su operación se guía por una secuencia de lemas y definiciones, si bien cada teorema se demuestra automáticamente mediante heurísticos y la aplicación de inducción, lemas, reescritura y simplificación.

  Ejemplos de este tipo de sistemas son: **Nqthm** [BM79], demostrador de teoremas de la lógica de Boyer-Moore; **ACL2** [KM95], que utiliza la lógica ACL2 (**A C**omputational **L**ogic for Applicative Common **L**isp); **Eves** [CKM+88]; **LP** [GG88]; **REVE** [Les83] y **RRL** [KM87].

- Demostradores propiamente dichos: Han sido usados para formalizar y verificar problemas complejos de matemáticas, así como para la verificación tanto hardware como software. Los ejemplo más ilustrativos de este tipo de demostradores de teoremas son:
  - **HOL**. Es un sistema demostrador de teoremas basado en lógica de orden superior, 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.

    En [GM93] se puede encontrar una descripción detallada del sistema HOL.
  - **Isabelle:** Isabelle es un demostrador de teoremas genérico. Isabelle posibilita la introducción de nuevas lógicas especificando su sintaxis y reglas de inferencia. Proporciona un alto grado de automatización.

    En [Pau94] se puede encontrar una descripción detallada de la funcionalidad de la herramienta.
  - Otros ejemplos de este tipo de demostradores son: **Coq** [CCF+85], **LEGO** [LP92], **LCF** [GMW79] 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.
  - **PVS** [ORS92] y **STeP** [Bea96] combinan potentes procedimientos de decisión con *model checking* para realizar demostraciones interactivas. PVS ha sido usado con éxito para verificar diseños hardware, así como sistemas reactivos, de tiempo real y tolerantes a fallos.

2.5 Conclusiones

El principal objetivo de los métodos formales es facilitar el desarrollo de sistemas más fiables y con mayor calidad. Con una sólida base matemática, su uso va destinado a los equipos de desarrollo tanto hardware como software. En la década pasada se han realizado notables avances en este campo.

No obstante, y a pesar de las ventajas potenciales que ofrecen las FDTs [Tur93], la situación actual es que tienen una amplia aceptación en la industria del hardware –lo que ha dado lugar al desarrollo de herramientas de verificación y validación muy potentes [DR96]– pero en la industria del software su uso suscita todavía una gran resistencia [BH95b]. En [Hal90] y [BH95a] se enumeran algunas de las posibles razones por las que se da esta situación, entre las que cabe destacar: la dificultad que tienen los clientes para entender una especificación formal; la deficiente integración de los métodos formales dentro del proceso de desarrollo software [PLB96, HB96a]; y la inexistencia de herramientas adecuadas de soporte al proceso de diseño [MFV94].

A continuación, se describen las principales líneas de investigación que, actualmente, se están llevando a cabo en el campo de los métodos formales [CW96b].

2.5.1 Conceptos Fundamentales

En el campo de los métodos formales se han producido avances significativos a partir de resultados fundamentales en otros campos de la Ingeniería del Software. Dentro de esta línea, sería importante avanzar en las siguientes áreas:

- Composición. Sería deseable disponer de una metodología de composición de métodos, especificaciones, modelos, teorías y demostraciones.

- Descomposición. Asimismo, sería de gran ayuda el disponer de métodos eficientes para descomponer una propiedad global en propiedades locales cuya verificación fuera computacionalmente más sencilla.

- Abstracción. Los sistemas reales son muy difíciles de especificar y verificar sin realizar abstracciones. Se hace necesario identificar diferentes niveles de abstracción, apropiados para ciertas clases de sistemas o de problemas.

- Reutilización. Sería mejor disponer de componentes, teorías y modelos parametrizables que definirlos cada vez que se acomete el diseño de un nuevo sistema.

- Combinación de teorías matemáticas. Muchos sistemas con fuertes requisitos de seguridad tienen componentes tanto analógicos como digitales. El razonamiento sobre estos sistemas híbridos requieren el empleo de matemática continua y discreta.

- Estructuras de datos y algoritmos. El diseño de grandes sistemas exige el manejo de grandes estructuras de datos y grandes espacios de estados. Es necesario encontrar nuevos mecanismos de almacenamiento y nuevos algoritmos de manejo de estructuras complejas de datos que optimicen el consumo de recursos.

2.5.2 Integración de Métodos

Por no existir ningún método formal apropiado para describir y analizar cualquier aspecto de un sistema complejo, parece lógico combinar el uso de diferentes métodos.

Se hace necesario realizar investigaciones en este campo para que la combinación de métodos formales no se convierta en una simple mezcla de los mismos, sino que, posibilite la disposición de visiones distintas, aunque relacionadas, de la especificación de un sistema, de forma que el análisis y refinamiento de una visión se plasme automáticamente en la/s otra/s.

2.5.2.1 Model Checking y Demostradores de Teoremas

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. 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.

  Es posible usar una lógica suficientemente expresiva para definir operadores temporales sobre sistemas de transición de estados finitos mediante puntos fijos máximos y mínimos. Para sistemas de transición de estados finitos estos puntos fijos pueden evaluarse empleando tanto *model checking* como demostradores de teoremas. Para estructuras con un espacio de estados ilimitado se puede utilizar la inducción sobre los puntos fijos.

- 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*.

- Para la verificación de sistemas compuestos de procesos de estados de finitos, puede utilizarse la inducción combinada con el *model checking*.

2.5.2.2 Integración en el Proceso de Desarrollo

Los métodos formales pueden usarse en combinación con otros métodos menos formales dentro del proceso de desarrollo software. Es, por tanto, importante definir el papel de los métodos formales dentro del proceso de desarrollo.

Los métodos formales han sido usados con éxito, casi exclusivamente, en la especificación y verificación de sistemas. Sin embargo, pueden ser también de utilidad en otras fases como el análisis de requisitos, el refinamiento y la prueba de sistemas:

- **Análisis de requisitos**. Los clientes de un sistema suelen tener una idea imprecisa de lo que quieren. Los métodos formales pueden ser una valiosa ayuda para definir los requisitos de una manera precisa.

- **Refinamiento**. El refinamiento es un proceso complementario a la verificación. Toma como entrada un sistema a un nivel de abstracción dado y produce como salida, tras la aplicación de una transformación o refinamiento, una especificación más concreta (con un nivel de abstracción menor) o una implementación del sistema.

  En la literatura se pueden encontrar muchos trabajos de investigación sobre este tema, si bien sus resultados no han sido transferidos todavía a la práctica del desarrollo software, en

gran parte, debido a la imposibilidad de disponer de un amplio catálogo de transformaciones motivado, a su vez, por la dificultad de definir transformaciones genéricas.

- **Prueba**. Los métodos formales están llamados a jugar un papel importante en el proceso de validación. Las especificaciones formales se usarán para generar casos de prueba [ROM89], y los modelos y herramientas de demostración para determinar relaciones formales entre especificaciones y pruebas, así como entre éstas y el código.

2.5.2.3 Educación y Transferencia Tecnológica

La aplicación con éxito de los métodos formales en la industria exige, por una parte, una tarea de educación: a nivel de estudiantes su inclusión en los estudios de ingeniería del software y, a nivel de ingenieros del software, el desarrollo de cursos de formación adecuados; y por otra, la transferencia tecnológica a la empresa, para lo cual es necesario disponer de herramientas potentes de soporte del proceso de desarrollo. Las características ideales que deben tener dichas herramientas son muy variadas (beneficios inmediatos y proporcionales al esfuerzo; uso múltiple e integrado; facilidad de uso y aprendizaje; eficiencia, etc.), por lo que parece adecuado construir meta-herramientas que integren diferentes métodos y herramientas.

Parte II

Definición de la Metodología Formal SCTL-MUS

Capítulo 3

Modelo de Desarrollo Software

3.1 Combinación de FDTs en el Proceso de Desarrollo Software

Tal y como se describió en el capítulo 1, parece apropiado el empleo combinado de FDTs orientadas a propiedades y FDTs constructivas, ya que proporcionan características complementarias, según la fase del proceso de desarrollo software en que se utilicen:

- **Fase de especificación y análisis de requisitos:** La utilización de técnicas de descripción formal se plantea como una solución para detectar los errores en las primeras etapas de captura y especificación de requisitos, ofreciendo entre otras, las siguientes ventajas [HB95, NPT92]: proporcionan una base para un diseño software elegante; reducen la probabilidad de errores u omisiones, ya que proporcionan una mejor comprensión de los requisitos del sistema; la base matemática que sustenta los métodos formales posibilita el análisis de la especificación con métodos matemáticos; y finalmente, permiten el procesamiento automático de la especificación a través de herramientas desarrolladas para tal fin.

  El resultado de esta primera fase debería ser una especificación consistente y completa de los requisitos del sistema. Para lograr este objetivo, es necesario realizar un prototipado rápido del sistema especificado, aplicando una técnica de verificación y validación eficiente que proporcione la mayor información posible al usuario, de manera que éste pueda determinar con exactitud los requisitos del sistema.

  Sin embargo, el proceso de especificación es inherentemente informal, ya que tiene lugar a través del lenguaje natural, que es el lenguaje que conoce el usuario o cliente, siendo por tanto, el lenguaje utilizado para especificar los requisitos del sistema. Esta especificación se hace, generalmente, de manera ambigua con los consiguientes riesgos ante errores no detectados en esta fase.

  A pesar de ello, los usuarios o clientes siguen realizando las especificaciones de requisitos mediante el lenguaje natural, debido, fundamentalmente, al carácter informal de esta fase, por lo que será necesario proporcionar algún mecanismo que facilite la obtención de dicha especificación formal. Se puede concluir que la utilización de técnicas formales en la

fase inicial de especificación de requisitos ofrece considerables ventajas. Sin embargo, la introducción de estas técnicas encuentra dificultades debidas, principalmente, al desconocimiento del usuario o cliente de este tipo de técnicas, y a la necesidad de formalizar un proceso inherentemente informal.

Por otra parte, de los distintos tipos de técnicas formales existentes, una técnica de descripción formal orientada a propiedades se adapta mejor a la fase inicial de especificación de requisitos, ya que en ella, el usuario especifica las necesidades o requisitos del sistema como propiedades que éste debe satisfacer. Es necesario, por tanto, ofrecer al usuario una FDT orientada a propiedades próxima al lenguaje natural y al proceso inherentemente informal de esta fase.

- **Fase de diseño:**. Esta fase, cuyo punto de partida es la especificación resultante de la fase anterior, tiene como objetivo el diseño de la arquitectura del sistema, por lo que es necesario un formalismo que permita expresar los componentes del sistema, su interacción, así como las interfaces entre ellos. Es decir, es necesario utilizar una FDT constructiva.

Por tanto, parece acertado utilizar distintas técnicas en el modelo de desarrollo software. Una FDT orientada a propiedades para las primeras fases de especificación de requisitos, y una FDT constructiva para la siguiente fase de diseño de la arquitectura del sistema.

Sin embargo, la combinación de distintas FDTs es insuficiente para integrarlas en el proceso de desarrollo software, ya que, los usuarios o clientes no están acostumbrados a utilizar dichas técnicas formales, por lo que no aprovechan su potencial.

Además, la dificultad que representa para los usuarios el cambio de una especificación informal a otra utilizando FDTs, contribuye, en gran medida, a que estas técnicas no se integren en la industria del software [HB96b, Ber94]. Se hace necesario, por tanto, acercar al usuario las técnicas de descripción formal, permitiendo a éste realizar especificaciones con cierto carácter informal. En este trabajo avanzaremos por esta dirección.

3.2 Modelo de Desarrollo Iterativo con Prototipado

Los modelos tradicionales del proceso de desarrollo software definen un conjunto de actividades que se van desarrollando de forma secuencial y que se pueden organizar en dos fases:

- Una primera fase –denominada *fase de desarrollo*– que incluye las actividades de especificación y análisis de requisitos, diseño, implementación y prueba; caracterizada por un cambio continuo del sistema hasta que se consigue superar satisfactoriamente la etapa de pruebas.

- Una segunda fase –denominada *fase de mantenimiento*– que comienza cuando el sistema se entrega al cliente o usuario y se pone en funcionamiento. Actualmente, esta fase está muy poco formalizada, y su principal objetivo es corregir errores no detectados durante el proceso

de pruebas, así como realizar todas las modificaciones y cambios necesarios para que el sistema pueda seguir siendo útil.

En las últimas décadas, la rápida implantación de los sistemas software en todos los ámbitos de nuestra sociedad hace que el número de operaciones de mantenimiento crezca sin cesar. La experiencia muestra que la ejecución de una tarea de mantenimiento puede suponer la realización de actividades no sólo de implementación y prueba, sino también de diseño, o incluso de especificación y análisis de requisitos; convirtiéndose así –la *fase de mantenimiento*– en una *fase de desarrollo* propiamente dicha, cuya principal peculiaridad reside en que el punto de partida es un sistema desarrollado previamente que se encuentra en operación [Dav93].

Con estos condicionantes, parece más adecuado aplicar un modelo de desarrollo iterativo que no haga distinción explícita entre fases de desarrollo y mantenimiento. En el modelo propuesto en [LG97] (ver figura 3.1) las actividades de evolución del sistema se entrelazan con el proceso de desarrollo, y continúan incluso después de haber entregado la primera versión del sistema al usuario –en contraposición con los modelos tradicionales de ciclo de vida, que asumen que los requisitos pueden determinarse correctamente al inicio del proceso de desarrollo–.

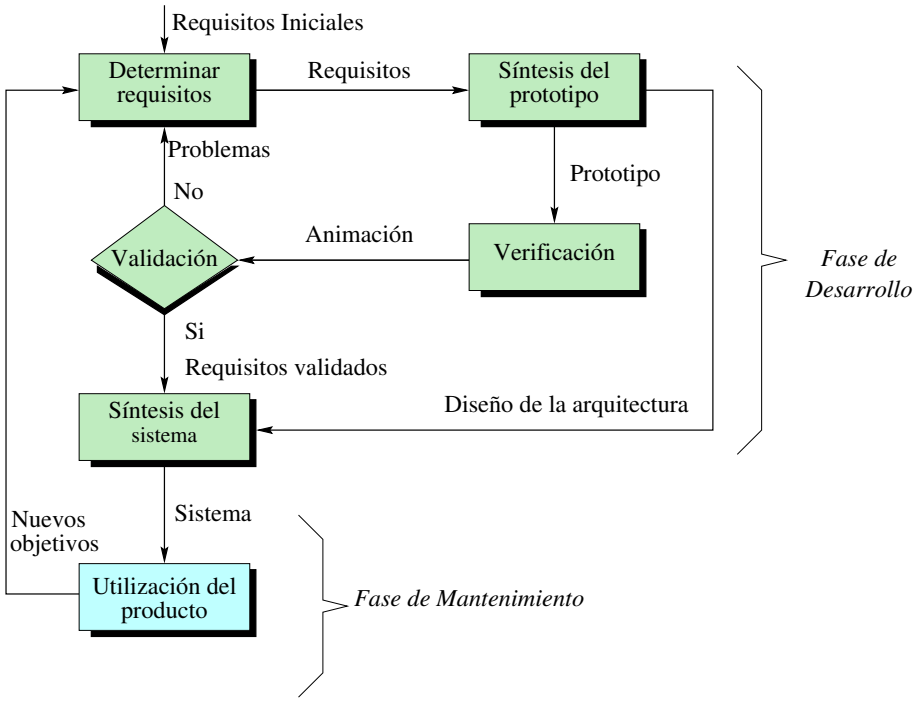


Figura 3.1: Ciclo de vida iterativo con prototipado.

Este modelo de desarrollo distingue dos tipos de cambios o evoluciones del sistema: cambios sobre el prototipo, que se corresponderían con los cambios de la primera fase de desarrollo; y cambios sobre el producto entregado al cliente. En cualquiera de los dos casos, tal y como se explicó anteriormente, es necesario una nueva iteración en el modelo de desarrollo software.

3.3 Formalización del Modelo

El empleo de varios tipos de FDTs parece estar en contradicción con el hecho de que cada cambio –independientemente de la fase en la que se produzca– supone una nueva iteración del modelo de desarrollo propuesto. Sin embargo, se ha optado por utilizar una FDT diferente en cada fase de desarrollo. Esta decisión, justificada en la sección 3.1, está reforzada por la distinta naturaleza de los cambios o evoluciones del sistema, dependiendo de la fase de desarrollo en la que estos se producen. En una primera fase, se producirán numerosos cambios, producto de ir añadiendo iterativamente y de manera incremental cada uno de los requisitos iniciales del sistema. A continuación, se detectarán inconsistencias sobre los requisitos iniciales, lo que supondrá la incursión de nuevos requisitos que no habían sido detectados previamente. Este ciclo se repetirá, amortiguándose en cada iteración el número de cambios detectados, hasta que, finalmente, se obtenga un producto sujeto únicamente a cambios esporádicos (ver figura 3.2).

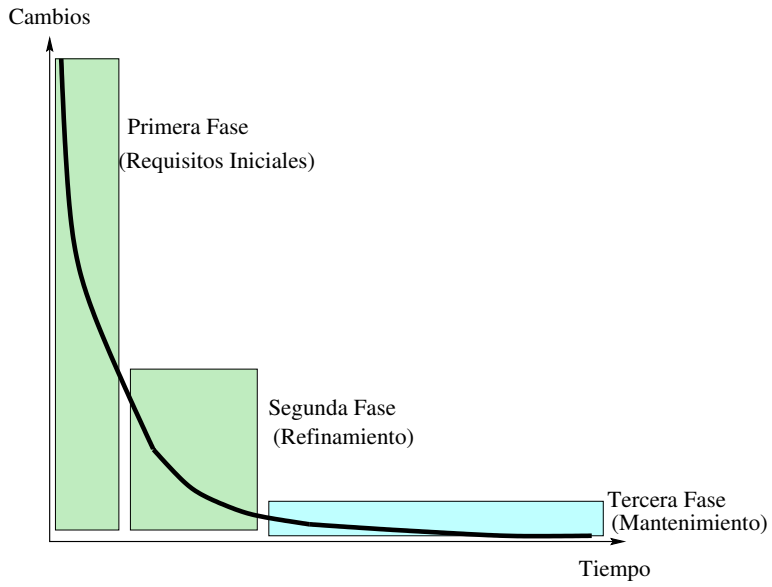


Figura 3.2: Evolución de un producto software.

En este trabajo, se propone para la primera fase –*objetivos iniciales*– modelar el sistema mediante una máquina de estados finitos, que se ha denominado MUS. MUS, que se define formalmente en el capítulo 4, incluye el concepto de subespecificación, cuyo objetivo es formalizar el carácter informal e incompleto del proceso de especificación. El concepto de subespecificación permite dotar a MUS de una mayor expresividad que los modelos de estados tradicionales.

Para la segunda fase –*refinamiento*– se propone utilizar una técnica de descripción formal constructiva, en la que poder expresar la estructura del sistema. La FDT elegida para esta fase es E-LOTOS, y su elección se basa en dos razones fundamentales:

- Por una parte, aprovechar los conocimientos y experiencias adquiridas sobre dicha FDT, integrando en la metodología propuesta el entorno transformacional LIRA [PA95, GS99], orientado a la construcción de especificaciones E-LOTOS mediante refinamientos sucesivos.

- Y por otra, su fácil integración en la metodología propuesta. Para ello, basta con traducir una especificación según MUS a la FDT constructiva elegida. En el caso de E-LOTOS esta traducción es inmediata.

El trabajo de la presente tesis se centra en la primera fase –*objetivos iniciales*– obteniendo finalmente una especificación del sistema en E-LOTOS –denominada arquitectura inicial del sistema–, que sirve de entrada al proceso de refinamiento del entorno transformacional LIRA. La metodología SCTL-MUS desarrollada en este trabajo, propone la utilización de una lógica temporal, denominada SCTL, como FDT orientada a propiedades para la fase inicial de especificación y análisis de requisitos. A continuación, se citan las principales características de SCTL, que se define formalmente en el capítulo 5.

- En primer lugar, es una lógica temporal simple, con sólo tres operadores temporales cercanos a cualquier cliente o diseñador: *A la vez, antes y después* (lógica temporal simple).
- En segundo lugar, y con el objetivo de facilitar al cliente la utilización de dicha lógica, se propone acercar dicha lógica al lenguaje natural, el cual está basado en el lenguaje causal, donde una premisa causa una consecuencia (lógica causal).
- La combinación de la lógica temporal y una lógica causal permite expresar proposiciones donde una premisa "causa" una consecuencia en un momento determinado por un operador temporal. El cliente especificará los requisitos según la siguiente proposición:

  Si es posible... (Premisa), entonces (A la vez, antes o después) debe ser posible... (Consecuencia).
- Finalmente, se enriquece la expresividad de SCTL con la introducción de un nuevo grado de verdad –denominado *subespecificado*–, cuyo principal objetivo es acercarse al carácter incompleto e informal del proceso de especificación. Un elemento subespecificado puede evolucionar hacia un grado de verdad especificado: *verdadero* o *falso*.

Una vez elegida una FDT orientada a propiedades para la primera fase de desarrollo, es posible interpretar MUS –la máquina de estados finitos con la que se modela el sistema– como un paso intermedio entre SCTL y una FDT constructiva (E-LOTOS), integrando así las ventajas de los dos tipos de FDTs en el modelo de desarrollo software. Sin embargo, la elección de MUS se basa también en otras razones que se resumen a continuación:

- Uno de los objetivos de la presente tesis es la integración de distintos tipos de técnicas de descripción formal, así como de los distintos tipos de técnicas de verificación. Por ello, además de dotar de una base matemática sólida a SCTL que permita realizar verificación mediante la técnica de demostración de teoremas (ver capítulo 7), la metodología SCTL-MUS proporciona una técnica de verificación basada en *model checking* (ver capítulo 8).
- La demostración de teoremas no se adapta al proceso de especificación incremental en el que se basa el modelo de desarrollo software propuesto. Esto es debido, fundamentalmente,

a que no es posible realizar transformaciones en las especificaciones mediante dicha técnica de verificación, salvo el añadir mediante un operador lógico una nueva fórmula a la especificación.

Esto no permite aprovechar el concepto de subespecificación, mediante el que es posible transformar las especificaciones sin alterar los requisitos especificados, sin más que perder la subespecificación existente.

- MUS permite la verificación de requisitos SCTL, ya que éstos especifican restricciones temporales entre los eventos posibles del sistema; y dichas restricciones quedan totalmente reflejadas en un modelo de estados.
- Una especificación en SCTL no permite observar el comportamiento del sistema especificado, por lo que MUS puede interpretarse como una representación gráfica de SCTL, que permite mostrar los aspectos fundamentales del comportamiento del sistema especificado, así como sintetizar prototipos.
- Un modelo de estados se adapta al proceso incremental de la fase de especificación de requisitos, permitiendo sintetizar incrementalmente –añadiendo nuevos estados al prototipo– el conjunto de requisitos especificados por el usuario.

Por tanto, MUS proporciona un punto intermedio entre los dos tipos de técnicas de descripción formal utilizadas en el proceso de desarrollo, y un método de verificación de propiedades o requisitos expresados en SCTL. Este método –basado en *model-checking*– es una alternativa a la verificación mediante demostración de teoremas, si bien en la metodología SCTL-MUS también se proporciona la base matemática necesaria para aplicar esta última técnica de verificación.

La figura 3.3 muestra el modelo de desarrollo propuesto, incluyendo las técnicas formales utilizadas en la metodología SCTL-MUS, que se centra en la primera fase –*objetivos iniciales*– y aborda la última fase –*mantenimiento*– a nivel de especificación de requisitos. La fase intermedia –*refinamiento*– forma parte de un trabajo de investigación complementario que se centra en la FDT E-LOTOS [PA95, GS99].

A continuación, se enumeran los distintos pasos que tienen lugar en una iteración de dicho modelo en su primera fase –*objetivos iniciales*–:

1. Especificación de un nuevo requisito SCTL.
2. Traducción del requisito especificado en SCTL a su correspondiente modelo de estados MUS, sobre el que se puede observar el comportamiento de dicho requisito, así como verificar propiedades o nuevos requisitos SCTL.
3. Verificación del nuevo requisito especificado sobre el prototipo actual del sistema. Si el requisito es inconsistente con el prototipo actual, es necesario reformular los requisitos actuales del sistema, o modificar el nuevo requisito especificado. Si dicho requisito ya se satisface en el prototipo actual, la especificación del nuevo requisito es redundante, por lo que el sistema no evoluciona. En caso contrario, ir al siguiente paso.

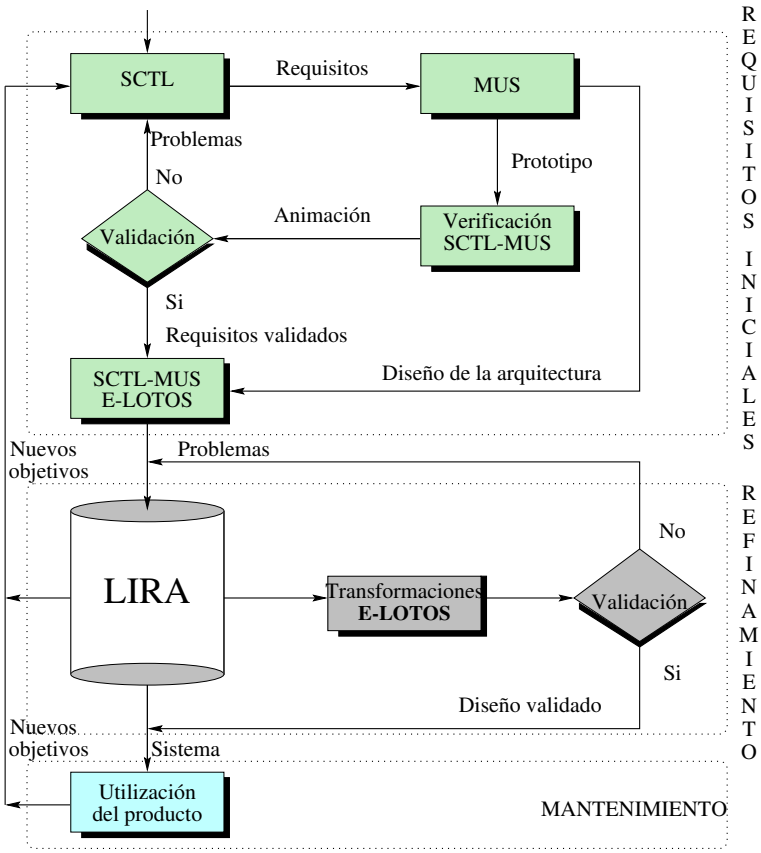


Figura 3.3: Modelo de desarrollo software propuesto.

4. Síntesis incremental. El nuevo requisito se añade al prototipo actual del sistema, sintetizando un nuevo prototipo que satisface la totalidad de requisitos especificados.

5. Si el prototipo satisface las expectativas del usuario, pasar a la siguiente fase, traduciendo la especificación SCTL-MUS a E-LOTOS.

Capítulo 4

Modelo de Estados Subespecificados: MUS

4.1 Introducción a los Modelos de Estados

Los diagramas de estados clásicos, que modelan el comportamiento de un sistema, se basan en el concepto matemático de grafo [FG94, GT96]. Un grafo, en general, consta de una serie de nodos o estados, y un conjunto de arcos uniendo dichos nodos. Si los arcos son dirigidos, es decir, especifican un estado origen y un estado destino, se dice que el grafo es dirigido.

El comportamiento de un sistema modelado mediante un grafo se basa en la evolución de un estado a otro, de manera que de un estado sólo se puede evolucionar a otro estado si entre ellos existe un arco dirigido hacia este último. El sistema evoluciona de un estado a otro cuando se produce una acción o comportamiento observable, lo que conlleva etiquetar cada arco con una acción, indicando el estado al cual evolucionaría el sistema si en el estado origen se produce el comportamiento que representa dicha acción.

Según los modelos de estados clásicos descritos, el sistema modelado se compone de un conjunto de estados, un conjunto de arcos y un conjunto de acciones. De dicho conjunto de estados existe un estado inicial que es aquél en el que se encuentra inicialmente el sistema, y a partir del cual puede evolucionar a nuevos estados; según los arcos existentes y a través de las acciones asociadas a cada arco.

Por tanto, cada sistema se caracteriza por la especificación de tres conjuntos:

1. El **conjunto de estados** del sistema.
2. El conjunto de posibles evoluciones de un estado a otros o **conjunto de arcos**.
3. El **conjunto de acciones** asociado a cada uno de los arcos especificados.

Dado un conjunto universal de estados, un conjunto universal de acciones y el conjunto total de arcos que pueden existir entre cada uno de los estados del conjunto universal, la especificación

del sistema se reduce a especificar si cada uno de los elementos de los tres conjuntos, estados, arcos y acciones, existe o no en el sistema especificado. Es decir, la especificación se reduce a asignar un valor *verdadero* o *falso* a cada uno de los elementos de dicho conjunto.

Si entre dos estados del sistema (estados con el valor *verdadero*) existe un arco, se asignará a dicho arco el valor *verdadero*. De la misma manera, al conjunto de acciones a través de las que se puede evolucionar mediante dicho arco *verdadero*, se les asignará igualmente el valor *verdadero*.

Naturalmente, una especificación en estos términos no está al alcance del diseñador o cliente:

- En primer lugar, el diseñador no tiene por qué concebir el sistema como un modelo de estados, si bien dicho modelo puede mostrar el comportamiento observable del mismo.

- En segundo lugar, el diseñador no conoce el comportamiento del sistema que desea especificar, sino sus características deseables. Es decir, el conjunto de propiedades que dicho sistema debe satisfacer.

- En tercer lugar, dicha especificación no puede plasmarse de una manera formal de manera que pueda razonarse sobre ella.

Por tanto, para obtener la especificación de un sistema según un modelo de estados, es necesario proporcionar al diseñador una interfaz previa, que le permita: especificar formalmente las propiedades del sistema deseado; y acercarse al lenguaje que conoce –el lenguaje natural–. En la metodología SCTL-MUS, dicha interfaz es la lógica SCTL, a partir de la cual se obtienen especificaciones según los modelos de estados MUS.

Por otra parte, los modelos de estados clásicos descritos anteriormente, requieren poder obtener toda la información del sistema especificado, ya que cada uno de los elementos del mismo sólo puede tener dos valores: *verdadero* o *falso*. Sin embargo, en las primeras etapas del diseño, el sistema se va construyendo a base de refinamientos sucesivos, de manera que si un estado, arco o acción, no se especifica, no significa que no deba o no pueda existir en un futuro. En los modelos de estados clásicos, si un arco (estado o acción) no existe (no es *verdadero*) significa que es *falso*, y si se verifica alguna propiedad que involucra dicho arco (estado o acción), el resultado de la verificación es el mismo que si el usuario hubiera especificado explícitamente que dicho arco (estado o acción) no existe en el sistema.

Es necesario, por tanto, enriquecer el modelo de estados de manera que cada uno de sus elementos pueda adoptar un nuevo valor –denominado **subespecificado**– en el que todos los elementos del modelo se encuentran al inicio del diseño, y desde el que pueden evolucionar a los valores *verdadero* o *falso*, a medida que el usuario refina la especificación. Además, la subespecificación permite enriquecer los grados de satisfacción de una propiedad, ya que si la verificación involucra un elemento subespecificado, es posible devolver un valor relacionado con el grado de subespecificación de dicha propiedad, que dependerá, en general, de los elementos subespecificados que afectan a la misma (ver parte III).

En la siguiente sección se definen formalmente las características de los grafos MUS. Es posible obtener un modelo de estados MUS correspondiente a cada requisito SCTL especificado. En el capítulo 5 se define formalmente SCTL, y en el capítulo 6 se describe un algoritmo que traduce un requisito SCTL a una especificación según MUS.

4.2 Definición de MUS

Básicamente, MUS se corresponde con un grafo dirigido en el que se ha añadido el concepto de subespecificación como un tercer valor de los arcos del mismo. Por tanto, entre dos nodos de un grafo MUS puede existir un arco (1), no existir (0), o estar subespecificado ($\frac{1}{2}$).

Un sistema se modela según un grafo MUS –denotado por $\mathcal{M}$– mediante un conjunto de estados –$\mathcal{E}_{\mathcal{M}} \triangleq \{E_1, E_2, ..., E_n\}$– que representa la evolución temporal del sistema modelado. El modelo evoluciona de un estado a otro cuando se produce un suceso o comportamiento observable denominado acción.

Sea $\Lambda \triangleq \{a_1, a_2, ..., a_m\}$ el conjunto de acciones, y sea $E_j \in \mathcal{E}_{\mathcal{M}}$ un estado del grafo MUS $\mathcal{M}$. Es posible especificar cada acción $a_i \in \Lambda$ en dicho estado –denotado por $E_j[a_i]$– como un valor $\psi \in \Psi \triangleq \{falso,\ subespecificado,\ verdadero\} \triangleq \{false, unspecified, true\} \triangleq \{0, \frac{1}{2}, 1\}$.

Un grafo MUS $\mathcal{M}$ se identifica por su estado inicial $E_0 \in \mathcal{E}_{\mathcal{M}}$, a partir del cual es posible recorrer cada uno de los estados a los que puede evolucionar el sistema modelado. A continuación, se enumeran los componentes que caracterizan un estado $E_j \in \mathcal{E}_{\mathcal{M}}$ de un grafo MUS $\mathcal{M}$:

1. Un conjunto de acciones **posibles** en dicho estado.

   $$\{a_i \in \Lambda : E_j[a_i] = 1\}$$

   Toda acción posible a_i de un estado E_j puede tener asociado un estado de **evolución**. Dicho estado –denotado por $E^{\{i,j\}}$– representa el estado al cual se evoluciona desde el estado E_j si se produce el comportamiento representado por la acción a_i. Si no se especifica el estado de evolución $E^{\{i,j\}}$, se dice que dicho estado está subespecificado –$E^{\{i,j\}} = \frac{1}{2}$–.

2. Un conjunto de acciones **no posibles** en dicho estado.

   $$\{a_i \in \Lambda : E_j[a_i] = 0\}$$

   El sistema no puede evolucionar a través de las acciones no posibles –$E^{\{i,j\}} = \emptyset$–.

3. Un conjunto de acciones que no pertenecen a ninguno de los dos conjuntos anteriores. Dichas acciones se denominan **subespecificadas**, y pueden ser especificadas como posibles o no posibles, pasando a formar parte de cualquiera de los dos conjuntos anteriores.

   $$\{a_i \in \Lambda : E_j[a_i] = \tfrac{1}{2}\}$$

   Toda acción a_i subespecificada en un estado E_j puede tener asociado un estado de **evolución potencial**. Dicho estado –denotado por $E^{\{i,j\}}$– representa el estado al cual se evolucionaría

desde el estado E_j si la acción a_i se especificara como posible –perdiendo así su subespecificación–, y se produjera el comportamiento representado por dicha acción. Si no se especifica el estado de evolución potencial $E^{\{i,j\}}$, se dice que dicho estado está subespecificado –$E^{\{i,j\}} = \frac{1}{2}$–.

4. Un conjunto de estados inmediatamente anteriores, es decir, aquellos desde los cuales se puede evolucionar al estado E_j.

$$\{E_k \in \mathcal{E}_{\mathcal{M}} : \exists a_h \in \Lambda,\ E_k[a_h] \neq 0 \ / \ E^{\{h,k\}} = E_j\}$$

Definición 4.1. Se define un **Estado de Parada** como un estado en el que todas las acciones se especifican como no posibles. $\forall a_i \in \Lambda, E_j \in \mathcal{E}_{\mathcal{M}},\ E_j[a_i] = 0$. El sistema no puede evolucionar en un estado de parada.

4.3 Representación Matricial de MUS

4.3.1 Introducción

Un grafo está compuesto por un conjunto de estados $\mathcal{E}$ (nodos, vértices) conectados mediante arcos ($\mathcal{A}$). Cada arco se asocia con dos estados, es decir, se establece una correspondencia entre cada arco y un par (que puede ser ordenado) de estados.

Definición 4.2. Un **Grafo** [GT96, RW92] $\mathcal{G} = (\mathcal{E}_{\mathcal{G}}, \mathcal{A})$ consta, por definición, de un conjunto no vacío $\mathcal{E}_{\mathcal{G}}$ llamado **conjunto de estados del grafo**; y de un conjunto $\mathcal{A}$, llamado **conjunto de arcos**, de subconjuntos de $\mathcal{E}_{\mathcal{G}}$ de dos elementos.

Dos estados que forman un arco se denominan **adyacentes**. Si un arco se corresponde con un par ordenado de estados se dice que el arco es **dirigido**, en caso contrario, se dice que el arco es **no-dirigido**. En un arco dirigido, el primer estado se denomina **origen** y el segundo se denomina **destino**.

Un grafo en el que todos los arcos son dirigidos se denomina **grafo dirigido**. Un grafo dirigido en el que no existen dos arcos paralelos (dos arcos que unen el mismo par ordenado de estados) se denomina **grafo simple**.

4.3.2 Definición de Grafo Subespecificado

Definición 4.3. Dado $\mathcal{E}$ un conjunto finito de estados ($\sharp\mathcal{E} = n$)[1], dado el conjunto $\mathcal{A}$ de todos los subconjuntos de $\mathcal{E}$ de dos elementos, $\{\mathcal{A}\} \triangleq \{a = (E_1, E_2) \in \mathcal{A} : E_1, E_2 \in \mathcal{E}\}$ ($\sharp\mathcal{A} = n^2$), y dado el conjunto $\{\Psi\} = \{\psi \in \Psi : \psi \in (0, \frac{1}{2}, 1)\}$, se define el **Conjunto de Arcos Tipados** $\{\mathcal{A}_t\} \triangleq \{a_t = (a, \psi) \in \mathcal{A}_t : a \in \mathcal{A}, \psi \in \Psi\}$ ($\sharp\mathcal{A}_t = 3n^2$), en el que a cada elemento de $\mathcal{A}$ se le asocia un elemento de Ψ.

[1] $\sharp\mathcal{E}$: Cardinal o número de elementos del conjunto $\mathcal{E}$.

Los elementos del conjunto de arcos tipados se denominan arcos **posibles** ($\psi = 1$), **no posibles** ($\psi = 0$), y **subespecificados** ($\psi = \frac{1}{2}$).

Definición 4.4. Se define un **Grafo Subespecificado** como un conjunto $\mathcal{G} = (\mathcal{E}_\mathcal{G}, \mathcal{T})$, donde el conjunto $\mathcal{T}$ es cualquier subconjunto de $\mathcal{A}_t$ que cumple las siguientes propiedades:

(i) Todo estado es origen de n arcos tipados. $\forall E_i \in \mathcal{E}_\mathcal{G}, \exists \{t_{i1}, ..., t_{in} \in \mathcal{T}/t_{ij} = (E_i, E_{i_j}, \psi_{i_j})\}$.

(ii) No existen dos arcos paralelos que unen el mismo par ordenado de estados.

(iii) Un estado sólo puede ser origen de un arco posible:

Sean $t_i = (E_{i1}, E_{i2}, \psi_i),\ t_j = (E_{j1}, E_{j2}, \psi_j)$ dos arcos tipados del conjunto $\mathcal{T}$, entonces:

$$\left.\begin{array}{l} E_{i1} = E_{j1} \\ E_{i2} = E_{j2} \end{array}\right\} \Rightarrow\ t_i = t_j \qquad \left.\begin{array}{l} E_{i1} = E_{j1} \\ \psi_i = \psi_j = 1 \end{array}\right\} \Rightarrow\ t_i = t_j$$

Corolario 4.1. En un grafo subespecificado, $\sharp\mathcal{T} = n^2$.

Corolario 4.2. Un grafo subespecificado es un grafo simple dirigido.

Definición 4.5. Se define un **Grafo Subespecificado de Grado** m como un conjunto de m grafos subespecificados que comparten el mismo conjunto $\mathcal{E}$ de estados. $\mathcal{G}^m = (\mathcal{G}_1, \mathcal{G}_2, ..., \mathcal{G}_m)$, donde cada $\mathcal{G}_i$ es un grafo subespecificado de la forma $\mathcal{G}_i = (\mathcal{E}, \mathcal{T}_i)$.

Corolario 4.3. Un grafo MUS se corresponde con un grafo subespecificado de grado $m = \sharp(\Lambda)$.

4.3.3 Representación Matricial de un Grafo Subespecificado

En esta sección se presenta un método de representación de grafos subespecificados utilizando matrices. Este método de representación tiene varias ventajas: las matrices pueden ser almacenadas fácilmente en un ordenador, y las operaciones estudiadas en el álgebra de matrices pueden aplicarse para el cálculo de caminos, ciclos y otras características propias de los grafos subespecificados que se verán en las próximas secciones y capítulos.

Es necesario asumir alguna clase de orden en los estados de un grafo, en el sentido de poder referirse a un estado como el primero, el segundo, etc. La matriz de representación dependerá del orden de los estados.

Definición 4.6. Dado un grafo subespecificado $\mathcal{G} = (\mathcal{E}_\mathcal{G}, \mathcal{T})$, donde $\mathcal{E}_\mathcal{G} = \{E_1, E_2, ..., E_n\}$, siendo E_1 el primer estado y E_n el último, se define una matriz $\mathcal{D}_\mathcal{G}$ de tamaño $n \times n$ denominada **Matriz de Adyacencia** del grafo $\mathcal{G}$, cuyos elementos d_{ij} están dados por:

$$d_{ij} = \psi_{ij} = \begin{cases} 1, & \text{si } (E_1, E_j, 1) \in \mathcal{T} \\ \frac{1}{2}, & \text{si } (E_1, E_j, \frac{1}{2}) \in \mathcal{T} \\ 0, & \text{si } (E_1, E_j, 0) \in \mathcal{T} \end{cases}$$

Según la definición 4.4, en una fila i como mucho podrá haber un elemento a 1, y no podrán existir dos arcos tipados uniendo el mismo par ordenado de estados, por lo que la matriz de adyacencia define completamente un grafo subespecificado.

El estado E_i de un grafo subespecificado está totalmente definido por la fila y columna i-ésima de su matriz de adyacencia. La fila i-ésima representa los arcos tipados que tienen como origen el estado E_i. Similarmente, la columna i-ésima representa los arcos tipados que tienen como destino el estado E_i.

Un grafo subespecificado de n estados puede representarse, por tanto, mediante una matriz bidimensional $n \times n$, donde el elemento d_{ij} representa el tipo de arco dirigido que existe entre el estado i (origen) y el estado j (destino).

La tabla 4.1 muestra una clasificación de los tipos de un estado E_i de un grafo subespecificado, en función de los siguientes criterios: grado de especificación, origen de arcos posibles y destino de arcos posibles.

Grado de Especificación	
Especificado	Si $\forall E_j$ se tiene $d_{ij} \neq \frac{1}{2}$
Parcialmente Especificado o Parcialmente Subespecificado	Si $\exists\ E_j, E_k/$ $d_{ij} = \frac{1}{2}$ y $d_{ik} \neq \frac{1}{2}$
Subespecificado	Si $\forall E_j$ se tiene $d_{ij} = \frac{1}{2}$

Origen de arcos posibles	
De evolución	Si $\exists E_j / d_{ij} = 1$
De parada	Si $\forall E_j,\ d_{ij} = 0$

Destino de arcos posibles	
De inicio	Si $\nexists E_j / d_{ji} = 1$
Aislado	Si $\forall E_j,\ d_{ji} = 0$

Tabla 4.1: Posibles tipos del estado E_i de un grafo subespecificado.

Para poder obtener una representación matricial de MUS basta con identificar tantos grafos subespecificados como número de acciones. De esta manera, es posible interpretar un grafo MUS como un grafo subespecificado de grado $m = \sharp(\Lambda)$, $\mathcal{G}^m = \{\mathcal{G}_1, ..., \mathcal{G}_m\}$, donde cada grafo $\mathcal{G}_i$ representa la evolución del sistema mediante la acción a_i. Según la definición de grafo subespecificado, en un estado existirá como mucho una única evolución hacia otro estado mediante la acción representada por dicho grafo.

Por tanto, la representación matricial de un grafo MUS de un sistema con n estados y con m acciones, se corresponde con una matriz tridimensional[2] $n \times n \times m$.

4.3.4 Operaciones sobre Grafos Subespecificados

Definición 4.7. Sea $\mathcal{G}$ un grafo subespecificado y $\mathcal{D}_\mathcal{G}$ su matriz de adyacencia. Sea $E_i \in \mathcal{E}_\mathcal{G}$ el

[2]**Notación.** $d[a_i][j][k]$: Elemento j, k de la matriz de adyacencia correspondiente al grafo subespecificado de la acción a_i.

estado i-ésimo de dicha matriz, se define el **Conjunto de Operadores Temporales Básicos Θ**, $\Theta \triangleq \{$*A la vez, Después, Antes*$\} \triangleq \{\Rightarrow, \Rightarrow \bigcirc, \Rightarrow \bigodot\}$. Cada elemento del conjunto Θ es una función que obtiene, a partir del estado E_i, un conjunto de estados relacionados con el primero.

$\bigoplus \in \Theta$		
$\mathcal{E}_{\mathcal{G}}$	$\longrightarrow$	$\{\mathcal{E}_{\mathcal{G}}\}$
E_i	$\longmapsto$	$\bigoplus(E_i)$

En el caso del operador *A la vez*, el resultado es el mismo estado E_i. En el caso del operador *Después*, el conjunto de estados son aquellos a los que se puede evolucionar desde el estado E_i a través de un arco posible –en un grafo subespecificado como mucho habrá un estado de estas características–. Finalmente, para el operador *Antes*, el conjunto de estados resultantes son aquellos desde los cuales se puede evolucionar, a través de un arco posible, al estado E_i.

$$\Rightarrow (E_i) \triangleq \{E_i\}$$
$$\Rightarrow \bigcirc(E_i) \triangleq \{E_k \in \mathcal{E}_{\mathcal{G}} : d_{ik} = 1\}$$
$$\Rightarrow \bigodot(E_i) \triangleq \{E_k \in \mathcal{E}_{\mathcal{G}} : d_{ki} = 1\}$$

La extensión de los operadores temporales básicos a grafos subespecificados de grado m (grafos MUS) es inmediata, sin más que aplicar los operadores temporales definidos a cada uno de los grafos correspondientes a cada acción, siendo el resultado la unión de todos los estados resultantes.

4.4 Ejemplos

En la figura 4.1.a se muestra un grafo MUS con las siguientes características:

- $\Lambda = \{a_1, a_2, a_3\}, \mathcal{E}_{\mathcal{M}} = \{E_0, E_1\}$.
- Existe un estado inicial E_0 desde el que se puede evolucionar al estado E_1 si se produce la acción a_1. A su vez, desde el estado E_1 se puede evolucionar de nuevo al estado E_0 si se produce la acción a_3.
- El sistema puede crecer –perder subespecificación– a través del estado E_1, ya que, en él las acciones a_1 y a_2 están subespecificadas:
  - Si la acción a_1 se especifica como posible (figura 4.1.b), el sistema no crece, ya que su estado asociado es el estado E_0 –$E^{\{1,1\}} = E_0$–.
  - Sin embargo, el estado asociado a la acción a_2 está subespecificado $E^{\{2,1\}} = \frac{1}{2}$, por lo que es posible que el sistema crezca creándose un nuevo estado. En la figura 4.1.c se muestra el nuevo grafo MUS después de especificar la acción a_2 como posible en el estado E_1, y su estado asociado E_2 con todas las acciones subespecificadas.

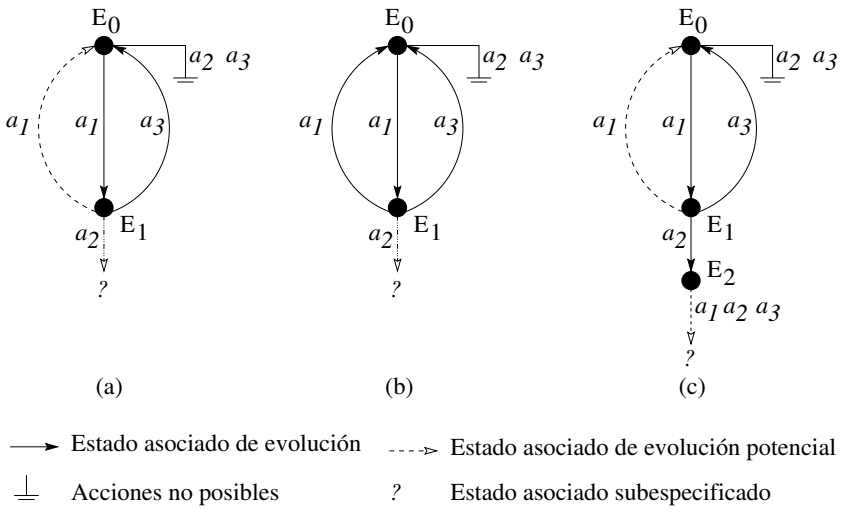


Figura 4.1: Ejemplo de MUS.

En el ejemplo 4.1[3] se muestra el tipo de cada uno de los estados de un grafo subespecificado $\mathcal{G}$ con 6 estados, cuya matriz de adyacencia $\mathcal{D}_{\mathcal{G}}$ se muestra a continuación:

$$\mathcal{D}_{\mathcal{G}} = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & \frac{1}{2} & 0 \\ 1 & \frac{1}{2} & \frac{1}{2} & \frac{1}{2} & \frac{1}{2} & \frac{1}{2} \\ 0 & 0 & 0 & \frac{1}{2} & 0 & 0 \\ \frac{1}{2} & \frac{1}{2} & \frac{1}{2} & \frac{1}{2} & \frac{1}{2} & \frac{1}{2} \end{pmatrix}$$

En el ejemplo 4.2 se muestra la representación matricial del grafo MUS de la figura 4.1.c. En dicha figura se observa que el modelo de estados consta de tres estados y de tres acciones, por lo que su matriz de representación o matriz de adyacencia, será una matriz $\mathcal{D}_{3\times3\times3}$.

En el capítulo 6 se desarrollan una serie de algoritmos orientados a obtener los grafos MUS correspondientes a los requisitos expresados mediante la lógica SCTL. En dicho capítulo se aborda de nuevo la representación matricial de los grafos MUS, explicando detalladamente la aparición de nuevos estados y acciones para poder expresar totalmente el concepto de subespecificación.

[3] f_i, c_i: Fila y columna i-ésima de la matriz $\mathcal{D}_{\mathcal{G}}$.

Ejemplo 4.1 Ejemplo de tipo de estados de un grafo subespecificado.

Estado	**Arcos Especificados**	**Tipo**
$E_1 =$	$f_1 = (1, 0, 0, 0, 0, 0)$ $c_1 = (1, 0, 1, 1, 0, \frac{1}{2})$	De evolución especificado
$E_2 =$	$f_2 = (0, 0, 0, 0, 0, 0)$ $c_2 = (0, 0, 0, \frac{1}{2}, 0, \frac{1}{2})$	De parada especificado De inicio
$E_3 =$	$f_3 = (1, 0, 0, 0, \frac{1}{2}, 0)$ $c_3 = (0, 0, 0, \frac{1}{2}, 0\frac{1}{2})$	De evolución parcialmente especificado De inicio
$E_4 =$	$f_4 = (1, \frac{1}{2}, \frac{1}{2}, \frac{1}{2}, \frac{1}{2}, \frac{1}{2})$ $c_4 = 0, 0, 0, \frac{1}{2}, \frac{1}{2}, \frac{1}{2})$	De evolución parcialmente especificado De inicio
$E_5 =$	$f_5 = (0, 0, 0, \frac{1}{2}, 0, 0)$ $c_5 = (0, 0, \frac{1}{2}, \frac{1}{2}, 0, \frac{1}{2})$	Parcialmente especificado De inicio
$E_6 =$	$f_6 = (\frac{1}{2}, \frac{1}{2}, \frac{1}{2}, \frac{1}{2}, \frac{1}{2}, \frac{1}{2})$ $c_6 = (0, 0, 0, \frac{1}{2}, 0, \frac{1}{2})$	Subespecificado De inicio

Ejemplo 4.2 Representación matricial de MUS.

a_1	$\mathbf{E}_0$	$\mathbf{E}_1$	$\mathbf{E}_2$
$\mathbf{E}_0$	0	1	0
$\mathbf{E}_1$	1	0	0
$\mathbf{E}_2$	$\frac{1}{2}$	$\frac{1}{2}$	$\frac{1}{2}$

a_2	$\mathbf{E}_0$	$\mathbf{E}_1$	$\mathbf{E}_2$
$\mathbf{E}_0$	0	0	0
$\mathbf{E}_1$	0	0	1
$\mathbf{E}_2$	$\frac{1}{2}$	$\frac{1}{2}$	$\frac{1}{2}$

a_3	$\mathbf{E}_0$	$\mathbf{E}_1$	$\mathbf{E}_2$
$\mathbf{E}_0$	0	0	0
$\mathbf{E}_1$	1	0	0
$\mathbf{E}_2$	$\frac{1}{2}$	$\frac{1}{2}$	$\frac{1}{2}$

Capítulo 5

Lógica Temporal Causal Simple: SCTL

5.1 Introducción

La elección de SCTL se basa en la idea de conjugar una técnica formal con el proceso informal de la captura y especificación de requisitos. Sin embargo, la principal aportación en este sentido, no es tanto la elección de una u otra lógica, sino la introducción en su expresividad del concepto de **subespecificación**, al que se dedica esta sección. La subespecificación permite tratar con el debido rigor matemático el carácter incompleto de las especificaciones a través del lenguaje natural, formalizando dicho carácter incompleto sin eliminarlo de la especificación del usuario.

Debido al carácter informal de la fase de especificación de requisitos, y a que el usuario no especifica el sistema completamente, sino que lo hace a medida que identifica los requisitos del sistema deseado, es fundamental proporcionar una técnica de descripción formal orientada a propiedades, simple, cercana al lenguaje natural y que permita a éste especificar de manera **incremental** los requisitos del sistema deseado. Dicha especificación incremental añade una característica importante a la notación formal utilizada, ya que debe permitir, en todos los pasos intermedios de la especificación, la validación de la consistencia de los requisitos especificados, así como asegurar la satisfacción de los requisitos previamente especificados.

Por otra parte, la utilización de una técnica de especificación formal hace necesario obtener en cada una de las fases de diseño especificaciones completas, lo que supone considerar *verdadero* aquello que el usuario especifica y *falso* aquello que no ha especificado. Éste es el principal salto existente entre una especificación formal y la especificación del usuario, ya que los requisitos del sistema se especifican incrementalmente. Para formalizar este proceso incremental es necesario que un requisito no especificado no sea tratado igual que si dicho requisito se especificara como *falso* o no posible.

En este sentido se introduce el concepto de subespecificación como un tercer estado frente a la lógica *booleana* clásica, de manera que un requisito o propiedad pueda especificarse como *verdadero*, *falso* o *subespecificado*. Un requisito subespecificado equivaldría a un requisito que no se ha especificado, debido a que el usuario o cliente no realiza especificaciones completas, sino

que va refinando de manera incremental dichas especificaciones. Un requisito subespecificado podrá pasar a cualquiera de los otros dos estados (*verdadero* o *falso*), debido a dicho proceso de refinamiento en la fase de especificación de requisitos.

El concepto de subespecificación es el enlace entre el proceso informal de especificación de requisitos y el carácter formal de dicha fase obtenido mediante la utilización de SCTL. La subespecificación permite tratar especificaciones incompletas, proporcionadas por el usuario, como especificaciones completas desde el punto de vista del formalismo utilizado (SCTL).

El proceso de especificación de requisitos utilizando SCTL, parte de un sistema **totalmente subespecificado**. Dicho sistema irá perdiendo subespecificación a medida que el usuario vaya especificando requisitos. Este proceso se repite hasta que finalmente el sistema queda **totalmente especificado**. Un sistema **totalmente subespecificado** es aquél en el que el usuario no ha especificado ningún requisito, y que, por tanto, puede evolucionar en cualquier sentido, al carecer de restricciones. El concepto de sistema **totalmente especificado** se entiende de manera diferente según dos puntos de vista:

1. Desde el punto de vista del **usuario**, el sistema queda totalmente especificado cuando finaliza la fase de especificación de requisitos, ya que, en ese momento, el sistema satisface todos los requisitos deseados.

2. Sin embargo, la **especificación formal** de un sistema debe ser completa y por tanto, no puede contener partes subespecificadas. Es decir, una vez que el usuario da por finalizada la fase de especificación de requisitos, es necesario introducir una nueva fase, que se ha denominado **fase de pérdida de subespecificación**. Dicha fase tiene como objetivo obtener una especificación formal completa consistente con la especificación incompleta obtenida en la fase anterior.

5.2 Definición de SCTL

Definición 5.1. Sea $\Lambda = \{a_1, a_2, ..., a_m\}$ el conjunto de acciones correspondiente a un grafo MUS $\mathcal{M}$ de n estados. Una **Fórmula SCTL** $\mathcal{F}$ se construye mediante la combinación de los siguientes elementos:

1. **Acciones** del conjunto Λ.

2. **Operadores Temporales** del conjunto $\Theta \triangleq \{\Rightarrow, \Rightarrow \bigcirc, \Rightarrow \bigodot\}$.

3. **Operadores Lógicos** del conjunto $\Gamma \triangleq \{\vee, \wedge, \neg\}$.

4. **Constantes** del conjunto $\Psi = \{0, \frac{1}{2}, 1\}$

5. **Identificadores** de requisitos SCTL[1] precedidos del símbolo $\uparrow$.

[1] Ver definición 5.2.

Definición 5.2. Se define un **Requisito SCTL** $\mathcal{R}$ como cualquier fórmula SCTL $\mathcal{F}$ construida según la sintaxis definida en la figura 5.1. Un requisito, en general, se compone de una premisa, un operador temporal y una consecuencia, donde la premisa y la consecuencia son, a su vez, requisitos SCTL. También pueden construirse requisitos SCTL mediante los operadores lógicos del conjunto Γ. El símbolo $\uparrow$ permite utilizar en la definición de requisitos otros requisitos ya definidos, mediante la instanciación de su identificador precedido de dicho símbolo. Esto posibilita la definición de requisitos SCTL recursivos.

Definición 5.3. Se define un **Requisito SCTL Especificado** en un estado $E_j \in \mathcal{E}_{\mathcal{M}}$ –denotado por $\mathcal{R}_{E_j}$–, como un requisito SCTL $\mathcal{R}$ que debe satisfacerse en el estado E_j del grafo MUS $\mathcal{M}$.

Definición 5.4. Dado un requisito SCTL $\mathcal{R}$ compuesto por una premisa, un operador temporal $\bigoplus \in \Theta$ y una consecuencia, se definen los **Estados de Aplicabilidad** de dicho requisito especificado en un estado $E_j \in \mathcal{E}_{\mathcal{M}}$ –denotados por $\perp(\mathcal{R}_{E_j})$–, como el conjunto de estados en los que la premisa causa la consecuencia. Dichos estados se calculan aplicando las funciones definidas para los operadores temporales básicos[2] al estado E_j en el que se especifica el requisito, $\perp(\mathcal{R}_{E_j}) \triangleq \bigoplus(E_j)$.

La extensión de SCTL mediante la definición de nuevos operadores temporales es inmediata, ya que basta con definir el conjunto de estados de aplicabilidad del nuevo operador.

Un requisito SCTL especificado –$\mathcal{R}_{E_j}$– puede especificar una acción $a_i \in \Lambda$ en cualquier estado $E_k \in \mathcal{E}_{\mathcal{M}}$ –denotado por $\mathcal{S}_{\mathcal{R}}(a_i, E_j)$– mediante la utilización de los operadores temporales básicos. Dicha especificación puede adoptar tres valores: *verdadero, falso o subespecificado*, $\mathcal{S}_{\mathcal{R}}(a_i, E_j) \triangleq \psi \in \Psi = \{0, \frac{1}{2}, 1\}$. Para especificar una acción como *no posible*, se la precede del operador $\neg$. Si una acción no se especifica en un estado, dicha acción está subespecificada, $\mathcal{S}_{\mathcal{R}}(a_i, E_j) = \frac{1}{2}$.

Definición 5.5. Se define un **Requisito Atómico** $\mathcal{R}_{at}$ como un requisito SCTL compuesto por una premisa, un operador temporal $\bigoplus \in \Theta$ y una consecuencia, donde la premisa es un elemento de Ψ y la consecuencia es una acción que puede estar precedida por el operador $\neg$. Por tanto, un requisito atómico será de la forma: $\mathcal{R}_{at} = \psi \bigoplus [\neg] a_i,\ \psi \in \Psi,\ a_i \in \Lambda$.

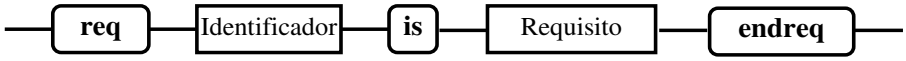
Definición 5.6. Las **Trazas de un Requisito** SCTL $\mathcal{R}$ se definen como un conjunto de requisitos SCTL $\{\mathcal{T}_R\}$, cuyos elementos se obtienen sustituyendo en el requisito $\mathcal{R}$ las expresiones $\mathcal{R}_1 \vee \mathcal{R}_2$ por $\mathcal{R}_1$, $\mathcal{R}_2$ y $\mathcal{R}_1 \wedge \mathcal{R}_2$. Por tanto, dado $\mathcal{R} = \mathcal{R}_1 \vee \mathcal{R}_2$, entonces $\{\mathcal{T}_R\} \triangleq \{\mathcal{R}_1, \mathcal{R}_2, \mathcal{R}_1 \wedge \mathcal{R}_2\}$.

Las trazas de un requisito permiten sustituir, en la fase de diseño, un requisito $\mathcal{R}$ por una de sus trazas, exactamente igual que si el usuario hubiera especificado dicha traza como requisito, pero teniendo en cuenta que:

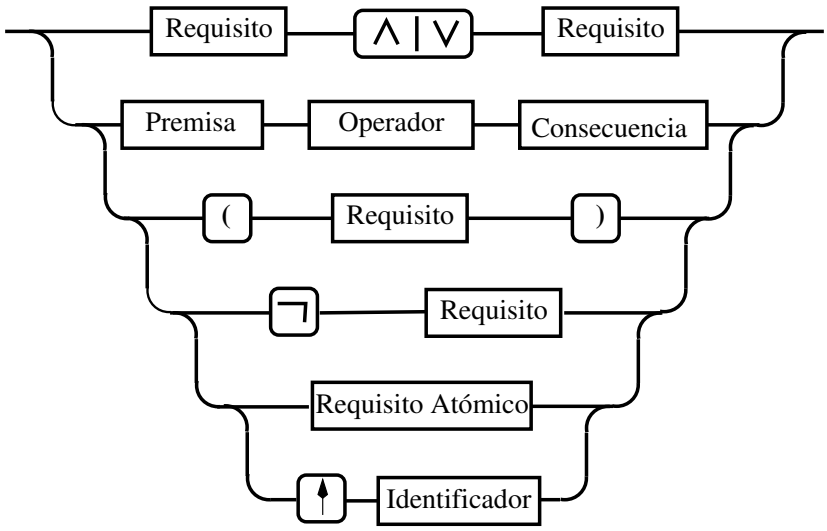
1. En las tareas de verificación de requisitos basta con verificar una de sus trazas.

[2] Ver definición 4.7.

Requisito SCTL:

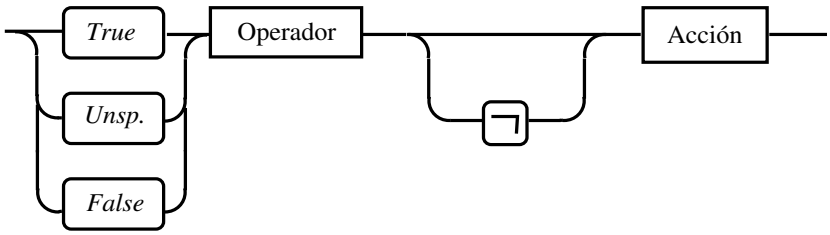


Requisito



Premisa, Consecuencia: Requisito

Requisito Atómico



Operador

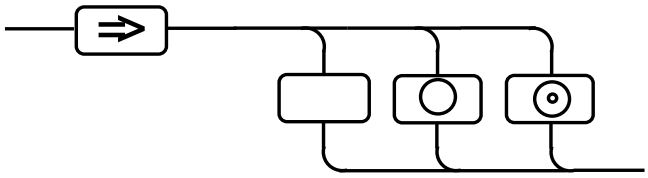


Figura 5.1: Sintaxis de la lógica SCTL.

2. Siempre que sea necesario es posible reemplazar la traza de un requisito por otra de sus trazas, sin alterar las especificaciones del usuario.

3. En la fase final del diseño, se sustituirá, en el documento que refleje los requisitos implementados, el requisito $\mathcal{R}$ por la traza correspondiente (la que ha sido utilizada durante el diseño); si bien, en el documento de especificación se mantendrá el requisito $\mathcal{R}$ para poder realizar las tareas de mantenimiento oportunas. De esta manera, en futuras modificaciones, podrá sustituirse la traza actual por una nueva sin alterar el documento de especificación del sistema.

Definición 5.7. Un requisito SCTL $\mathcal{R}$ está en **Forma Normal Positiva** sii el operador $\neg$ sólo aparece precediendo a acciones.

5.3 Ejemplos

En el ejemplo 5.1 se muestran cinco requisitos SCTL. El ejemplo 5.2 muestra las trazas del requisito SCTL $\mathcal{R}_{traz}$.

Ejemplo 5.1 Ejemplo de requisitos SCTL.

req $\mathcal{R}_1$ **is** $true \Rightarrow a_1$ **endreq**	**req** $\mathcal{R}_2$ **is** $true \Rightarrow \odot a_3$ **endreq**	**req** $\mathcal{R}_3$ **is** $\uparrow R_1 \Rightarrow \bigcirc \uparrow R_2$ **endreq**

req $\mathcal{R}_4$ **is** $\uparrow \mathcal{R}_3 \Rightarrow (\uparrow \mathcal{R}_1 \vee \neg \uparrow \mathcal{R}_4)$ **endreq**	**req** $\mathcal{R}_5$ **is** $((true \Rightarrow \bigcirc a_2) \wedge \uparrow \mathcal{R}_4) \Rightarrow (true \rightarrow \odot \neg a_1)$ **endreq**

req $\mathcal{R}_{traz}$ **is** $(true \Rightarrow a_1) \Rightarrow \odot((true \Rightarrow a_2) \vee (true \Rightarrow \bigcirc a_3))$

Ejemplo 5.2 Ejemplo de trazas de un requisito SCTL.

Trazas del Requisito $\mathcal{R}_{traz}$
$\mathcal{T}_R[1] \equiv (true \Rightarrow a_1) \Rightarrow \odot(true \Rightarrow a_2)$
$\mathcal{T}_R[2] \equiv (true \Rightarrow a_1) \Rightarrow \odot(true \Rightarrow \bigcirc a_3)$
$\mathcal{T}_R[3] \equiv (true \Rightarrow a_1) \Rightarrow \odot(true \Rightarrow a_2 \wedge (true \Rightarrow \bigcirc a_3)$

Capítulo 6

Traducción SCTL-MUS

6.1 Introducción

Una vez definido el modelo de estados MUS, mediante el que se representa el comportamiento de los sistemas especificados, y un formalismo SCTL basado en una lógica temporal causal, es necesario obtener un algoritmo que sea capaz de traducir un requisito SCTL a su correspondiente grafo MUS. Dicho algoritmo se ha denominado **algoritmo de traducción SCTL-MUS**.

El algoritmo de traducción hace uso de un conjunto de algoritmos auxiliares descritos en el apéndice B. Estos algoritmos permiten expresar los requisitos SCTL en notación inversa. Dicha notación es utilizada por la totalidad de algoritmos desarrollados en la metodología SCTL-MUS, ya que la interpretación de requisitos SCTL es inmediata a partir de su notación inversa, debido a su definición recursiva a partir de los requisitos atómicos.

Por otra parte, mediante la aplicación de los algoritmos descritos en el apéndice B, es posible obtener el conjunto de subrequisitos que forman un requisito SCTL dado. La unidad mínima de dichos subrequisitos es el requisito atómico, por lo que, para la definición del algoritmo de traducción se seguirán los siguientes pasos:

1. Definición de los grafos MUS correspondientes a los requisitos atómicos. Para ello, basta con definir tres modelos de estados diferentes, uno por cada operador temporal básico definido en SCTL.

2. Definición de un algoritmo que sea capaz de obtener el grafo MUS de un requisito SCTL atómico.

3. Obtención de los grafos MUS de un requisito no atómico, a partir de los grafos MUS de los subrequisitos que lo componen.

6.2 Revisión de la Representación Matricial de MUS

Todos los requisitos SCTL atómicos son de la forma: $\mathcal{R}_{at} = \psi \bigoplus [\neg] a_i,\ \psi \in \Psi,\ a_i \in \Lambda,\ \bigoplus \in \Theta$. De dichos requisitos atómicos, sólo se va a abordar la traducción de aquellos en los que $\psi = true$. Esta decisión se basa en la forma en la que se interpretan los requisitos SCTL: "si es posible la premisa, entonces *a la vez* (*después* o *antes*) debe ser posible la consecuencia". No tiene sentido, por tanto, abordar la traducción o verificación de requisitos cuya premisa no puede ser verdadera, como en el caso de los requisitos atómicos en los que $\psi \neq true$. En la parte III de esta memoria se revisará de nuevo esta cuestión.

Las siguientes secciones definen los grafos MUS correspondientes a los requisitos atómicos. Se muestra el grafo MUS en el caso de que la acción a_i se especifique como posible (no esté precedida del operador $\neg$[1]), y el estado en el que se especifica el requisito es el estado inicial E_0.

6.2.1 Operador A La Vez

Sea $\Lambda = \{a_1, a_2, ..., a_m\}$ el conjunto de acciones correspondientes a un grafo MUS $\mathcal{M}$ de n estados.

Un requisito SCTL atómico compuesto por el operador temporal *A la vez* permite expresar que una acción a_i debe ser posible en un estado E_0, sin especificar el estado al cual se debe evolucionar si dicha acción se produce en dicho estado –$E^{i,0} = \frac{1}{2}$–. De manera similar, permite expresar que una acción no puede ser posible en un estado. Para que este tipo de requisitos SCTL puedan representarse fielmente mediante los grafos MUS es necesario enriquecer la expresividad de su representación matricial.

Definición 6.1. Se define un **Estado subespecificado** E_{sub} como aquél en el que todas sus acciones y estados asociados están subespecificados: $\forall a_i \in \Lambda, E_{sub}[a_i] = \frac{1}{2}, E^{\{i,sub\}} = \frac{1}{2}$. Todo grafo MUS contiene un estado fijo subespecificado E_{sub} –denominado también estado de subespecificación– del que parte el sistema inicial. Dicho estado representa un grado de libertad para el sistema, ya que permite –desdoblándose en dos– la creación de nuevos estados a medida que se especifican nuevos requisitos[2]. El sistema inicial según MUS es, por tanto, un estado subespecificado –$\mathcal{E}_{\mathcal{M}} = \{E_{sub}\}$–.

A continuación, se define el significado de los elementos de la matriz de adyacencia que afectan al estado de subespecificación $E_{sub} \triangleq E_{n+1}$[3]:

- $d[a_i][j][E_{sub}],\ \forall a_i \in \Lambda, \forall E_j \in \mathcal{E}'_{\mathcal{M}}$. Indica la especificación de la acción a_i en el estado E_j:
  - **true**: La acción a_i es posible en el estado E_j, por lo que debe ser posible evolucionar desde el estado E_j a través de la acción a_i. Si no existe $E_k \in \mathcal{E}'_{\mathcal{M}}$, tal que

[1] En caso contrario, no existiría el enlace asociado a la acción a_i.
[2] Ver capítulo 9.
[3] Notación: $\mathcal{E}_{\mathcal{M}} \triangleq \mathcal{E}'_{\mathcal{M}} \bigcup E_{n+1}, \mathcal{E}'_{\mathcal{M}} = \{E_1, E_2, ..., E_n\}$.

$d[a_i][j][k] = 1$, el estado de evolución no está especificado en la fase actual de diseño del sistema. En la fase final del diseño, debe existir un estado $E_k \in \mathcal{E}'_{\mathcal{M}}$, tal que $d[a_i][j][k] = 1$. En caso contrario, debe perderse subespecificación, es decir, debe transformarse algún arco subespecificado –$d[a_i][j][k] = \frac{1}{2}$– en un arco especificado –$d[a_i][j][k] = 1$–. Si no fuera posible –$\forall E_k \in \mathcal{E}'_{\mathcal{M}}, d[a_i][j][k] = 0$– es necesario crear un nuevo estado E_h tal que $d[a_i][j][h] = 1$. En el capítulo 10 se describe la **fase de pérdida de subespecificación**, en la que se transforman los elementos (acciones, arcos y estados) subespecificados en elementos totalmente especificados (*verdaderos* o *falsos*).

- **false**: La acción a_i no es posible en el estado E_j. Es decir, desde el estado E_j no se puede evolucionar a ningún estado a través de la acción a_i. En este caso, los elementos $d[a_i][j][k], \forall E_k \in \mathcal{E}'_{\mathcal{M}}$ son redundantes y deben tomar el valor *false*, $d[a_i][j][k] = 0$.
- **unspecified**: En la fase actual del diseño, la acción a_i está subespecificada en el estado E_j –$E_j[a_i] = \frac{1}{2}$–. Si existe algún estado $E_k \in \mathcal{E}'_{\mathcal{M}}$, tal que $d[a_i][j][k] = 1$, dicho estado representa el estado de evolución potencial asociado a la acción a_i en el estado E_j –$E^{\{i,j\}} = E_k$–, que tendrá sentido en fases futuras del diseño, cuando la acción a_i se especifique en el estado E_j como posible. En ese caso, se puede considerar el arco $d[a_i][j][k] = 1$ como un **arco potencial**, ya que existirá sólo si la acción a_i se especifica como posible.

- $d[a_i][E_{sub}][k] \triangleq \frac{1}{2}$, $\forall E_k \in \mathcal{E}_{\mathcal{M}}$. Estos elementos carecen de significado.

Por tanto, el estado de subespecificación E_{sub} añade expresividad a MUS en dos sentidos:

1. En primer lugar, permite especificar que una acción a_i debe ser posible en un estado E_j –$d[a_i][j][E_{sub}] = 1$–, sin tener que especificar el estado asociado a a_i en E_j –el estado al cual se evoluciona cuando se produce dicha acción–. Esto permite representar fielmente el comportamiento especificado por los requisitos SCTL atómicos con el operador temporal *A la vez*.

2. En segundo lugar, permite especificar el estado asociado a una acción sin haber especificado previamente dicha acción (arco potencial).

Además, el estado de subespecificación permite obtener el resumen del grafo subespecificado correspondiente a cada una de las acciones, ya que la columna $d[a_i][j][E_{sub}]$, muestra la especificación –*verdadera*, *falsa* o *subespecificada*– de la acción a_i en cada estado $E_j \in \mathcal{E}'_{\mathcal{M}}$ del sistema.

Corolario 6.1. $E_j[a_i] \triangleq d[a_i][j][E_{sub}], \forall a_i \in \Lambda, E_j \in \mathcal{E}'_{\mathcal{M}}$.

La figura 6.1 muestra el grafo MUS correspondiente al requisito atómico $\mathcal{R}_{at} = true \Rightarrow a_i$. En la tabla 6.1 se muestra su representación matricial.

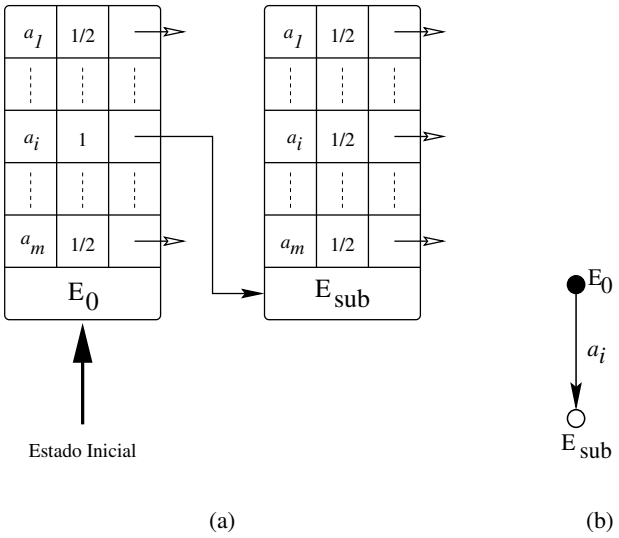


Figura 6.1: Grafo MUS del requisito atómico $\mathcal{R}_{at} = true \Rightarrow a_i$

a_i	$\mathbf{E}_0$	$\mathbf{E}_{sub}$
$\mathbf{E}_0$	$\frac{1}{2}$	1
$\mathbf{E}_{sub}$	$\frac{1}{2}$	$\frac{1}{2}$

$a_{j,j\neq i}$	$\mathbf{E}_0$	$\mathbf{E}_{sub}$
$\mathbf{E}_0$	$\frac{1}{2}$	$\frac{1}{2}$
$\mathbf{E}_{sub}$	$\frac{1}{2}$	$\frac{1}{2}$

Tabla 6.1: Representación matricial del requisito atómico $\mathcal{R}_{at} = true \Rightarrow a_i$

6.2.2 Operador Antes

Un requisito SCTL compuesto por el operador temporal básico *Antes* permite expresar que un estado es el siguiente a otro, sin especificar la acción a través de la cual se produce dicha evolución. Para poder expresar esta subespecificación según la representación matricial definida para MUS, es necesario definir una **acción de subespecificación**, denotada por $a_{sub} \triangleq a_{m+1}$[4].

A continuación, se define el significado de los elementos de la matriz de adyacencia correspondiente al grafo subespecificado de la acción de subespecificación:

- $d[a_{sub}][j][k], \forall E_j, E_k \in \mathcal{E}'_{\mathcal{M}}$. Muestra el tipo de arco dirigido desde el estado E_j al estado E_k:
  - **true**: Existe un arco posible uniendo los estados E_j y E_k. Si en la fase actual de diseño $\nexists a_i \in \Lambda'$ tal que, $d[a_i][j][k] = 1$, la acción a través de la que se produce la evolución entre los estados E_j y E_k está subespecificada. En la fase final del diseño será necesario perder subespecificación, transformando un arco subespecificado –$d[a_i][j][k] = \frac{1}{2}, a_i \in \Lambda'$–, en un arco posible –$d[a_i][j][k] = 1$–. Si no existe ninguna

[4]Notación: $\Lambda \triangleq \Lambda' \bigcup a_{m+1}, \Lambda' = \{a_1, a_2, ..., a_m\}$.

acción con dicho arco subespecificado –$\nexists a_i \in \Lambda' \ / \ d[a_i][j][k] = \frac{1}{2}$– la especificación del sistema es inconsistente.

Por tanto, si $\exists a_i \in \Lambda' \ / \ d[a_i][j][k] = 1$ entonces $d[a_{sub}][j][k] = 1$.

- **false**: No existe un arco dirigido del estado E_j hacia el estado E_k. Es decir, $\forall a_i \in \Lambda'$, $d[a_i][j][k] = 0$.
- **unspecified**: El arco dirigido que une los estados E_j y E_k está subespecificado. Es decir, $\nexists \ a_i \in \Lambda' \ / \ d[a_i][j][k] = 1$.

- $d[a_{sub}][j][E_{sub}], \forall E_j \in \mathcal{E}'_{\mathcal{M}}$. Resume el tipo de estado E_j según se indica a continuación:
  - **true**: El estado E_j es un **estado de evolución**. Por tanto, $\exists a_i \in \Lambda' \ / \ d[a_i][j][E_{sub}] = 1$.
  - **false**: El estado E_j es un **estado de parada**. Es decir, $\forall a_i \in \Lambda'$, $d[a_i][j][E_{sub}] = 0$.
  - **unspecified**: El estado E_j no es de evolución ni de parada. Si existe algún estado $E_k \in \mathcal{E}'_{\mathcal{M}} \ / \ d[a_{sub}][j][k] = 1$, el estado E_j es un **estado de evolución potencial**, ya que, existe (o existirá después de perder subespecificación) al menos un arco potencial entre él y el estado E_k, $\exists \ a_i \in \Lambda' \ / \ d[a_i][j][k] = 1$ y $d[a_i][j][E_{sub}] = \frac{1}{2}$.

- $d[a_{sub}][E_{sub}][k] \triangleq \frac{1}{2}, \forall E_k \in \mathcal{E}'_{\mathcal{M}}$. Estos elementos carecen de significado.

Por tanto, la acción de subespecificación aumenta la expresividad de MUS en dos sentidos:

1. Permite especificar un arco posible entre dos estados sin tener que especificar la acción a través de la que se produce dicha evolución. Esto permite reflejar fielmente el comportamiento de los requisitos SCTL atómicos con el operador temporal *Antes*.

2. La matriz de adyacencia correspondiente a dicha acción –denotada por $d[a_{sub}]_{n+1 \times n+1}$– muestra el comportamiento genérico del sistema, resumiendo las evoluciones existentes entre los estados del mismo (los arcos), sin especificar a través de qué acción se produce cada una de las evoluciones.

Corolario 6.2. Cada fila de la matriz de adyacencia correspondiente a la acción de subespecificación puede contener un número máximo de elementos a 1. Este número viene limitado por el número de elementos del conjunto Λ' –el número de acciones sin contar la acción de subespecificación–, ya que en cada estado sólo puede especificarse una evolución a otro estado por cada acción posible[5].

La figura 6.2 muestra el grafo MUS correspondiente al requisito atómico $\mathcal{R}_{at} = true \Rightarrow \bigodot a_i$. La tabla 6.2 muestra su representación matricial.

[5] Ver definición 4.4.

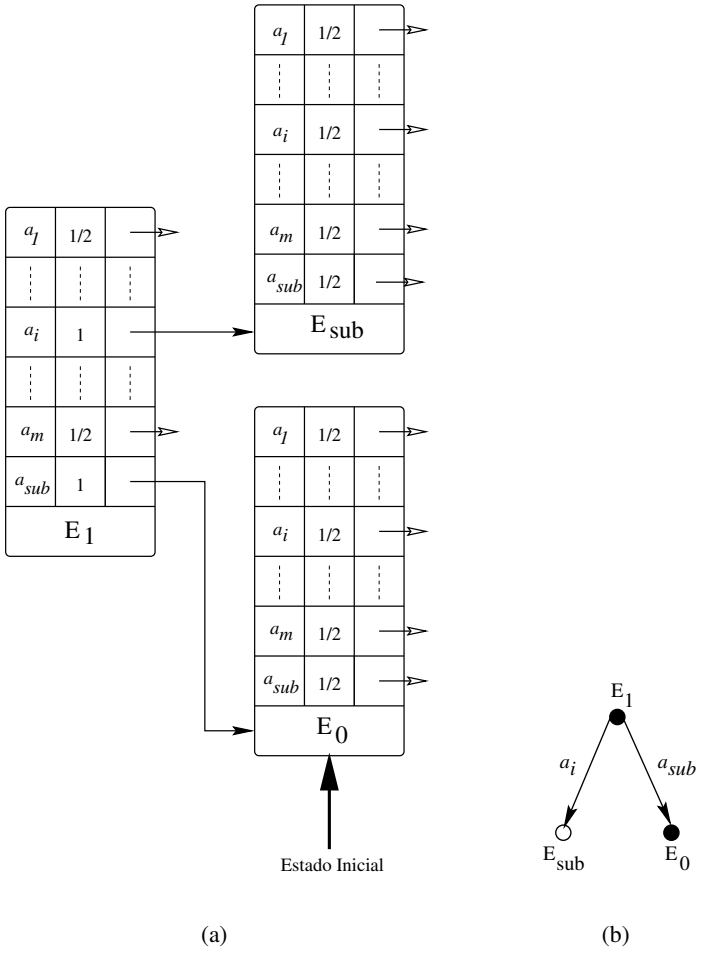


Figura 6.2: Grafo MUS del requisito atómico $\mathcal{R}_{at} = true \Rightarrow \bigodot a_i$

a_i	$\mathbf{E}_0$	$\mathbf{E}_1$	$\mathbf{E}_{sub}$
$\mathbf{E}_0$	$\frac{1}{2}$	$\frac{1}{2}$	$\frac{1}{2}$
$\mathbf{E}_1$	$\frac{1}{2}$	$\frac{1}{2}$	1
$\mathbf{E}_{sub}$	$\frac{1}{2}$	$\frac{1}{2}$	$\frac{1}{2}$

a_{sub}	$\mathbf{E}_0$	$\mathbf{E}_1$	$\mathbf{E}_{sub}$
$\mathbf{E}_0$	$\frac{1}{2}$	$\frac{1}{2}$	$\frac{1}{2}$
$\mathbf{E}_1$	1	$\frac{1}{2}$	1
$\mathbf{E}_{sub}$	$\frac{1}{2}$	$\frac{1}{2}$	$\frac{1}{2}$

$a_{j,j\neq i}$	$\mathbf{E}_0$	$\mathbf{E}_1$	$\mathbf{E}_{sub}$
$\mathbf{E}_0$	$\frac{1}{2}$	$\frac{1}{2}$	$\frac{1}{2}$
$\mathbf{E}_1$	$\frac{1}{2}$	$\frac{1}{2}$	$\frac{1}{2}$
$\mathbf{E}_{sub}$	$\frac{1}{2}$	$\frac{1}{2}$	$\frac{1}{2}$

Tabla 6.2: Representación matricial del requisito atómico $\mathcal{R}_{at} = true \Rightarrow \bigodot a_i$

6.2.3 Operador Después

Los estados de aplicabilidad de los operadores temporales *A la vez* y *Antes* dependen únicamente del estado en el que se especifica el requisito SCTL que lo contiene. En el primer caso, el estado de aplicabilidad es el mismo estado donde se formula el requisito, y en el segundo caso, los estados de aplicabilidad son el conjunto de estados inmediatamente anteriores al estado en el que se especifica la premisa. Sin embargo, en el caso del operador temporal *Después* es necesario identificar qué estados siguientes son estados de aplicabilidad, es decir, qué estados siguientes deben satisfacer la consecuencia del requisito SCTL; ya que puede existir un estado siguiente diferente por cada acción del conjunto Λ'. De lo contrario, y tal y como se definió el operador temporal *Después*, no se podrían especificar características diferentes para dos estados siguientes, con la consiguiente pérdida de expresividad.

SCTL permite seleccionar los estados de aplicabilidad del operador temporal *Después*. Para ello, basta con identificar las acciones sobre las que éste se aplica. A estas acciones se las denomina **acciones de aplicabilidad**, siendo los estados siguientes, con respecto a dichas acciones, los estados de aplicabilidad correspondientes al operador temporal *Después*. SCTL permite especificar una acción a_i de aplicabilidad mediante la especificación de un requisito de la forma: $(true \Rightarrow a_i)[\wedge Requisito...] \Rightarrow \bigcirc\, Consecuencia$. En el caso de no especificar ninguna acción de aplicabilidad –por ejemplo $\mathcal{R} = (true \Rightarrow \bigodot a_1) \Rightarrow \bigcirc\, (true \Rightarrow a_2)$–, los estados de aplicabilidad son todos los estados siguientes, independientemente de la acción a través de la que se evolucione.

Los requisitos atómicos correspondientes al operador temporal *Después* no contienen ninguna acción de aplicabilidad, ya que son de la forma: $true \Rightarrow \bigcirc[\neg]a_i$. Por tanto, sus estados de aplicabilidad son todos los estados siguientes. Dado que la representación según MUS de un requisito SCTL debe mostrar todos los aspectos posibles de su comportamiento, sería necesario obtener un grafo MUS de los requisitos atómicos con el operador temporal *Después* como el de la figura 6.3, en el que a cada acción $a_i \in \Lambda'$ se le asocia un estado potencial diferente –$E^{\{i,0\}}$– con la acción a_i especificada como posible –$d[a_i][E^{\{i,0\}}][E_{sub}] = 1$–.

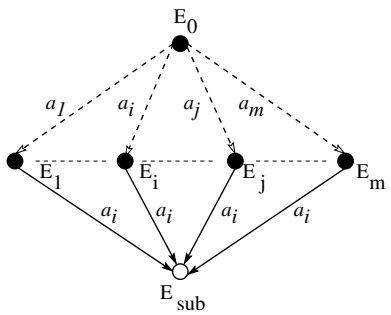


Figura 6.3: Modelo de estados subespecificados: $true \Rightarrow \bigcirc a_i$

La solución anterior presenta el problema de la aparición de un gran número de estados, que además no son estados reales del sistema, sino estados potenciales. Una solución para evitar esta explosión de estados, y poder mantener la misma expresividad es definir una nueva acción a_{res}

–denominada **acción de restricción**– cuya evolución o estado siguiente contenga las restricciones o requisitos aplicables para todos los estados siguientes a uno dado, creándose así un único estado siguiente. A continuación, se define el significado de los elementos de la matriz de adyacencia correspondiente a la acción a_{res}:

- $d[a_{res}][j][E_{sub}] = 1, E_j \in \mathcal{E}'_{\mathcal{M}}$. Existe algún estado $E_k \in \mathcal{E}'_{\mathcal{M}}$ / $d[a_{res}][j][k] = 1$. El estado E_k no es un estado real del sistema, únicamente contiene las restricciones comunes a todos los estados siguientes al estado E_j.

- $d[a_{res}][j][E_{sub}] = 0, E_j \in \mathcal{E}'_{\mathcal{M}}$. No pueden especificarse restricciones comunes a todos los estados siguientes al estado E_j.

- $d[a_{res}][j][E_{sub}] = \frac{1}{2}, E_j \in \mathcal{E}'_{\mathcal{M}}$. En otro caso.

Mediante la introducción de la acción de restricción es posible especificar restricciones para todos los estados siguientes a uno dado sin tener que crearlos. Hay que tener en cuenta que, de lo contrario, habría que crear un estado por cada acción que no estuviera especificada como no posible y que no tuviera especificado el estado siguiente –a fin de poder mostrar en el modelo de estados subespecificados las restricciones expresadas por el requisito SCTL traducido–, lo que conllevaría una explosión de estados. Sin embargo, si los estados de aplicabilidad no fueran todos los estados siguientes, no sería posible utilizar la acción de restricción para evitar la explosión de estados, y en el peor de los casos, habría que crear $m - 1$ estados nuevos ($\sharp\Lambda' = m$).

La solución propuesta engloba la obtenida mediante la definición de la acción de restricción. Dicha solución consiste en crear un único estado para reflejar las restricciones comunes a un grupo de estados siguientes. Dichos estados se denominan **estados representantes** del sistema, ya que, "representan" a un conjunto de estados reales del sistema. De esta manera, se puede eliminar la acción de restricción, ya que, ésta no es más que un caso particular del caso anterior, en el que existe un estado representante para todos los estados siguientes.

Además, es necesario poder distinguir un estado representante de otro real. Cuando un estado es real, las especificaciones sobre dicho estado se aplican directamente sobre sus acciones y enlaces. Sin embargo, cuando un estado es representante y se especifica alguno de los estados reales que representa, es necesario desdoblar dicho estado, creando un nuevo estado real sobre el que aplicar las restricciones particulares impuestas. Para indicar que un estado E_k siguiente a otro E_j a través de la acción a_i –$E^{\{i,j\}} = E_k$–, es un estado representante –denotado por $\widetilde{E_k}$–, basta con asignarle al elemento $d[a_i][E_j][E_k]$ de la matriz de adyacencia, el valor $\frac{3}{4}$ en vez del valor 1. El significado para el valor $\frac{3}{4}$ se corresponde con un nuevo grado de subespecificación que sólo puede alcanzar el valor *verdadero*. En el capítulo 7 se aborda de una manera exhaustiva la aparición de distintos grados de subespecificación.

Por tanto, a los estados representantes sólo llegan arcos con el valor $\frac{3}{4}$, mientras que a los estados reales llegan arcos con el valor 1. Para obtener los estados representantes de un grafo MUS, basta con explorar las filas de la matriz de adyacencia correspondiente a la acción de subespecifi-

cación, ya que, éstas resumen los arcos existentes en el sistema. Si $\widetilde{E_k}$ es un estado representante, entonces $\exists E_j \in \mathcal{E}'_{\mathcal{M}}$ $/d[a_{sub}][j][E_{sub}] = 1$, y $d[a_{sub}][j][k] = \frac{3}{4}$.

La figura 6.4 muestra el grafo MUS correspondiente al requisito atómico $\mathcal{R}_{at} = true \Rightarrow \bigcirc a_i$. El estado $\widetilde{E_1}$ es un estado representante de todos los estados siguientes al estado E_0. La tabla 6.3 muestra su representación matricial.

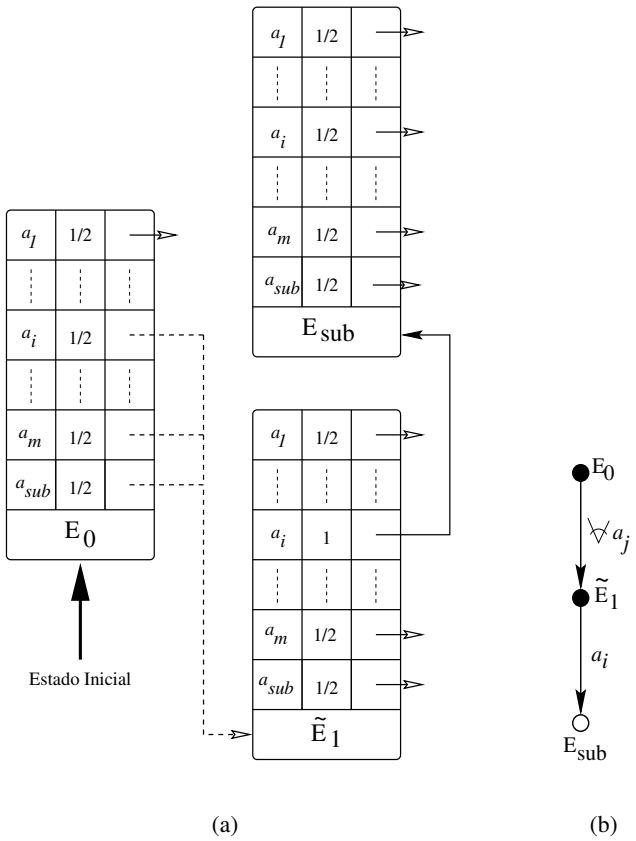


Figura 6.4: Grafo MUS del requisito atómico $\mathcal{R}_{at} = true \Rightarrow \bigcirc a_i$

a_i	$\mathbf{E}_0$	$\widetilde{\mathbf{E}_1}$	$\mathbf{E}_{sub}$
$\mathbf{E}_0$	$\frac{1}{2}$	$\frac{3}{4}$	$\frac{1}{2}$
$\widetilde{\mathbf{E}_1}$	$\frac{1}{2}$	$\frac{1}{2}$	1
$\mathbf{E}_{sub}$	$\frac{1}{2}$	$\frac{1}{2}$	$\frac{1}{2}$

a_{sub}	$\mathbf{E}_0$	$\widetilde{\mathbf{E}_1}$	$\mathbf{E}_{sub}$
$\mathbf{E}_0$	$\frac{1}{2}$	$\frac{3}{4}$	1
$\widetilde{\mathbf{E}_1}$	$\frac{1}{2}$	$\frac{1}{2}$	1
$\mathbf{E}_{sub}$	$\frac{1}{2}$	$\frac{1}{2}$	$\frac{1}{2}$

$a_{j,j\neq i}$	$\mathbf{E}_0$	$\widetilde{\mathbf{E}_1}$	$\mathbf{E}_{sub}$
$\mathbf{E}_0$	$\frac{1}{2}$	$\frac{3}{4}$	$\frac{1}{2}$
$\widetilde{\mathbf{E}_1}$	$\frac{1}{2}$	$\frac{1}{2}$	$\frac{1}{2}$
$\mathbf{E}_{sub}$	$\frac{1}{2}$	$\frac{1}{2}$	$\frac{1}{2}$

Tabla 6.3: Representación matricial del requisito atómico $\mathcal{R}_{at} = true \Rightarrow \bigcirc a_i$

6.2.4 Conclusiones de la Revisión de MUS

Se ha enriquecido la expresividad de los grafos MUS definiendo una nueva acción –de subespecificación–, un nuevo estado –de subespecificación–, y un nuevo tipo de arco –hacia estados representantes–. Con estos tres nuevos elementos es posible:

- Expresar correcta y eficientemente todas las características soportadas por SCTL y el concepto de subespecificación. El estado de subespecificación permite: especificar una acción en un estado sin tener que especificar su estado de evolución, por lo que no es necesario crear dicho estado; y especificar estados asociados a una acción sin especificar ésta como posible. La acción de subespecificación permite especificar un arco entre dos estados sin especificar la acción a través de la cual se produce dicha evolución. Finalmente, los estados representantes permiten almacenar de manera eficiente las restricciones comunes a un conjunto de estados siguientes a uno dado, sin tener que crear un estado siguiente para cada una de las acciones asociadas a dichos estados.

- Resumir y clasificar el grafo MUS. La columna de la matriz de adyacencia correspondiente al estado de subespecificación permite obtener la especificación de todas las acciones en cada estado del sistema. La matriz de adyacencia correspondiente a la acción de subespecificación permite obtener un grafo resumido con las evoluciones existentes entre los distintos estados del sistema, sin especificar a través de qué acción se producen dichas evoluciones.

Tanto la acción de subespecificación como el estado de subespecificación pueden ser tratados, a efectos de almacenamiento, operaciones y algoritmos, como el resto de acciones y estados. $\mathcal{E}_{\mathcal{M}} = \mathcal{E}'_{\mathcal{M}} \bigcup E_{sub},\ \mathcal{E}'_{\mathcal{M}} = \{E_1, E_2, ..., E_n\},\ \Lambda = \Lambda' \bigcup a_{sub},\ \Lambda' = \{a_1, a_2, ..., a_m\}$.

Por tanto, no es necesario modificar la definición de MUS, basta con tener en cuenta el significado especial de dicha acción y estado. En cuanto a los arcos etiquetados con el valor $\frac{3}{4}$ pueden ser tratados como si fueran arcos posibles, con la salvedad de modificar el estado destino –el estado representante– sólo cuando la modificación afecte a la totalidad de estados representados. En caso contrario, debe crearse un nuevo estado, que podrá ser a su vez representante, en el que aplicar las modificaciones particulares o especializaciones del antiguo estado representante.

En el ejemplo 6.1 se muestra la matriz de representación de un grafo MUS en el que se resumen los distintos tipos de especificación que soporta MUS con respecto a las acciones, estados y arcos del mismo.

6.3 Traducción de Requisitos SCTL Atómicos

6.3.1 Algoritmo de Traducción de Requisitos Atómicos I

El algoritmo 6.1 obtiene el grafo MUS de un requisito atómico SCTL. El algoritmo recibe como parámetro el requisito SCTL en notación inversa, $\overline{\mathcal{R}}_{at} = \bigoplus true[\neg]a_i, \bigoplus \in \Theta, a_i \in \Lambda'$.

Ejemplo 6.1 Tipos de especificación soportados por MUS.

a_1	$\mathbf{E}_0$	$\mathbf{E}_1$	$\widetilde{\mathbf{E}_2}$	$\mathbf{E}_{sub}$
$\mathbf{E}_0$	$\frac{1}{2}$	1	$\frac{1}{2}$	1
$\mathbf{E}_1$	$\frac{1}{2}$	$\frac{1}{2}$	$\frac{3}{4}$	1
$\widetilde{\mathbf{E}_2}$	$\frac{1}{2}$	$\frac{1}{2}$	$\frac{1}{2}$	1
$\mathbf{E}_{sub}$	$\frac{1}{2}$	$\frac{1}{2}$	$\frac{1}{2}$	$\frac{1}{2}$

a_2	$\mathbf{E}_0$	$\mathbf{E}_1$	$\widetilde{\mathbf{E}_2}$	$\mathbf{E}_{sub}$
$\mathbf{E}_0$	1	$\frac{1}{2}$	$\frac{1}{2}$	$\frac{1}{2}$
$\mathbf{E}_1$	$\frac{1}{2}$	$\frac{1}{2}$	$\frac{3}{4}$	$\frac{1}{2}$
$\widetilde{\mathbf{E}_2}$	$\frac{1}{2}$	$\frac{1}{2}$	$\frac{1}{2}$	$\frac{1}{2}$
$\mathbf{E}_{sub}$	$\frac{1}{2}$	$\frac{1}{2}$	$\frac{1}{2}$	$\frac{1}{2}$

Fila	**Significado**
$d[a_1][0]$	Acción a_1 especificada. Estado siguiente especificado.
$d[a_1][1]$	Acción a_1 especificada. Estado siguiente representante.
$d[a_1][2]$	Acción a_1 especificada. Estado siguiente subespecificado.
$d[a_2][0]$	Acción a_2 subespecificada. Estado siguiente especificado. Arco potencial.
$d[a_2][1]$	Acción a_2 subespecificada. Estado siguiente representante. Arco potencial.
$d[a_2][2]$	Acción a_2 subespecificada. Estado siguiente subespecificado.

El grafo MUS se obtiene a partir de un grafo MUS inicial –$\mathcal{M}$– que contiene únicamente un estado –$\mathcal{E}'_{\mathcal{M}} = \{E_0\}$–, donde se desea traducir el requisito atómico $\mathcal{R}_{at}$. La tabla 6.4 muestra la representación matricial del grafo MUS inicial $\mathcal{M}$.

$\forall a_i \in \Lambda$	$\mathbf{E}_0$	$\mathbf{E}_{sub}$
$\mathbf{E}_0$	$\frac{1}{2}$	$\frac{1}{2}$
$\mathbf{E}_{sub}$	$\frac{1}{2}$	$\frac{1}{2}$

Tabla 6.4: Grafo MUS inicial $\mathcal{M}$.

A continuación, se describe el comportamiento general del algoritmo 6.1.

[1] Si el operador temporal que forma el requisito es el operador *A la vez*, entonces el estado de aplicabilidad es el mismo estado E_0 donde se traduce el requisito:

- **[a]** Se añade, a la matriz de adyacencia del grafo subespecificado de la acción a_i, la especificación del requisito $\mathcal{R}_{at}$ de dicha acción en el estado E_0 –$\mathcal{S}_{\mathcal{R}_{at}}(a_i, E_0)$– . De esta manera, se especifica que la acción a_i es posible ($true \Rightarrow a_i$) o no ($true \Rightarrow \neg a_i$) en el estado E_0.
- **[b]** Si $\mathcal{R}_{at}$ especifica que a_i es posible en E_0, es necesario reflejar, mediante la acción de subespecificación, que el estado E_0 es de evolución.
- **[c]** Fin del algoritmo.

Algoritmo 6.1 Algoritmo de Traducción de Requisitos Atómicos ($\overline{\mathcal{R}}_{at} = \bigoplus true[\neg]a_i$)

[1] Si $\bigoplus = \Rightarrow$:
  [a] $d[a_i][0][E_{sub}] = \mathcal{S}_{\mathcal{R}_{at}}(a_i, E_0)$;
  [b] Si $\mathcal{S}_{\mathcal{R}_{at}}(a_i, E_0) = 1,\ d[a_{sub}][0][E_{sub}] = 1$;
  [c] Fin;
[2] Nuevo estado E_k;
[3] $d[a_i][k][E_{sub}] = \mathcal{S}_{\mathcal{R}_{at}}(a_i, E_k)$;
[4] Si $\mathcal{S}_{\mathcal{R}_{at}}(a_i, E_k) = 1,\ d[a_{sub}][k][E_{sub}] = 1$;
[5] Si $\bigoplus = \Rightarrow \bigodot$, $d[a_{sub}][k][0] = 1$;
[6] Si $\bigoplus = \Rightarrow \bigcirc$, $\forall a_l \in \Lambda,\ d[a_l][0][k] = \frac{3}{4}$;
[7] Fin;

[2] En caso contrario, es necesario crear un nuevo estado E_k donde aplicar la consecuencia del requisito. Dicho estado se corresponde con un estado anterior o un estado siguiente a E_0, en función del operador temporal del requisito $\mathcal{R}_{at}$.

[3] Se añade la especificación del requisito $\mathcal{R}_{at}$ como en el paso **[1][a]**, pero al nuevo estado –el de aplicabilidad– E_k.

[4] Se actualiza la matriz correspondiente a la acción de subespecificación, de manera similar al paso **[1][b]**.

[5] Si el operador temporal que forma el requisito $\mathcal{R}_{at}$ es el operador *Antes*, el estado E_k debe ser anterior al estado E_0. Para ello, basta con especificar el enlace entre los dos estados a través de la acción de subespecificación.

[6] Si el operador temporal que forma el requisito $\mathcal{R}_{at}$ es el operador *Después*, el estado E_k es un estado representante –$\widetilde{E_k}$– de todos los estados siguientes al estado E_0. Para ello, basta con especificar un arco con el valor $\frac{3}{4}$ entre los dos estados a través de todas las acciones del conjunto Λ.

[7] Fin del algoritmo.

6.3.2 Evaluación del Algoritmo de Traducción de Requisitos Atómicos I

El algoritmo 6.1 obtiene los grafos MUS correspondientes a los requisitos atómicos. No obstante, es necesario modificar dicho algoritmo para abordar la traducción de requisitos SCTL no atómicos, siguiendo así la estrategia indicada en la sección 6.1. A continuación, se citan las consideraciones necesarias para la adaptación de dicho algoritmo:

1. En el algoritmo 6.1 se parte de un grafo MUS $\mathcal{M}$ totalmente subespecificado. Sin embargo, la traducción de un requisito (aunque sea atómico) se realizará, en general, sobre grafos

MUS con partes especificadas –grafos sobre los que previamente podrán haberse traducido otros requisitos–. Por tanto, es necesario contemplar la posibilidad de que la especificación del requisito atómico se tenga que aplicar sobre una acción previamente especificada en el grafo MUS $\mathcal{M}$. En ese caso, la especificación de la acción en el requisito –$\mathcal{S}_{\mathcal{R}_{at}}(a_i, E_k)$– deberá ser la misma que la existente en el grafo MUS –$E_k[a_i]$–. Si ambas especificaciones no coinciden, se obtiene una especificación inconsistente, como la que se produce si se especifica el requisito:

req $\mathcal{R}_{inc}$ **is**
$\quad true \Rightarrow a_1 \ \wedge \ true \Rightarrow \neg a_1$
endreq

Por el mismo motivo que en el caso anterior, es necesario especificar el estado en el que se debe traducir el requisito atómico, ya que el grafo inicial $\mathcal{M}$ podrá tener más estados que el estado E_0 previsto en el algoritmo 6.1.

2. En el algoritmo 6.1 siempre se crea un nuevo estado para los operadores *Antes* y *Después*, debido a que no existe ningún otro estado en el sistema. Los nuevos estados creados permiten mostrar en el grafo $\mathcal{M}$ el comportamiento expresado por el requisito SCTL traducido. Sin embargo, si el grafo MUS inicial $\mathcal{M}$ contiene varios estados, será necesario decidir en qué estados debe aplicarse la consecuencia del requisito a traducir. La creación de nuevos estados deberá realizarse únicamente cuando sea estrictamente necesario –evitando la aparición de nuevos estados innecesarios–, tal y como sucede en el algoritmo 6.1. Es necesario, por tanto, la definición de un nuevo algoritmo –denominado **algoritmo de aplicabilidad**–, que obtenga los estados del grafo MUS $\mathcal{M}$ en los que debe aplicarse la consecuencia del requisito a traducir.

3. Para evitar traducir varias veces un requisito SCTL en el mismo estado del grafo MUS $\mathcal{M}$, realizando llamadas innecesarias al algoritmo de traducción, es necesario almacenar en cada estado del sistema una lista con los requisitos traducidos en dicho estado. El primer paso del algoritmo de traducción será, por tanto, comprobar si el requisito ya ha sido traducido en el estado correspondiente, en cuyo caso se finaliza el algoritmo.

Antes de proponer un nuevo algoritmo de traducción de requisitos atómicos que tenga en cuenta las consideraciones anteriores, es necesario definir los estados de aplicabilidad de los requisitos SCTL, a fin de aplicar el algoritmo de traducción en dichos estados. Por ello, en las siguientes secciones se describe, detalladamente, el algoritmo de aplicabilidad.

6.3.3 Estados de Aplicabilidad de los Requisitos SCTL

6.3.3.1 Algoritmo de Acciones de Aplicabilidad

En la sección 6.2.3 se definen las acciones de aplicabilidad de un requisito SCTL formado por el operador temporal *Después*, permitiendo así especificar el conjunto de estados siguientes en

los que debe aplicarse la consecuencia con dicho operador. La definición de acciones de aplicabilidad enriquece la expresividad de la lógica SCTL, ya que, permite especificar requisitos cuya consecuencia se aplique a uno, a varios o la totalidad de estados siguientes.

Definición 6.2. Sea $\mathcal{R}$ un requisito SCTL de la forma: $\mathcal{R} = (\mathcal{R}_1 \wedge \mathcal{R}_2 \wedge ...\mathcal{R}_n) \Rightarrow \bigcirc\, \mathcal{R}_{cons}$. Se definen las **Acciones de Aplicabilidad** del requisito $\mathcal{R}$, –denotadas por $\{\Lambda_{\mathcal{R}}\}$–, como aquellas acciones que se especifican como posibles en algún requisito atómico $\mathcal{R}_k$ de la premisa de $\mathcal{R}$, tal que: $\mathcal{R}_k = true \Rightarrow a_i$. $\{\Lambda_{\mathcal{R}}\} \triangleq \{a_i \in \Lambda' : \exists \mathcal{R}_k = true \Rightarrow a_i\}$.

El algoritmo 6.2 muestra el pseudocódigo del **algoritmo de acciones de aplicabilidad** que extrae los subrequisitos $\mathcal{R}_i$ con dichas características, devolviendo las acciones de aplicabilidad del requisito SCTL que recibe como parámetro. Este algoritmo sólo debe ser invocado por requisitos SCTL unidos por el operador temporal *Después*.

Algoritmo 6.2 Algoritmo de Acciones de Aplicabilidad ($\overline{\mathcal{R}} = \{\mathcal{R}[0], ..., \mathcal{R}[n]\}$)

---

[1] Si $\mathcal{R}$ es un requisito atómico: $\{\Lambda_{\mathcal{R}}\} = \Lambda$
[2] En caso contrario:
  [a] $\{\overline{\mathcal{R}}_{prem}, \overline{\mathcal{R}}_{consec}\}$ = **Algoritmo de Partición**[6] $(\overline{\boldsymbol{\mathcal{R}}})$;
  [b] $\{\Lambda_{\mathcal{R}}\}$ = **Buscar Acciones Aplicabilidad** $(\overline{\boldsymbol{\mathcal{R}}}_{\boldsymbol{prem}}, \emptyset)$;
  [c] Si $\{\Lambda_{\mathcal{R}}\} = \emptyset$: $\{\Lambda_{\mathcal{R}}\} = \Lambda$
[3] Devolver $\{\Lambda_{\mathcal{R}}\}$;

Buscar Acciones de Aplicabilidad $\overline{\mathcal{R}}, \{\mathcal{R}_{expand}\})$

---

[1] Si $\overline{\mathcal{R}} = \uparrow Id_{req}$:
  [a] Si $Id_{req} \in \{\mathcal{R}_{expand}\}$: devolver $\{A_{\mathcal{R}}\} = \emptyset$:
  [b] $\overline{\mathcal{R}}$ = **Expandir Requisito** (Id_{req});
  [c] $\{\mathcal{R}_{expand}\} = \{\mathcal{R}_{expand}\} \bigcup Id_{req}$;
[2] Si $\overline{\mathcal{R}}$ es un requisito atómico de la forma $true \Rightarrow a_i$, devolver $\{\Lambda_{\mathcal{R}}\} = \{a_i\}$;
[3] En caso contrario, si $\overline{\mathcal{R}}[0] = \wedge$:
  [a] $\{\overline{\mathcal{R}}_{sub_1}, \overline{\mathcal{R}}_{sub_2}\}$ = **Algoritmo de Partición** $(\overline{\boldsymbol{\mathcal{R}}})$;
  [b] $\{\Lambda_{\mathcal{R}}\} = \{\Lambda_{\mathcal{R}}\} \bigcup$ **Buscar Acciones de Aplicabilidad** $(\overline{\boldsymbol{\mathcal{R}}}_{\boldsymbol{sub_1}}, \boldsymbol{\mathcal{R}_{expand}})$;
  [c] $\{\Lambda_{\mathcal{R}}\} = \{\Lambda_{\mathcal{R}}\} \bigcup$ **Buscar Acciones de Aplicabilidad** $(\overline{\boldsymbol{\mathcal{R}}}_{\boldsymbol{sub_2}}, \boldsymbol{\mathcal{R}_{expand}})$;
[4] Devolver $\{\Lambda_{\mathcal{R}}\}$;

Es necesario expandir, al menos una vez, los requisitos instanciados mediante el símbolo $\uparrow$, ya que, estos pueden contener alguna acción de aplicabilidad. Para evitar expandir un requisito

[6] El algoritmo de partición extrae la premisa y la consecuencia de un requisito SCTL. Este algoritmo se describe detalladamente en el apéndice B (ver algoritmo B.2).

evaluado en alguna recursión anterior del algoritmo, se mantiene una lista –$\{\mathcal{R}_{expand}\}$– con los requisitos expandidos en cada una de las recursiones del mismo.

En el ejemplo 6.2 se muestra el conjunto $\{\Lambda_{\mathcal{R}}\}$ devuelto en cada recursión del algoritmo 6.2, cuando éste se invoca con un requisito SCTL, cuya premisa –$\mathcal{R}_{aplic}$– se muestra a continuación.

req $\mathcal{R}_{aplic}$ is
$(true \Rightarrow a_1) \wedge (((true \Rightarrow a_2) \ \Rightarrow \bigcirc (true \Rightarrow \bigodot a_2)) \wedge (true \Rightarrow \neg a_2) \wedge (true \Rightarrow a_3))$
endreq

Ejemplo 6.2 Ejemplo del Algoritmo de Acciones de Aplicabilidad.

Rec.	$\{\Lambda_{\mathcal{R}}\}$	$\overline{\mathcal{R}}$
1	$\{a_1, a_3\}$	$\mathcal{R}_{aplic}$
2	$\{a_1\}$	$true \Rightarrow a_1$
3	$\{a_3\}$	$((true \Rightarrow a_2) \ \Rightarrow \bigcirc (true \Rightarrow \bigodot a_2)) \wedge (true \Rightarrow \neg a_2) \wedge (true \Rightarrow a_3)$
4	$\emptyset$	$(true \Rightarrow a_2) \ \Rightarrow \bigcirc (true \Rightarrow \bigodot a_2)$
5	$\{a_3\}$	$(true \Rightarrow \neg a_2) \wedge (true \Rightarrow a_3)$
6	$\emptyset$	$true \Rightarrow \neg a_2$
7	$\{a_3\}$	$true \Rightarrow a_3$

El algoritmo 6.3 muestra el pseudocódigo del **algoritmo de aplicabilidad** que obtiene los estados de aplicabilidad de un operador temporal $\bigoplus \in \Theta$. Dicho algoritmo recibe como parámetros un requisito SCTL en notación inversa $\overline{\mathcal{R}}$, y el estado E_h del grafo MUS $\mathcal{M}$ donde se especifica dicho requisito; devolviendo el conjunto de estados de aplicabilidad –denotados por $\perp (\mathcal{R}, E_h)$–.

Algoritmo 6.3 Algoritmo de Aplicabilidad ($\overline{\mathcal{R}} = \{\overline{\mathcal{R}}[0], ..., \overline{\mathcal{R}}[n]\}, E_h$)

[1] Si $\overline{\mathcal{R}}[0] = \ \Rightarrow, \{E_{aplic}\} = \{E_h\}$;
[2] Si $\overline{\mathcal{R}}[0] = \ \Rightarrow \bigodot$;
$\{E_{aplic}\} = \{E_j \in \mathcal{E}'_{\mathcal{M}} : \exists a_i \in \Lambda \ / \ d[a_i][j][h] \geq \frac{3}{4}$ y $d[a_i][j][E_{sub}] = 1\}$;
[3] Si $\overline{\mathcal{R}}[0] = \ \Rightarrow \bigcirc$:
[a] $\{\Lambda_{\mathcal{R}}\}$ = **Algoritmo de Acciones de Aplicabilidad ($\mathcal{R}$)**;
[b] $\{E_{aplic}\} = \{E_j \in \mathcal{E}'_{\mathcal{M}} : a_i \in \Lambda_{\mathcal{R}} \ / \ d[a_i][h][j] \geq \frac{3}{4}$ y $d[a_i][h][E_{sub}] = 1$;
[4] Devolver: $\perp (\mathcal{R}, E_h) \triangleq \{E_{aplic}\}$;

Se consideran, como estados de aplicabilidad, los estados cuyos arcos se especifican mediante la acción de subespecificación –a_{sub}–, ya que, dichos arcos existen aunque no esté especificada la acción a través de la cual se produce la evolución. En el caso del operador *Antes*, se trata la

acción de subespecificación como cualquier otra acción del conjunto Λ'. En el caso del operador *Después*, las evoluciones a través de la acción de subespecificación sólo se consideran en el caso de que dicho operador afecte a todas las acciones; es decir, si el requisito es atómico o si no se especifica ninguna acción de aplicabilidad (ver los pasos [1] y [2][c] del algoritmo 6.2).

Obsérvese que el estado de subespecificación y los estados con arcos potenciales no se consideran como estados de aplicabilidad. Es decir, el algoritmo de aplicabilidad sólo considera las acciones y arcos totalmente especificados (cuyo valor es $true$ o $false$), limitándose a devolver un conjunto de estados en función del operador temporal especificado en el requisito SCTL. Esto es debido a que es utilizado por la totalidad de algoritmos desarrollados, tanto para el proceso de síntesis como de verificación, siendo necesario en algunos casos poder aplicar dichos algoritmos únicamente a los elementos especificados del sistema. Sin embargo, el algoritmo de traducción necesita considerar y almacenar toda la información posible según los grafos MUS, con el fin de representar fielmente el comportamiento especificado por el requisito SCTL a traducir.

Por tanto, es necesario la inclusión de un nuevo algoritmo, que se ha denominado **algoritmo de aplicabilidad potencial**, que considere los estados de aplicabilidad no tenidos en cuenta por el algoritmo 6.3. Dichos estados son los siguientes:

- Estados de aplicabilidad cuyos arcos están especificados como posibles pero cuya acción correspondiente está subespecificada (arcos potenciales).
- Nuevos estados creados para mostrar fielmente el comportamiento del requisito a traducir, tal y como se puso de manifiesto en el algoritmo 6.1, al no existir estados de aplicabilidad en el sistema. En el caso del operador *Después* será necesario crear un nuevo estado para el conjunto de acciones de aplicabilidad del requisito que no tienen especificado el estado siguiente y que están especificadas como *posibles*. En el caso del operador *Antes* se creará un nuevo estado únicamente cuando no exista ningún estado anterior (ningún estado de aplicabilidad), mostrando así el comportamiento especificado por el requisito SCTL a traducir.
- Estados representantes, ya que, estos integran un conjunto de estados del sistema. Dichos estados deberán ser considerados como estados de aplicabilidad si lo fueran todos sus estados representados. En caso contrario –si el requisito especificado afecta sólo a algún estado representado–, es necesario crear un nuevo estado (igual al representante) donde aplicar el requisito especificado. Estos nuevos estados creados son una réplica del estado representante correspondiente, y pueden ser nuevos estados representantes o estados reales, en función de que almacenen restricciones comunes a uno o a varios estados.

6.3.3.2 Algoritmo de Aplicabilidad Potencial

El algoritmo de aplicabilidad potencial (ver algoritmo 6.4) calcula los estados de aplicabilidad que no tiene en cuenta el algoritmo de aplicabilidad, creando los estados (representantes o no) necesarios.

Algoritmo 6.4 Algoritmo de Aplicabilidad Potencial ($\overline{\mathcal{R}} = \{\overline{\mathcal{R}}[0], ..., \overline{\mathcal{R}}[n]\}, E_h$)

[1] Si $\overline{\mathcal{R}}[0] = \Rightarrow$, $\{E_{pot}\} = \{\emptyset\}$;
[2] Si $\overline{\mathcal{R}}[0] = \Rightarrow \bigodot$:
  [a] $\{E_{pot}\} = \{E_j \in \mathcal{E}'_{\mathcal{M}} : \exists a_i \in \Lambda' \ / \ d[a_i][j][h] = 1$ y $d[a_i][j][E_{sub}] = \frac{1}{2}\}$;
  [b] Si $\{E_{pot}\} = \{\emptyset\}$ y $\nexists a_i \in \Lambda', E_j \in \mathcal{E}'_{\mathcal{M}} \ / \ d[a_i][j][h] = 1$:
    [i] Nuevo estado E_k;
    [ii] $d[a_{sub}][k][h] = 1$;
    [iii] $\{E_{pot}\} = \{E_k\}$;
[3] Si $\overline{\mathcal{R}}[0] = \Rightarrow \bigcirc$:
  [a] $\{\Lambda_{\mathcal{R}}\}$ = **Algoritmo de Acciones de Aplicabilidad** $(\mathcal{R})$;
  [b] $\{E_{pot}\} = \{E_j \in \mathcal{E}'_{\mathcal{M}} : \exists a_i \in \Lambda'_{\mathcal{R}} \ / \ d[a_i][h][j] = 1$ y $d[a_i][h][E_{sub}] = \frac{1}{2}\}$;
  [c] $\forall a_i \in \Lambda_{\mathcal{R}}, \widetilde{E_j} \in \mathcal{E}'_{\mathcal{M}} \ / \ d[a_i][h][j] = \frac{3}{4}$:
    [i] $\{A_{rep_h}\} = \{a_l \in \Lambda' : d[a_l][h][j] = \frac{3}{4}\}$;
    [ii] Si $\{A_{rep_h}\} \subseteq \{A_{\mathcal{R}}\}$ y $\nexists\, a_i \in \Lambda', E_g \in \mathcal{E}'_{\mathcal{M}} - \{E_h\} \ / \ d[a_i][g][j] = \frac{3}{4}$:
      $\{E_{pot}\} = \{E_{pot}\} \bigcup \widetilde{E_j}$;
    [iii] En caso contrario:
      [A] $\{A_{\mathcal{R}_{rep}}\} = \{A_{\mathcal{R}}\} \bigcap \{A_{rep_h}\}$;
      [B] Si $\sharp\{A_{\mathcal{R}_{rep}}\} > 1$: $tipo\_arco = \frac{3}{4}$. En caso contrario: $tipo\_arco = 1$;
      [C] Desdoblar estado representante $\widetilde{E_j}$ creando E_k;
      [D] $\forall a_i \in \{A_{\mathcal{R}_{rep}}\}$: $d[a_i][h][j] = \frac{1}{2}$, $d[a_i][h][k] = tipo\_arco$;
      [E] $\{E_{pot}\} = \{E_{pot}\} \bigcup E_k$;
  [d] $\{A_{pot}\} = \{a_i \in \Lambda_{\mathcal{R}} : \nexists E_j \in \mathcal{E}'_{\mathcal{M}} \ / \ d[a_i][h][j] = 1$ y $d[a_i][h][E_{sub}] \neq 0\}$;
  [e] Si $\{A_{pot}\} \neq \{\emptyset\}$:
    [i] Si $\sharp\{A_{pot}\} > 1$: $tipo\_arco = \frac{3}{4}$. En caso contrario: $tipo\_arco = 1$;
    [ii] Nuevo estado E_k;
    [iii] $\forall a_i \in \{A_{pot}\}$, $d[a_i][h][k] = tipo\_arco$;
    [iv] $\{E_{pot}\} = \{E_{pot}\} \bigcup E_k$;
[4] Devolver: $\perp' (\mathcal{R}, E_h) \triangleq \{E_{pot}\}$;

A continuación, se resumen los estados de aplicabilidad considerados por dicho algoritmo:

- Operador temporal *Antes*: Enlaces especificados a través de acciones subespecificadas –arcos potenciales–. Ver el paso [2][a].
- Operador temporal *Después*:
  - Enlaces especificados a través de acciones subespecificadas –arcos potenciales–. Ver el paso [3][b].
  - Estados representantes, cuyos estados representados están todos incluidos en los estados de aplicabilidad del requisito especificado. Ver el paso [3][c][ii].

Además, el algoritmo 6.4 crea nuevos estados en los siguientes casos:

- Operador temporal *Antes*: Si no existe ningún estado de aplicabilidad, incluyendo los estados con arcos potenciales. Ver el paso [2][b].
- Operador temporal *Después*:
  - Si es necesario desdoblar un estado representante, debido a que el requisito se debe aplicar a sólo una parte de los estados representados. Ver paso [3][c][iii].
  - Si no existen los estados siguientes correspondientes a las acciones de aplicabilidad, siempre y cuando éstas no estén especificadas como no posibles. Ver paso [3][e].

6.3.4 Algoritmo de Traducción de Requisitos Atómicos II

En el algoritmo 6.5 se muestra el pseudocódigo del **algoritmo de traducción de requisitos atómicos II**, que tiene en cuenta las consideraciones apuntadas en la sección anterior. Dicho algoritmo recibe como parámetro, además del requisito SCTL atómico a traducir, el estado E_h donde se especifica dicho requisito. El algoritmo devuelve un código de error si no es posible obtener un modelo de estados consistente según el requisito especificado, proporcionando así, un mecanismo de detección de requisitos SCTL inconsistentes.

Algoritmo 6.5 Algoritmo de Traducción de Requisitos Atómicos II ($\overline{\mathcal{R}}_{at} = \bigoplus true[\neg]a_i, E_h$)

[1] Si ya se ha traducido $\overline{\mathcal{R}}$ en el estado E_h de $\mathcal{M}$, devolver OK;
[2] $\{E_{aplic}\} =$ **Algoritmo de Aplicabilidad** $(\overline{\boldsymbol{\mathcal{R}}}, \mathbf{E_h})$;
[3] $\{E_{aplic}\} = \{E_{aplic}\} \bigcup$ **Algoritmo de Aplicabilidad Potencial** $(\overline{\boldsymbol{\mathcal{R}}}, \mathbf{E_h})$;
[4] $\forall E_j \in \{E_{aplic}\},\ d[a_i][j][E_{sub}] = d[a_i][j][E_{sub}] \cup$ [7] $\mathcal{S}_{\mathcal{R}_{at}}(a_i, E_j)$;
[5] Si $\mathcal{S}_{\mathcal{R}_{at}}(a_i, E_j) = 1,\ d[a_{sub}][j][E_{sub}] = 1$;
[6] Si alguna operación *unión* da un resultado no incluido en Ψ: devolver ERROR;
[7] En caso contrario: devolver OK;

Los estados devueltos por el algoritmo de aplicabilidad potencial se unen a los estados de aplicabilidad devueltos por el algoritmo de aplicabilidad, tal y como se indica en el paso **[3]** del algoritmo 6.5.

6.4 Algoritmo de Traducción SCTL-MUS

Una vez definidos los modelos de estados subespecificados correspondientes a los requisitos atómicos, es posible obtener un algoritmo de traducción entre requisitos SCTL y grafos MUS.

[7] La operación *unión* devuelve error si los operandos son 1 y 0. Si los operandos son iguales, el resultado es el mismo valor. En caso contrario, el resultado se corresponde con el valor del elemento especificado. La operación *unión* se define formalmente en el capítulo 7.

Su pseudocódigo se muestra en el algoritmo 6.6. Dicho algoritmo recibe como parámetros un requisito SCTL en notación inversa y el estado E_h del grafo MUS donde se especifica el requisito. Aunque por simplicidad en el pseudocódigo no se indica, el algoritmo devuelve un código de error si el requisito especificado no puede traducirse a un modelo de estados subespecificados, detectando así la inconsistencia del mismo.

Algoritmo 6.6 Algoritmo de Traducción SCTL-MUS ($\overline{\mathcal{R}} = \{\overline{\mathcal{R}}[0], ..., \overline{\mathcal{R}}[n]\}, E_h$)

[1] Si ya se ha traducido $\overline{\mathcal{R}}$ en el estado E_h de $\mathcal{M}$, devolver OK;
[2] Si $\overline{\mathcal{R}}$ es un requisito atómico:
  [a] Algoritmo de Traducción de Requisitos Atómicos $(\overline{\boldsymbol{\mathcal{R}}}, \mathbf{E_h})$;
  [b] Fin;
[3] $\{\overline{\mathcal{R}}_{sub_1}, \overline{\mathcal{R}}_{sub_2}\} =$ **Algoritmo de Partición** $(\overline{\boldsymbol{\mathcal{R}}})$;
[4] Algoritmo de Traducción SCTL-MUS $(\overline{\boldsymbol{\mathcal{R}}}_{\boldsymbol{sub_1}}, \mathbf{E_h})$;
[5] Si $\overline{\mathcal{R}}[0] = \wedge$:
  [a] Algoritmo de Traducción SCTL-MUS $(\overline{\boldsymbol{\mathcal{R}}}_{\boldsymbol{sub_2}}, \mathbf{E_h})$;
  [b] Fin;
[6] Si $\overline{R}[0] \in \Theta$:
  [a] $\{E_{aplic}\} =$ **Algoritmo de Aplicabilidad** $(\overline{\boldsymbol{\mathcal{R}}}, \mathbf{E_h})$;
  [b] $\{E_{aplic}\} = \{E_{aplic}\} \bigcup$ **Algoritmo de Aplicabilidad Potencial** $(\overline{\boldsymbol{\mathcal{R}}}, \mathbf{E_h})$;
  [c] $\forall E_j \in \{E_{aplic}\}$, **Algoritmo de Traducción SCTL-MUS** $(\overline{\boldsymbol{\mathcal{R}}}_{\boldsymbol{sub_2}}, \mathbf{E_j})$;
  [d] Fin;

El funcionamiento general del algoritmo de traducción de requisitos SCTL-MUS, que se muestra en el algoritmo 6.6, es el siguiente:

- Si $\mathcal{R}$ es un requisito atómico, el grafo MUS se obtiene mediante el algoritmo de traducción de requisitos atómicos. Ver el paso [2][a].

- En caso contrario, se obtienen, mediante el algoritmo de partición[8], los dos subrequisitos que componen el requisito $\mathcal{R}$. Si el requisito $\mathcal{R}$ se forma uniendo dos requisitos mediante el operador lógico *And* –$\mathcal{R}[0] = \wedge$–, el grafo MUS correspondiente se obtiene mediante sendas llamadas recursivas a dicho algoritmo con los subrequisitos $\mathcal{R}_{sub_1}$ y $\mathcal{R}_{sub_2}$, y el mismo estado E_h. Ver los pasos [4] y [5].

- Si el requisito $\mathcal{R}$ se forma uniendo dos requisitos mediante un operador temporal $\bigoplus \in \Theta$, el modelo de estados también se obtiene mediante sendas llamadas recursivas a dicho algoritmo con los subrequisitos $\mathcal{R}_{sub_1}$ y $\mathcal{R}_{sub_2}$. Sin embargo, mientras que en la primera llamada se mantiene el estado E_h –estado donde debe satisfacerse la premisa–, la segunda llamada se repite para todos los estados de aplicabilidad del operador temporal que une ambos requisitos. Ver el paso [6].

[8] Ver algoritmo B.2.

Obsérvese que en el algoritmo 6.6 no se tienen en cuenta los requisitos SCTL unidos mediante el operador lógico *Or*. Esto es debido a que dicho algoritmo traduce trazas de requisitos SCTL en forma normal positiva –un requisito SCTL sin el operador lógico *Or*, y con la negación absorbida hasta las acciones–. Los algoritmos encargados de realizar las tareas de síntesis incremental y de verificación deberán ser capaces de extraer las trazas de un requisito SCTL, aplicando dichos algoritmos sobre la traza correspondiente, según las decisiones de diseño tomadas.[9]

6.5 Traducción de Requisitos Recursivos

La traducción de requisitos SCTL recursivos puede conllevar la síntesis de un grafo MUS con infinitos estados. Naturalmente, no es posible sintetizar un modelo de estados en estos términos, por lo que se tiene que adoptar una solución ante la aparición de dichos requisitos recursivos.

Una posible solución parte por encontrar trazas que se repiten dentro del grafo MUS correspondiente al requisito a traducir, formando así bucles, y evitando la síntesis de un grafo con un número infinito de estados. El problema se centra, por tanto, en la detección de bucles dentro del grafo MUS. Este problema no tiene fácil solución, ya que, obliga a buscar dichos bucles cada vez que aparece un nuevo estado en el sistema, o simplemente cada vez que aparece una nueva acción en un estado ya existente.

Sin embargo, los requisitos SCTL recursivos se construyen mediante la instanciación de otros requisitos SCTL, anteponiendo el símbolo $\uparrow$ al identificador del requisito correspondiente. Debido a que únicamente es posible obtener grafos MUS con infinitos estados a partir de requisitos recursivos, la solución se limita a no traducir el grafo correspondiente al requisito instanciado hasta que sea necesario (por ejemplo, para tareas de verificación), y mantener en el grafo MUS, que a partir de cierto estado el sistema se comporta según el requisito instanciado. Para ilustrar la traducción de requisitos recursivos, en la figura 6.5 se muestra el modelo de estados subespecificados del requisito R_{rec}:

req $\mathcal{R}_{rec}$ **is**
$(true \Rightarrow a_1) \Rightarrow \bigcirc \uparrow \mathcal{R}_{rec}$
endreq

El requisito R_{rec} es equivalente a especificar que la acción a_1 debe ser siempre posible a partir del estado en el que se especifica dicho requisito. Por tanto, la traducción de R_{rec} a un modelo de estados subespecificados conllevaría la síntesis de un grafo con infinitos estados. Tal y como se explicó anteriormente, la solución adoptada parte por permitir especificar en un estado de MUS la instanciación de un requisito SCTL –figura 6.5.a–, que podrá ser traducido a medida que es necesario, lo que equivale a expandir el grafo –figura 6.5.b–. Por tanto, la representación matricial de un grafo subespecificado se enriquece con la aparición de una nueva matriz compuesta por N filas, una por cada estado del grafo. Cada fila, podrá constar de un número indeterminado de

[9] Ver partes III y IV.

identificadores de requisitos, correspondientes a los requisitos instanciados en dicho estado, que pueden ser expandidos. En el ejemplo de la figura 6.5(a) existiría un par $[E_0][R_{rec}]$, mientras que en el grafo expandido correspondiente a la figura 6.5(b), existiría el par $[E_1][R_{rec}]$.

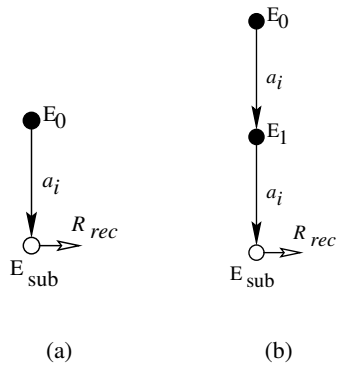


Figura 6.5: Traducción del requisito SCTL recursivo R_{rec}.

6.6 Ejemplo del Algoritmo de Traducción SCTL-MUS

La figura 6.6 muestra el modelo de estados subespecificados correspondiente al requisito $\mathcal{R}_{trad}$. los ejemplos 6.3 y 6.4 detallan cada uno de los pasos ejecutados por el algoritmo de traducción hasta obtener el modelo de estados subespecificados correspondiente al requisito $\mathcal{R}$.

req $\mathcal{R}_{trad}$ is

$$(true \Rightarrow a_1) \wedge (true \Rightarrow \odot\, a_2) \Rightarrow \bigcirc\, (true \Rightarrow \odot\, a_3)$$

endreq

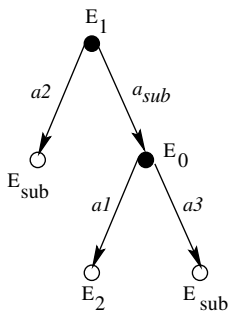


Figura 6.6: Ejemplo del Algoritmo de Traducción SCTL-MUS.

Ejemplo 6.3 Ejemplo del Algoritmo de Traducción I.

[1] Sistema inicial:

$\forall a_i$	$\mathbf{E}_0$	$\mathbf{E}_{sub}$
$\mathbf{E}_0$	$\frac{1}{2}$	$\frac{1}{2}$
$\mathbf{E}_{sub}$	$\frac{1}{2}$	$\frac{1}{2}$

[2] Algoritmo de Traducción $((true \Rightarrow a_1) \wedge (true \Rightarrow \odot a_2) \Rightarrow \bigcirc(true \Rightarrow \odot a_3), E_0)$;

[a] Algoritmo de Traducción $((true \Rightarrow a_1) \wedge (true \Rightarrow \odot a_2), E_0)$;

[i] Algoritmo de Traducción $((true \Rightarrow a_1), E_0)$;

[A] $\{E_{aplic}\} = \{E_0\} \bigcup \{\emptyset\}$:

$\mathbf{a}_1$	$\mathbf{E}_0$	$\mathbf{E}_{sub}$
$\mathbf{E}_0$	$\frac{1}{2}$	1
$\mathbf{E}_{sub}$	$\frac{1}{2}$	$\frac{1}{2}$

$\mathbf{a}_{sub}$	$\mathbf{E}_0$	$\mathbf{E}_{sub}$
$\mathbf{E}_0$	$\frac{1}{2}$	1
$\mathbf{E}_{sub}$	$\frac{1}{2}$	$\frac{1}{2}$

[ii] Algoritmo de Traducción $((true \Rightarrow \odot a_2), E_0)$;

[A] $\{E_{aplic}\} = \{\emptyset\} \bigcup \{E_1\}$;

$\mathbf{a}_1$	$\mathbf{E}_0$	$\mathbf{E}_1$	$\mathbf{E}_{sub}$
$\mathbf{E}_0$	$\frac{1}{2}$	$\frac{1}{2}$	1
$\mathbf{E}_1$	$\frac{1}{2}$	$\frac{1}{2}$	$\frac{1}{2}$
$\mathbf{E}_{sub}$	$\frac{1}{2}$	$\frac{1}{2}$	$\frac{1}{2}$

$\mathbf{a}_2$	$\mathbf{E}_0$	$\mathbf{E}_1$	$\mathbf{E}_{sub}$
$\mathbf{E}_0$	$\frac{1}{2}$	$\frac{1}{2}$	$\frac{1}{2}$
$\mathbf{E}_1$	$\frac{1}{2}$	$\frac{1}{2}$	1
$\mathbf{E}_{sub}$	$\frac{1}{2}$	$\frac{1}{2}$	$\frac{1}{2}$

$\mathbf{a}_3$	$\mathbf{E}_0$	$\mathbf{E}_1$	$\mathbf{E}_{sub}$
$\mathbf{E}_0$	$\frac{1}{2}$	$\frac{1}{2}$	$\frac{1}{2}$
$\mathbf{E}_1$	$\frac{1}{2}$	$\frac{1}{2}$	$\frac{1}{2}$
$\mathbf{E}_{sub}$	$\frac{1}{2}$	$\frac{1}{2}$	$\frac{1}{2}$

$\mathbf{a}_{sub}$	$\mathbf{E}_0$	$\mathbf{E}_1$	$\mathbf{E}_{sub}$
$\mathbf{E}_0$	$\frac{1}{2}$	$\frac{1}{2}$	1
$\mathbf{E}_1$	1	$\frac{1}{2}$	1
$\mathbf{E}_{sub}$	$\frac{1}{2}$	$\frac{1}{2}$	$\frac{1}{2}$

Ejemplo 6.4 Ejemplo del Algoritmo de Traducción II.

[b] $\{E_{aplic}\} = \{\emptyset\} \bigcup \{E_2\}$:

$\mathbf{a}_1$	$\mathbf{E}_0$	$\mathbf{E}_1$	$\mathbf{E}_2$	$\mathbf{E}_{sub}$
$\mathbf{E}_0$	$\frac{1}{2}$	$\frac{1}{2}$	1	1
$\mathbf{E}_1$	$\frac{1}{2}$	$\frac{1}{2}$	$\frac{1}{2}$	$\frac{1}{2}$
$\mathbf{E}_2$	$\frac{1}{2}$	$\frac{1}{2}$	$\frac{1}{2}$	$\frac{1}{2}$
$\mathbf{E}_{sub}$	$\frac{1}{2}$	$\frac{1}{2}$	$\frac{1}{2}$	$\frac{1}{2}$

$\mathbf{a}_2$	$\mathbf{E}_0$	$\mathbf{E}_1$	$\mathbf{E}_2$	$\mathbf{E}_{sub}$
$\mathbf{E}_0$	$\frac{1}{2}$	$\frac{1}{2}$	$\frac{1}{2}$	$\frac{1}{2}$
$\mathbf{E}_1$	$\frac{1}{2}$	$\frac{1}{2}$	$\frac{1}{2}$	1
$\mathbf{E}_2$	$\frac{1}{2}$	$\frac{1}{2}$	$\frac{1}{2}$	$\frac{1}{2}$
$\mathbf{E}_{sub}$	$\frac{1}{2}$	$\frac{1}{2}$	$\frac{1}{2}$	$\frac{1}{2}$

$\mathbf{a}_3$	$\mathbf{E}_0$	$\mathbf{E}_1$	$\mathbf{E}_2$	$\mathbf{E}_{sub}$
$\mathbf{E}_0$	$\frac{1}{2}$	$\frac{1}{2}$	$\frac{1}{2}$	$\frac{1}{2}$
$\mathbf{E}_1$	$\frac{1}{2}$	$\frac{1}{2}$	$\frac{1}{2}$	$\frac{1}{2}$
$\mathbf{E}_2$	$\frac{1}{2}$	$\frac{1}{2}$	$\frac{1}{2}$	$\frac{1}{2}$
$\mathbf{E}_{sub}$	$\frac{1}{2}$	$\frac{1}{2}$	$\frac{1}{2}$	$\frac{1}{2}$

$\mathbf{a}_{sub}$	$\mathbf{E}_0$	$\mathbf{E}_1$	$\mathbf{E}_2$	$\mathbf{E}_{sub}$
$\mathbf{E}_0$	$\frac{1}{2}$	$\frac{1}{2}$	1	1
$\mathbf{E}_1$	1	$\frac{1}{2}$	$\frac{1}{2}$	$\frac{1}{2}$
$\mathbf{E}_2$	$\frac{1}{2}$	$\frac{1}{2}$	$\frac{1}{2}$	$\frac{1}{2}$
$\mathbf{E}_{sub}$	$\frac{1}{2}$	$\frac{1}{2}$	$\frac{1}{2}$	$\frac{1}{2}$

[c] Algoritmo de Traducción$((true \Rightarrow \bigodot a_3), E_2)$;

[i] $\{E_{aplic}\} = \{E_0\} \bigcup \{\emptyset\}$:

$\mathbf{a}_1$	$\mathbf{E}_0$	$\mathbf{E}_1$	$\mathbf{E}_2$	$\mathbf{E}_{sub}$
$\mathbf{E}_0$	$\frac{1}{2}$	$\frac{1}{2}$	1	1
$\mathbf{E}_1$	$\frac{1}{2}$	$\frac{1}{2}$	$\frac{1}{2}$	$\frac{1}{2}$
$\mathbf{E}_2$	$\frac{1}{2}$	$\frac{1}{2}$	$\frac{1}{2}$	$\frac{1}{2}$
$\mathbf{E}_{sub}$	$\frac{1}{2}$	$\frac{1}{2}$	$\frac{1}{2}$	$\frac{1}{2}$

$\mathbf{a}_2$	$\mathbf{E}_0$	$\mathbf{E}_1$	$\mathbf{E}_2$	$\mathbf{E}_{sub}$
$\mathbf{E}_0$	$\frac{1}{2}$	$\frac{1}{2}$	$\frac{1}{2}$	$\frac{1}{2}$
$\mathbf{E}_1$	$\frac{1}{2}$	$\frac{1}{2}$	$\frac{1}{2}$	1
$\mathbf{E}_2$	$\frac{1}{2}$	$\frac{1}{2}$	$\frac{1}{2}$	$\frac{1}{2}$
$\mathbf{E}_{sub}$	$\frac{1}{2}$	$\frac{1}{2}$	$\frac{1}{2}$	$\frac{1}{2}$

$\mathbf{a}_3$	$\mathbf{E}_0$	$\mathbf{E}_1$	$\mathbf{E}_2$	$\mathbf{E}_{sub}$
$\mathbf{E}_0$	$\frac{1}{2}$	$\frac{1}{2}$	$\frac{1}{2}$	1
$\mathbf{E}_1$	$\frac{1}{2}$	$\frac{1}{2}$	$\frac{1}{2}$	$\frac{1}{2}$
$\mathbf{E}_2$	$\frac{1}{2}$	$\frac{1}{2}$	$\frac{1}{2}$	$\frac{1}{2}$
$\mathbf{E}_{sub}$	$\frac{1}{2}$	$\frac{1}{2}$	$\frac{1}{2}$	$\frac{1}{2}$

$\mathbf{a}_{sub}$	$\mathbf{E}_0$	$\mathbf{E}_1$	$\mathbf{E}_2$	$\mathbf{E}_{sub}$
$\mathbf{E}_0$	$\frac{1}{2}$	$\frac{1}{2}$	1	1
$\mathbf{E}_1$	1	$\frac{1}{2}$	$\frac{1}{2}$	1
$\mathbf{E}_2$	$\frac{1}{2}$	$\frac{1}{2}$	$\frac{1}{2}$	$\frac{1}{2}$
$\mathbf{E}_{sub}$	$\frac{1}{2}$	$\frac{1}{2}$	$\frac{1}{2}$	$\frac{1}{2}$

[3] Devolver OK;

Parte III

Verificación

Capítulo 7

Grados de Satisfacción de los Requisitos SCTL

7.1 Introducción

Una vez obtenida una especificación formal SCTL, es necesario poder realizar tareas de análisis y verificación sobre dicha especificación. En la metodología SCTL-MUS, la verificación adquiere un papel relevante, debido al carácter incremental del proceso de desarrollo propuesto. Esto supone realizar nuevas tareas de verificación cada vez que el usuario identifica un nuevo requisito SCTL –en cada iteración del modelo de desarrollo software propuesto–. El objetivo de dichas tareas es asegurar la consistencia de los requisitos especificados, así como comprobar que se mantiene la satisfacción de los requisitos previamente especificados.

La especificación de un sistema utilizando la metodología SCTL-MUS se compone de un conjunto de requisitos SCTL unidos mediante los operadores lógicos *And* y *Or*. Para poder analizar dicha especificación, así como para poder añadirle un nuevo requisito SCTL, es necesario definir una relación de equivalencia entre fórmulas o requisitos SCTL. De esta manera, para añadir un requisito a una especificación habrá que seguir los siguientes pasos:

1. Comprobar si la especificación implica (bajo dicha relación de equivalencia) el nuevo requisito SCTL especificado.

2. En caso afirmativo, la especificación SCTL ya satisface el nuevo requisito.

3. En caso contrario, es necesario añadir a la especificación el nuevo requisito. Para ello, bastará con unirlo mediante el operador lógico *And* a la especificación lógica anterior.

4. Comprobar que la nueva especificación es consistente, es decir, que no contiene requisitos contradictorios.

La verificación se reduce, por tanto, a comprobar si una fórmula lógica implica otra. Si además

se satisface la relación inversa, se obtienen fórmulas lógicas equivalentes, o lo que es lo mismo, especificaciones SCTL equivalentes.

La técnica de verificación que se propone *a priori* es, por tanto, la demostración de teoremas. Para ello, es necesario dotar de una sólida base matemática a la lógica SCTL y a la relación de equivalencia definida entre sus fórmulas. Es decir, es necesario dotar a SCTL de un conjunto de reglas, propiedades, axiomas, etc; que permitan reducir y transformar dichas fórmulas, obteniendo otras equivalentes.

En primer lugar, se debe definir una relación de equivalencia entre requisitos SCTL. Dicha relación está basada en el grado de satisfacción de los requisitos involucrados. Informalmente, diremos que dos requisitos son equivalentes si los dos tienen el mismo grado de satisfacción.

En los sistemas clásicos, el grado de satisfacción de una propiedad o requisito se reduce a un valor *booleano* –*verdadero* o *falso*–. La lógica SCTL introduce el concepto de subespecificación, lo que permite obtener grados de satisfacción intermedios. Estos grados de satisfacción de un requisito SCTL proporcionan una información muy útil en las primeras etapas de diseño, enriqueciendo la expresividad de la lógica utilizada.

A continuación, se muestra cómo un conjunto de grados de satisfacción más amplio que el *verdadero* o *falso* proporciona información adicional al diseñador, lo que permitirá detectar errores en la versión actual del sistema, así como detectar errores potenciales –errores que se producirán si el sistema sufre una determinada evolución–:

- Si un requisito no se especifica –está subespecificado–, su grado de satisfacción no debería ser *falso*, como ocurre en especificaciones con lógicas de dos estados. Los requisitos subespecificados deben tener, por tanto, un grado de satisfacción relacionado con su subespecificación, ya que pueden satisfacerse o no, en función de cómo se especifiquen en el futuro. Naturalmente, en la fase de implementación no pueden existir elementos subespecificados, por lo que el grado de satisfacción de todos los requisitos será finalmente *verdadero* o *falso*, producto de la pérdida de subespecificación necesaria en la fase de implementación [1].

- Dentro de los requisitos subespecificados, es posible distinguir varios grados de subespecificación. Esto es debido a que un requisito puede estar parcialmente subespecificado, lo que se corresponde con que algunos de los elementos que lo componen estén subespecificados y otros no. El grado de satisfacción de un requisito parcialmente subespecificado y uno totalmente subespecificado no debería ser el mismo. De lo contrario, se estarían tratando ambos requisitos de la misma manera, con la consiguiente pérdida de información y expresividad. Dentro de los requisitos parcialmente subespecificados, es posible distinguir entre:

  - Requisitos que no pueden ser *verdaderos*. Es decir, su grado actual de satisfacción es subespecificado, pero si se pierde dicha subespecificación, evolucionarán hacia un grado de satisfacción *falso*.

[1] Ver capítulo 10.

- Requisitos que no pueden ser *falsos*. Similares a los anteriores, salvo que la pérdida de subespecificación hace que dichos requisitos sean *verdaderos*.
- Requisitos que no pueden ser *falsos* y que no pueden ser *verdaderos*. Son requisitos que no satisfacen la condición de aplicabilidad de los requisitos SCTL. Es decir, son requisitos cuya premisa no puede satisfacerse, por lo que se les asigna un nuevo grado de satisfacción denominado *contradictorio*.

En resumen, el grado de satisfacción de un requisito varía en función de su proximidad al valor *verdadero* o al *falso*. Un requisito SCTL está totalmente especificado si su grado de satisfacción es *verdadero* o *falso*. Si por el contrario, el requisito está subespecificado, es posible distinguir entre requisitos totalmente subespecificados y requisitos parcialmente subespecificados. Además, los requisitos parcialmente subespecificados pueden clasificarse en función de su posibilidad para evolucionar hacia los grados de satisfacción *verdadero* o *falso*.

A continuación, se define formalmente un conjunto con seis grados de satisfacción, así como tres operaciones (suma, producto y complementación) sobre dicho conjunto, que le dotan de una estructura de álgebra de *De Morgan* [RW92], que, en esta tesis, se ha particularizado y denominado álgebra de **I**ncertidumbre del **P**unto **M**edio. De esta manera, es posible aplicar todos los teoremas ya estudiados y demostrados para el álgebra de *De Morgan*, facilitando así la realización de verificación de propiedades –requisitos SCTL– mediante la demostración de teoremas.

7.2 Álgebra de Incertidumbre del Punto Medio

Definición 7.1. Sea $\Phi \triangleq \{0,\ \frac{1}{4},\ \widehat{\frac{1}{2}},\ \frac{1}{2},\ \frac{3}{4},\ 1\}$ el **Conjunto de Grados de Satisfacción** . Se definen las siguientes operaciones internas a dicho conjunto:

Definición 7.2. Una operación binaria denominada **Suma** y denotada por $+$:

$$\begin{array}{ccc} \Phi \text{ x } \Phi & \rightarrow & \Phi \\ (a,b) & \mapsto & a+b \end{array}$$

$+$	0	$\frac{1}{4}$	$\widehat{\frac{1}{2}}$	$\frac{1}{2}$	$\frac{3}{4}$	1
0	0	$\frac{1}{4}$	$\widehat{\frac{1}{2}}$	$\frac{1}{2}$	$\frac{3}{4}$	1
$\frac{1}{4}$	$\frac{1}{4}$	$\frac{1}{4}$	$\widehat{\frac{1}{2}}$	$\frac{1}{2}$	$\frac{3}{4}$	1
$\widehat{\frac{1}{2}}$	$\widehat{\frac{1}{2}}$	$\widehat{\frac{1}{2}}$	$\widehat{\frac{1}{2}}$	$\frac{1}{2}$	$\frac{3}{4}$	1
$\frac{1}{2}$	$\frac{1}{2}$	$\frac{1}{2}$	$\frac{1}{2}$	$\frac{1}{2}$	$\frac{1}{2}$	$\frac{1}{2}$
$\frac{3}{4}$	$\frac{3}{4}$	$\frac{3}{4}$	$\frac{3}{4}$	$\frac{3}{4}$	$\frac{3}{4}$	1
1	1	1	1	1	1	1

Definición 7.3. Una operación denominada **Producto** y denotada por $\cdot$:

$$\begin{array}{ccc} \Phi \text{ x } \Phi & \rightarrow & \Phi \\ (a,b) & \mapsto & a \cdot b \end{array}$$

$\cdot$	0	$\frac{1}{4}$	$\widehat{\frac{1}{2}}$	$\frac{1}{2}$	$\frac{3}{4}$	1
0	0	0	0	0	0	0
$\frac{1}{4}$	0	$\frac{1}{4}$	$\frac{1}{4}$	$\frac{1}{4}$	$\frac{1}{4}$	$\frac{1}{4}$
$\widehat{\frac{1}{2}}$	0	$\frac{1}{4}$	$\widehat{\frac{1}{2}}$	$\widehat{\frac{1}{2}}$	$\widehat{\frac{1}{2}}$	$\widehat{\frac{1}{2}}$
$\frac{1}{2}$	0	$\frac{1}{4}$	$\widehat{\frac{1}{2}}$	$\frac{1}{2}$	$\frac{3}{4}$	1
$\frac{3}{4}$	0	$\frac{1}{4}$	$\widehat{\frac{1}{2}}$	$\frac{1}{2}$	$\frac{3}{4}$	$\frac{3}{4}$
1	0	$\frac{1}{4}$	$\widehat{\frac{1}{2}}$	$\frac{1}{2}$	$\frac{3}{4}$	1

Definición 7.4. Una operación monaria denominada **Complementación** y denotada por $^{-}$:

$$\begin{array}{ccc} \Phi & \rightarrow & \Phi \\ a & \mapsto & \overline{a} \end{array}$$

a	$\overline{\mathbf{a}}$
0	1
$\frac{1}{4}$	$\frac{3}{4}$
$\widehat{\frac{1}{2}}$	$\frac{1}{2}$
$\frac{1}{2}$	$\widehat{\frac{1}{2}}$
$\frac{3}{4}$	$\frac{1}{4}$
1	0

Definición 7.5. Se define un Álgebra de **I**ncertidumbre del **P**unto **M**edio (en adelante álgebra IPM), como la cuádrupla $(\Phi,\ +,\ \cdot,\ ^{-})$. Álgebra de Incertidumbre del Punto Medio

7.2.1 Estructura del Álgebra IPM: Un Álgebra de De Morgan

Teorema 7.1. La suma y el producto son conmutativos.

$$\forall a, b \in \Phi, a + b = b + a \ \text{ y } \ a \cdot b = b \cdot a$$

Demostración: Definición 7.2 y definición 7.3.

Teorema 7.2. La suma y el producto son asociativos.

$$\forall a, b, c \in \Phi, (a + b) + c = a + (b + c) \ \text{ y } \ (a \cdot b) \cdot c = a \cdot (b \cdot c)$$

Demostración: Definición 7.2 y definición 7.3.

Teorema 7.3. 0 es el único elemento neutro para la suma, 1 es el único elemento neutro para el producto, 0 es un elemento absorbente para el producto:

$$\forall a \in \Phi, a \cdot 0 = 0 = 0 \cdot a \ \text{ y } \ a + 1 = 1 = 1 + a$$

Demostración: Definición 7.2, definición 7.3 y teorema 7.1.

Teorema 7.4. El producto es distributivo respecto de la suma.

$$a \cdot (b + c) = (a \cdot b) + (a \cdot c), \quad a + (b \cdot c) = (a + b) \cdot (a + c)$$

Demostración: Definición 7.2 y definición 7.3.

Teorema 7.5. Φ es un retículo distributivo.

Demostración: Teorema 7.1, teorema 7.2, teorema 7.3 y teorema 7.4.

Definición 7.6. Un **Álgebra de De Morgan** es, por definición [RW92, Tri80], un retículo distributivo provisto de una negación llamada por abuso de lenguaje "complemento", que verifica las leyes de idempotencia y de *De Morgan*.

Definición 7.7. Un **Álgebra de Boole** [PG93], es un álgebra de *De Morgan* en la que el complemento lo es de verdad. Por ejemplo, sea $P(\mathcal{U})$ el conjunto de partes del universo $\mathcal{U}$. Escojamos la unión como operación suma, la intersección como operación producto y la complementación como operación monaria. Entonces $(P(\mathcal{U}), \cup, \cap, ^{-})$ tiene estructura de álgebra de *Boole*, ya que la operación complementación cumple: $\forall A \in P(\mathcal{U}),\ A \cup \overline{A} = \mathcal{U}$ y $A \cap \overline{A} = \emptyset$.

Teorema 7.6. (De idempotencia). $\forall a \in \Phi,\ a + a = a$ y $a \cdot a = a$

Demostración: Definición 7.2 y definición 7.3.

Teorema 7.7. (De De Morgan). Si a y b son dos elementos de Φ, $\overline{a + b} = \overline{a} \cdot \overline{b}$ y $\overline{a \cdot b} = \overline{a} + \overline{b}$

Demostración: Definición 7.2, definición 7.3 y definición 7.4.

Teorema 7.8. La cuádrupla $(\Phi, +, \cdot, ^{-})$ tiene estructura de álgebra de *De Morgan*.

Demostración: Definición 7.6, teorema 7.5, teorema 7.6 y teorema 7.7.

7.2.2 Teoremas Comunes del Álgebra IPM al Álgebra de Boole

Teorema 7.9. Para todo elemento $a \in \Phi$, $\overline{\overline{a}} = a$.

Demostración: Definición 7.4.

Teorema 7.10. Los elementos 0 y 1 son complementarios: $\overline{0} = 1$ y $\overline{1} = 0$. El elemento 1 es absorbente para la suma: $\forall a \in \Phi,\ a + 1 = 1$.

Demostración. Definición 7.2, definición 7.3 y definición 7.4.

Teorema 7.11. (De absorción). En una suma de productos, cada término absorbe a sus múltiplos. Así:

$$\begin{aligned} a + (a \cdot b) &= a && \text{(absorción por } a\text{)} \\ a \cdot b &= a \cdot b + (a \cdot b \cdot c) + (a \cdot b \cdot \overline{c} \cdot d) && \text{(absorción por } a \cdot b\text{)} \end{aligned}$$

Demostración:

$$\begin{aligned} a + (a \cdot x) &= (a \cdot 1) + (a \cdot x) && \text{(Teorema 7.3)} \\ &= a \cdot (1 + x) && \text{(Teorema 7.4)} \\ &= a \cdot 1 && \text{(Teorema 7.9)} \\ &= a && \text{(Teorema 7.3)} \end{aligned}$$

En la tabla 7.1 se resumen las leyes en las que difieren el álgebra de *Boole* y el álgebra de *De Morgan*, y por tanto, las leyes en las que difieren el álgebra de *Boole* y el álgebra IPM.

Leyes	**Álgebra de Boole**	**Álgebra de De Morgan**
Tercero Excluido $a + \overline{a} = 1$	Sí	No
Equivalencia entre incoherencia y contradicción $a + b = 0 \Leftrightarrow a \leq \overline{b}$	Sí	Sólo $\Rightarrow$
No-contradicción $a \cdot \overline{a} = 0$	Sí	No

Tabla 7.1: Leyes no comunes al álgebra de *Boole*.

7.2.3 Reglas y Teoremas sobre Igualdades en un Álgebra IPM

Regla 7.1. Se puede sumar o multiplicar un mismo término a los dos miembros de una igualdad: $\forall a, b, c \in \Phi$,

$$\begin{aligned} a = b &\Rightarrow a + c = b + c \\ a = b &\Rightarrow a \cdot c = b \cdot c \end{aligned}$$

Demostración: Esta regla se deduce inmediatamente del hecho de que $+$ y $\cdot$ son aplicaciones de Φ^2 en Φ. Contrariamente a lo que ocurre en el álgebra clásica,

$$\begin{aligned} a + c = b + c &\quad \text{no implica} \quad a = b \\ a \cdot c = b \cdot c &\quad \text{no implica} \quad a = b \end{aligned}$$

Teorema 7.12. $\forall a, b, c \in \Phi$,

$$\left.\begin{array}{rcl} a+c & = & b+c \\ a \cdot c & = & b \cdot c \end{array}\right\} \Rightarrow \; a \; = \; b$$

Demostración: Sean $a, b, c \in \Phi$, siempre se tiene $(a+c) \cdot a \; = \; a \cdot a + a \cdot c \; = a + a \cdot c \; = \; a$. Supongamos que $a + c \; = \; b + c$ y que $a \cdot c \; = \; b \cdot c$, entonces:

$$(a+c) \cdot a = (b+c) \cdot a = a \cdot b + a \cdot c = a \cdot b + b \cdot c = b \cdot (a+c) =$$
$$= b \cdot (b+c) = b \cdot b + b \cdot c = b + b \cdot c = b.$$

En conclusión, bajo las hipótesis dadas, se obtiene $a \; = \; b$.

Regla 7.2. Dos igualdades se pueden sumar o multiplicar miembro a miembro: $\forall a, b, c, d \in \Phi$,

$$\left.\begin{array}{rcl} a & = & b \\ c & = & d \end{array}\right\} \Rightarrow \; a+c \; = \; b+d \qquad \left.\begin{array}{rcl} a & = & b \\ c & = & d \end{array}\right\} \Rightarrow \; a \cdot c \; = \; b \cdot d$$

Demostración: Esta regla se deduce fácilmente del hecho de que $+$ y $\cdot$ son aplicaciones de Φ^2 en Φ

Regla 7.3. Dos elementos son iguales si, y sólo si, sus complementos son iguales:

$$\forall a, b \in \Phi, \; a \; = \; b \iff \overline{a} \; = \; \overline{b}$$

Demostración: Esta regla se deduce inmediatamente del hecho de que la complementación es una aplicación de Φ en Φ y de que, $\forall a \in \Phi, \; \overline{\overline{a}} \; = \; a$

Teorema 7.13. $\forall a, b \in \Phi$,

$$\left.\begin{array}{r} a \cdot b \; = \; 1 \iff (a \; = \; 1 \text{ y } b \; = \; 1) \\ a + b \; = \; 0 \iff (a \; = \; 0 \text{ y } b \; = \; 0) \end{array}\right\} \quad \left.\begin{array}{r} a \cdot b \; = \; 0 \iff (a \; = \; 0 \text{ ó } b \; = \; 0) \\ a + b \; = \; 1 \iff (a \; = \; 1 \text{ ó } b \; = \; 1) \end{array}\right\}$$

Demostración: Definición 7.2 y definición 7.3.

7.3 Definición de la Relación de Satisfacción

A continuación, se define una función o relación de satisfacción para obtener el grado de satisfacción de un requisito SCTL $\mathcal{R}$ especificado en un estado E_j de un grafo MUS $\mathcal{M}$ –$\mathcal{R}_{E_j}$–. Dicha relación de satisfacción se basa en las proposiciones causales formuladas en la lógica SCTL: *"un requisito SCTL se satisface sii se satisface su premisa (condición de aplicabilidad) y en los estados indicados por su operador temporal, se satisface su consecuencia"*.

Definición 7.8. Sea $\Phi = \{0,\ \frac{1}{4},\ \widehat{\frac{1}{2}},\ \frac{1}{2},\ \frac{3}{4},\ 1\}$ el conjunto de grados de satisfacción definido en la sección 7.2. Sea $\mathcal{R}$ un requisito SCTL, y sea $\{\mathcal{E}'_{\mathcal{M}}\} = \{E_1, E_2, ..., E_n\}$ el conjunto de estados de un grafo MUS $\mathcal{M}$. Se define la **Relación de Satisfacción** como una función de SCTL $\times$ $\{\mathcal{E}'_{\mathcal{M}}\}$ en Φ –denotada por $\models (\mathcal{R}, E_j)$–, que obtiene el grado de satisfacción de $\mathcal{R}$ en un estado E_j del grafo MUS $\mathcal{M}$.

$\models$		
$SCTL \times \{\mathcal{E}'_{\mathcal{M}}\}$	$\longrightarrow$	Φ
$(\mathcal{R}, E_j)$	$\longmapsto$	$\varphi = \models (\mathcal{R}, E_j) \in \{0,\ \frac{1}{4},\ \widehat{\frac{1}{2}},\ \frac{1}{2},\ \frac{3}{4},\ 1\}$

La tabla 7.2 resume el significado de cada uno de los grados de satisfacción de un requisito SCTL[2].

$\varphi \in \Phi$	**Grado de satisfacción de R**
0	NO se satisface.
$\frac{1}{4}$	NO PUEDE satisfacerse.
$\widehat{\frac{1}{2}}$	NO PUEDE satisfacerse y NO PUEDE NO satisfacerse.
$\frac{1}{2}$	PUEDE satisfacerse y PUEDE NO satisfacerse.
$\frac{3}{4}$	NO PUEDE NO satisfacerse.
1	SI se satisface.

Tabla 7.2: Grados de satisfacción de un requisito SCTL.

Definición 7.9. Dados dos elementos $a, b \in \Psi = \{0, \frac{1}{2}, 1\}$, se define la operación interna **Satisfacción Atómica** –denotada por $\vdash (a, b) = a \vdash b$–, como sigue:

$\vdash$		
Ψ x Ψ	$\longrightarrow$	Ψ
$(a,\ b)$	$\longmapsto$	$a \vdash b$

$\vdash$	0	$\frac{1}{2}$	1
0	1	$\frac{1}{2}$	0
$\frac{1}{2}$	1	1	1
1	0	$\frac{1}{2}$	1

Definición 7.10. Tal y como se explicó en el capítulo 5, un requisito SCTL $\mathcal{R}$ puede especificar las acciones $a_i \in \Lambda'$ en cada estado E_j de un grafo MUS $\mathcal{M}$ –especificación denotada por $\mathcal{S}_{\mathcal{R}}(a_i, E_j)$–, como elementos del conjunto Ψ. Se define el **Grado de Satisfacción de**

[2]En el capítulo 8 se ilustra con un ejemplo cada uno de los grados de satisfacción definidos.

$\boldsymbol{\mathcal{S}_{\mathcal{R}}(a_i, E_j)}$ –denotado por $\models \mathcal{S}_{\mathcal{R}}(a_i, E_j)$– como un elemento del conjunto Ψ, que se obtiene aplicando la función satisfacción atómica a dicha especificación y a la especificación de la acción correspondiente en el grafo:

$$\vdash (\mathcal{S}_{\mathcal{R}}(a_i, E_j),\ E_j[a_i]) \ = \ \vdash (\mathcal{S}_{\mathcal{R}}(a_i, E_j),\ d[a_i][j][E_{unsp}])$$

Definición 7.11. Dados dos elementos $a, b \in \Psi$, se define la operación externa denominada **Unión** [3] –denotada por $a \cup b$–, como sigue:

$\bigcup$		
Ψ x Ψ	$\longrightarrow$	$\Psi \bigcup ?$
$(a,\ b)$	$\longmapsto$	$a \cup b$

$\bigcup$	0	$\frac{1}{2}$	1
0	0	0	?
$\frac{1}{2}$	0	$\frac{1}{2}$	1
1	?	1	1

Definición 7.12. Dados dos elementos $a, b \in \Phi$, se define la operación interna **Causal** –denotada por $\rightarrow (a, b) \ = \ a \rightarrow b$–, como sigue:

$\rightarrow$		
Φ x Φ	$\longrightarrow$	Φ
$(a,\ b)$	$\longmapsto$	$a \rightarrow b$

$\rightarrow$	0	$\frac{1}{4}$	$\widehat{\frac{1}{2}}$	$\frac{1}{2}$	$\frac{3}{4}$	1
0	$\widehat{\frac{1}{2}}$	$\widehat{\frac{1}{2}}$	$\widehat{\frac{1}{2}}$	$\widehat{\frac{1}{2}}$	$\widehat{\frac{1}{2}}$	$\widehat{\frac{1}{2}}$
$\frac{1}{4}$	$\widehat{\frac{1}{2}}$	$\widehat{\frac{1}{2}}$	$\widehat{\frac{1}{2}}$	$\widehat{\frac{1}{2}}$	$\widehat{\frac{1}{2}}$	$\widehat{\frac{1}{2}}$
$\widehat{\frac{1}{2}}$	$\widehat{\frac{1}{2}}$	$\widehat{\frac{1}{2}}$	$\widehat{\frac{1}{2}}$	$\widehat{\frac{1}{2}}$	$\widehat{\frac{1}{2}}$	$\widehat{\frac{1}{2}}$
$\frac{1}{2}$	$\frac{1}{4}$	$\frac{1}{4}$	$\widehat{\frac{1}{2}}$	$\frac{1}{2}$	$\frac{3}{4}$	$\frac{3}{4}$
$\frac{3}{4}$	$\frac{1}{4}$	$\frac{1}{4}$	$\widehat{\frac{1}{2}}$	$\frac{1}{2}$	$\frac{3}{4}$	$\frac{3}{4}$
1	0	$\frac{1}{4}$	$\widehat{\frac{1}{2}}$	$\frac{1}{2}$	$\frac{3}{4}$	1

El grado de satisfacción de un requisito SCTL en un estado de un modelo de estados subespecificados se obtiene mediante la aplicación de la función satisfacción definida a continuación:

1. Si el requisito es un requisito atómico:
   (a) Si la premisa es *falsa* o está *subespecificada*, el grado de satisfacción de dicho requisito[4] es $\widehat{\frac{1}{2}}$.
   (b) En caso contrario –la premisa es *verdadera*–, el grado de satisfacción se obtiene mediante el producto de la operación *satisfacción atómica* aplicada a la especificación de la acción correspondiente al requisito atómico, y a la especificación de dicha acción en cada uno de los estados de aplicabilidad del operador temporal de dicho requisito.

[3] Esta operación ya ha sido utilizada en el algoritmo 6.6.

[4] A partir de ahora, se entenderán por requisitos atómicos únicamente aquellos en los que la premisa es la constante *verdadero*, ya que, el resto carecen de sentido por especificar una premisa que no puede satisfacerse.

2. En caso contrario –el requisito no es atómico–, el grado de satisfacción se obtiene aplicando la operación *causal* a una nueva recursión de la operación *satisfacción* aplicada a la premisa en el mismo estado, y al producto de la operación *satisfacción* aplicada a la consecuencia en cada uno de los estados de aplicabilidad del operador temporal del requisito a satisfacer.

A continuación, se define formalmente la relación de satisfacción:

Definición 7.13. Sea $\mathcal{R} = \mathcal{P} \bigoplus \mathcal{C}$ un requisito SCTL, donde $\mathcal{P}$ es su premisa, $\bigoplus$ es su operador temporal y $\mathcal{C}$ es su consecuencia. Sea E_j un estado de un grafo MUS $\mathcal{M}$ –$E_j \in \{\mathcal{E}'_{\mathcal{M}}\}$, $\{\mathcal{E}'_{\mathcal{M}}\} = \{E_1, E_2, ..., E_n\}$–, y sea $\Lambda' = \{a_1, a_2, ..., a_m\}$ el conjunto de acciones. Se define la operación **Satisfacción** –denotada por $\models (\mathcal{R}, E_j) = \models (\mathcal{R}_{E_j}) \in \Phi$–, por:

$$\models (\mathcal{R}, E_j) \;=\; \begin{cases} \prod\limits_k \vdash (\mathcal{S}_{\mathcal{R}}(a_i, E_k), E_k[a_i]) & \text{Si } \mathcal{R} \text{ es atómico, } \mathcal{R} = \; true \bigoplus [\neg]a_i \\ (\models (\mathcal{P}, E_j)) \rightarrow (\prod\limits_k \models (\mathcal{C}, E_k)) & \text{en otro caso.} \end{cases}$$

$\forall\, E_k \in \perp (\mathcal{R}, E_j)$, siendo $\perp (\mathcal{R}, E_j)$ el conjunto de estados de aplicabilidad definido por $\bigoplus$.

Definición 7.14. Sean $\mathcal{R}_1$ y $\mathcal{R}_2$ dos requisitos SCTL, y sea E_j un estado de un grafo MUS $\mathcal{M}$. Se define la operación satisfacción sobre requisitos SCTL unidos mediante los operadores lógicos como sigue[5]

$$\begin{aligned} \models ((\mathcal{R}_1 \;\wedge\; \mathcal{R}_2), E_j) \;&\triangleq\; \models (\mathcal{R}_1, E_j) \wedge \models (\mathcal{R}_2, E_j) \\ \models ((\mathcal{R}_1 \;\vee\; \mathcal{R}_2), E_j) \;&\triangleq\; \models (\mathcal{R}_1, E_j) \vee \models (\mathcal{R}_2, E_j) \\ \models (\neg R_1, E_j) \;&\triangleq\; \neg \models (R_1, E_j) \end{aligned}$$

Definición 7.15. Se define una **Relación de Equivalencia** entre requisitos SCTL, tal que, dados dos requisitos SCTL $\mathcal{R}_1$ y $\mathcal{R}_2$ se dice que dichos requisitos son **equivalentes** –denotado por $\mathcal{R}_1 \eqcirc \mathcal{R}_2$– sii tienen el mismo grado de satisfacción en cualquier estado de cualquier grafo MUS.

$$\mathcal{R}_1 \eqcirc \mathcal{R}_2, \text{ sii } \forall\, E_j \text{ de todo grafo MUS } \mathcal{M},\; \models (\mathcal{R}_1, E_j) \;=\; \models (\mathcal{R}_2, E_j)$$

Aplicando las propiedades definidas para el álgebra IPM y la definición 7.14 se obtienen requisitos equivalentes, como los mostrados en el ejemplo 7.1, $\mathcal{R}_1 \eqcirc \mathcal{R}_2$.

Ejemplo 7.1 Ejemplo de requisitos equivalentes.

$\mathcal{R}_1$	$\mathcal{R}_2$
$(true \Rightarrow a \;\wedge\; true \Rightarrow b)$	$\neg(true \Rightarrow \neg a \;\vee\; true \Rightarrow \neg b)$

En este punto es donde toma una mayor relevancia la estructura del álgebra IPM definida, ya que, la función satisfacción obtiene como resultado un elemento del conjunto Φ; y la relación de

[5] $\wedge$: Operación *producto*. $\vee$: Operación *suma*.

equivalencia se define en función de la igualdad de dos elementos de dicho conjunto. Es posible, por tanto, aplicar todos los teoremas y reglas sobre igualdades obtenidos y estudiados para un álgebra de De Morgan, obteniendo así conjuntos de requisitos SCTL equivalentes.

7.3.1 Propiedades

Además de las propiedades y reglas ya estudiadas y demostradas para el álgebra de *De Morgan*, se proporcionan las siguientes propiedades de los requisitos SCTL:

Propiedad 7.1. Sean a y b dos elementos de Ψ tal que $\vdash (a, b) \in \mathcal{B} \triangleq \{0, 1\}$, entonces se verifica:

$$\neg \vdash (a, b) \;=\; \vdash (a, \neg b)^6$$

Propiedad 7.2. $\forall a, b, c \in \Phi$ se verifica:

$$a \to (b \wedge c) \;=\; (a \to b) \wedge (a \to b) \qquad\qquad a \to (b \vee c) \;=\; (a \to b) \vee (a \to b)$$

Propiedad 7.3. Sean a y b dos elementos de Φ tal que $a \to b \in \mathcal{B}$, entonces se verifica:

$$\neg(a \to b) \;=\; (a \to \neg b)$$

Propiedad 7.4. Sean $\mathcal{R}_1 = \neg(\mathcal{P} \bigoplus \mathcal{C})$ y $\mathcal{R}_2 = \mathcal{P} \bigoplus \neg(\mathcal{C})$ dos requisitos SCTL, y sea E_j un estado del grafo MUS $\mathcal{M}$ tal que $\models (\mathcal{R}_1, E_j) \in \mathcal{B}$, entonces:

$$\models (\mathcal{R}_1, E_j) = \models (\mathcal{R}_2, E_j)$$

Propiedad 7.5. Sean $\mathcal{R}_1, \mathcal{R}_2$ y $\mathcal{R}_3$ requisitos SCTL, entonces:

$$(\mathcal{R}_1 \bigoplus (\mathcal{R}_2 \wedge \mathcal{R}_3)) \;=\; ((\mathcal{R}_1 \bigoplus \mathcal{R}_2) \;\wedge\; (\mathcal{R}_1 \bigoplus \mathcal{R}_3))$$
$$(\mathcal{R}_1 \bigoplus (\mathcal{R}_2 \vee \mathcal{R}_3)) \;=\; ((\mathcal{R}_1 \bigoplus \mathcal{R}_2) \;\vee\; (\mathcal{R}_1 \bigoplus \mathcal{R}_3))$$

7.3.2 Orden en el Álgebra IPM

Definición 7.16. En un álgebra IPM $\{\Phi,\ \cdot,\ +,\ ^{-}\}$, se define una **Relación de Orden** "menor o igual" del modo siguiente: sean a y b dos elementos de Φ, se dice que el elemento a es menor o igual que el elemento b cuando $a \cdot b \;=\; a$. $\forall a, b \in \Phi,\ a \leq b \iff a \cdot b \;=\; a$. El símbolo $\geq$ (mayor o igual) también se utiliza como en álgebra clásica, $a \geq b \iff b \leq a$.

Observación. $\forall a \in \Phi$, se tiene $0 \cdot a \;=\; 0$, luego $0 \leq a$, y $a \cdot 1 \;=\; a$, luego $a \leq 1$. Por consiguiente, 0 es menor que cualquier elemento de Φ y 1 es mayor que cualquier elemento de Φ. En particular, siempre se tiene $0 \leq 1$.

Teorema 7.14. En un álgebra IPM, la relación $\leq$ es una relación de orden. La estructura $(\Phi, \leq)$ se denomina un *orden*, o más correctamente, un *conjunto ordenado*.

Demostración: Sean $a,\ b$ y c tres elementos de Φ:

[6]Notación: $\neg$: Operación complementación o negación.

$$a \cdot a = a \Rightarrow a \leq a \qquad \textit{(Reflexividad)}$$
$$(a \leq b \text{ y } b \leq a) \Rightarrow (a \cdot b = a \text{ y } b \cdot a = b) \Rightarrow a = b \qquad \textit{(Antisimetría)}$$
$$(a \leq b \text{ y } b \leq c) \Rightarrow a = a \cdot b = a \cdot (b \cdot c) = (a \cdot b) \cdot c = a \cdot c \Rightarrow a \leq c \qquad \textit{(Transitividad)}$$

La relación de orden definida en el álgebra IPM, se corresponde con una relación de orden en el grado de satisfacción. El 0 es el menor grado de satisfacción de un requisito, mientras que el 1 es el mayor. De la misma manera, los valores $\widehat{\frac{1}{2}}$ y $\frac{1}{2}$ son puntos medios en dicha relación de orden; mientras que el primero está igual de *lejos* de los dos extremos sin poder alcanzar ninguno de los dos, el segundo está igual de *cerca* de ambos, pudiendo alcanzar cualquiera de los dos valores extremos. Esta es la razón por la que se adoptó el nombre de álgebra de **I**ncertidumbre del **P**unto **M**edio.

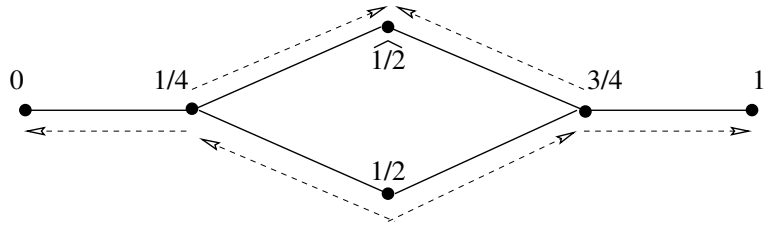


Figura 7.1: Álgebra de Incertidumbre del Punto Medio.

La metodología SCTL-MUS no proporciona un algoritmo concreto para la verificación mediante la demostración de teoremas, sino que proporciona la base matemática necesaria, mediante la obtención de una estructura de álgebra de *De Morgan*, para desarrollar dicha técnica de verificación. En el siguiente de capítulo se desarrolla un algoritmo de verificación basado en *model checking*.

Capítulo 8

Verificación SCTL-MUS

8.1 Introducción

La estructura de álgebra de De Morgan del conjunto de grados de satisfacción definido para los requisitos SCTL, permite realizar tareas de verificación mediante la técnica de demostración de teoremas. La utilización combinada de SCTL y MUS posibilita, además, la verificación de requisitos SCTL mediante *model checking*.

La verificación de un requisito SCTL se realiza a nivel de estado. Es decir, dado un grafo MUS, es posible verificar un requisito SCTL en cada uno de los estados del mismo. Por tanto, al formular un requisito SCTL, el usuario puede, además, especificar el conjunto de estados en los que quiere que se satisfaga dicho requisito. Esto es equivalente a enriquecer SCTL, dotándole de nuevos operadores temporales como: $Siempre\ (\Box),\ Alguna\ vez\ (\Diamond),\ Hasta(\mathcal{U})$...

En SCTL es posible formular dos tipos de requisitos en función de los estados en los cuales dicho requisito debe satisfacerse:

- **Invarianzas**: Son requisitos que deben satisfacerse siempre, durante toda la vida del sistema. Por tanto, un requisito SCTL formulado como invarianza se satisface sii se satisface en todos los estados del sistema.

- **Finalidades**: Son requisitos que representan los objetivos del sistema, es decir, requisitos que deben satisfacerse en al menos un estado de cada camino del grafo MUS donde se especifica.

En cualquier caso, la definición de unos u otros operadores para especificar en qué estados del sistema debe satisfacerse el requisito especificado, es independiente del desarrollo de un algoritmo que obtenga el grado de satisfacción de un requisito en un estado dado de un grafo MUS.

8.2 Interpretación de la Relación de Satisfacción

Dado un sistema cuyo comportamiento se representa mediante un grafo MUS, es posible obtener el grado de satisfacción de un requisito SCTL $\mathcal{R}$ en cualquier estado E_j del mismo. Para ello, basta con aplicar la función $\models (\mathcal{R}, E_j)$, obteniendo un resultado $\varphi \in \Phi$, correspondiente al grado de satisfacción de dicho requisito en el estado E_j.

La interpretación de cada grado de satisfacción se muestra en la tabla 7.2. De dicha tabla se extrae que, un requisito SCTL, $\mathcal{R} = \mathcal{P} \bigoplus \mathcal{C}$, se satisface en un estado de un grafo MUS que representa el prototipo actual del sistema especificado, sii en dicho estado se satisface su premisa, y en los estados de aplicabilidad indicados por su operador temporal $\overline{\mathcal{R}}[0] \in \Theta$ –calculados mediante el algoritmo 6.3–, se satisface su consecuencia.

A continuación, se muestra un ejemplo ilustrativo de cada uno de los grados de satisfacción definidos. Para ello, se va a evaluar el grado de satisfacción del requisito $\mathcal{R}_{grad}$, mostrado a continuación, en cada uno de los estados del grafo MUS $\mathcal{M}_{grad}$ de la figura 8.1.

req $\mathcal{R}_{grad}$ is
$(true \Rightarrow a_1) \;\Rightarrow\; (true \Rightarrow a_2)$
endreq

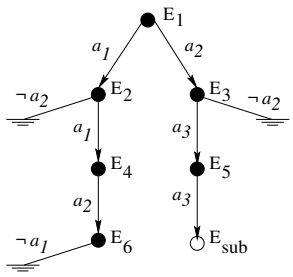


Figura 8.1: Modelo de estados subespecificados $\mathcal{M}_{grad}$.

- $\models (\mathcal{R}_{grad}, E_1)$:

  En el estado E_1 se satisface la premisa –$\models (true \Rightarrow a_1, E_1) = 1$–, ya que, en dicho estado la acción a_1 se especifica como posible –$E_1[a_1] = 1$– y $\mathcal{S}_{\mathcal{R}_{grad}}(a_1, E_1) = 1$. Además, la consecuencia –$true \Rightarrow a_2$– también se satisface en dicho estado, que es el único estado de aplicabilidad del requisito $\mathcal{R}_{grad}$, ya que, su premisa y consecuencia están unidas por el operador temporal *A la vez* –$\Rightarrow$–. Por tanto, el requisito $\mathcal{R}_{grad}$ está totalmente especificado en el estado E_1, siendo su grado de satisfacción *verdadero*, $\models (\mathcal{R}_{grad}, E_1) = 1$.

- $\models (\mathcal{R}_{grad}, E_2)$:

  En el estado E_2 se satisface la premisa de $\mathcal{R}_{grad}$, pero no su consecuencia, ya que $E[a_2] = 0$. El requisito $\mathcal{R}_{grad}$ está totalmente especificado en dicho estado, y su grado de satisfacción es *falso*, $\models (\mathcal{R}_{grad}, E_2) = 0$.

- $\models (\mathcal{R}_{grad}, E_5)$:

  $\mathcal{R}_{grad}$ está totalmente subespecificado en el estado E_5, ya que, $E_5[a_1] = \frac{1}{2}$ y $E_5[a_2] = \frac{1}{2}$. Por tanto, $\models (\mathcal{R}_{grad}, E_5) = \frac{1}{2}$.

En los estados E_3 y E_4, $\mathcal{R}_{grad}$ está parcialmente subespecificado, ya que en ambos no está especificada su premisa –$E_3[a_1] = E_4[a_i] = \frac{1}{2}$–, pero sí su consecuencia –$E_3[a_2] = 0$ y $E_4[a_2] = 1$– Sin embargo, el grado de satisfacción de $\mathcal{R}_{grad}$ es diferente en dichos estados.

- $\models (\mathcal{R}_{grad}, E_3)$:

  En E_3 no se satisface la consecuencia, ya que $E_3[a_2] = 0$, por lo que, aunque se especificara la acción a_1 como posible en dicho estado, perdiendo así su subespecificación, el grado de satisfacción de $\mathcal{R}_{grad}$ nunca podría ser *verdadero*, ya que no se satisface su consecuencia. Si $E_3[a_1] = 1$, entonces $\models (\mathcal{R}_{grad}, E_3) = 0$, y si $E_3[a_1] = 0$, entonces $\models (\mathcal{R}_{grad}, E_3) = \widehat{\frac{1}{2}}$. Por tanto, y para reflejar la subespecificación de $\mathcal{R}_{grad}$, es necesario asignarle un nuevo grado de satisfacción que indique que dicho requisito está actualmente subespecificado, pero que nunca podrá evolucionar hacia un grado de satisfacción verdadero, $\models (\mathcal{R}_{grad}, E_3) = \frac{1}{4}$.

- $\models (\mathcal{R}_{grad}, E_4)$:

  Por el contrario, en E_4 se satisface la consecuencia, ya que $E_4[a_2] = 1$, por lo que, si se pierde la subespecificación de la premisa –a_1–, el grado de satisfacción de $\mathcal{R}_{grad}$ sería *verdadero*. Si $E_3[a_1] = 1$, entonces $\models (\mathcal{R}_{grad}, E_3) = 1$, y si $E_3[a_1] = 0$, entonces $\models (\mathcal{R}_{grad}, E_3) = \widehat{\frac{1}{2}}$. Similarmente al caso anterior, es necesario definir un grado de satisfacción que refleje dicha subespecificación, $\models (\mathcal{R}_{grad}, E_4) = \frac{3}{4}$.

- $\models (\mathcal{R}_{grad}, E_6)$:

  Finalmente, en el estado E_6 no se satisface la premisa de $\mathcal{R}_{grad}$, ya que, en dicho estado se especifica la acción a_1 como no posible –$E_6[a_1] = 0$–. Es decir, no se satisface la condición de aplicabilidad de los requisitos SCTL, por lo que, no tiene sentido establecer un grado de satisfacción para $\mathcal{R}_{grad}$ en el estado E_6. Por tanto, se define un grado de satisfacción denominado *contradictorio*, que indica que el requisito $\mathcal{R}_{grad}$ ni se satisface ni se incumple, ya que, no satisface la condición de aplicabilidad, $\models (\mathcal{R}_{grad}, E_6) = \widehat{\frac{1}{2}}$.

Obsérvese que el grado de satisfacción de $\mathcal{R}_{grad}$ puede evolucionar en los estados en los que éste está total o parcialmente subespecificado. Mientras que en el estado E_5 puede evolucionar a cualquier grado de satisfacción –por estar totalmente subespecificado–, en los estados E_3 y E_4 sólo puede evolucionar a dos grados de satisfacción. En el estado E_3 puede evolucionar a un grado de satisfacción *falso* si su premisa se especifica como *verdadera*; o al grado de satisfacción *contradictorio*, si dicha premisa se especifica como *falsa*. Similarmente, en el estado E_4 puede evolucionar a un grado de satisfacción *verdadero* o *contradictorio*.

El ejemplo 8.1 resume el grado de satisfacción del requisito $\mathcal{R}_{grad}$ en cada uno de los estados del grafo MUS $\mathcal{M}_{grad}$ mostrado en la figura 8.1.

Ejemplo 8.1 Ejemplo de los grados de satisfacción de un requisito SCTL.

Estado	$\mathcal{R}_{grad}$	$true \Rightarrow a_1$	$true \Rightarrow a_2$
E_1	1	1	1
E_2	0	1	0
E_3	$\frac{1}{4}$	$\frac{1}{2}$	0
E_4	$\frac{3}{4}$	$\frac{1}{2}$	1
E_5	$\frac{1}{2}$	$\frac{1}{2}$	$\frac{1}{2}$
E_6	$\widehat{\frac{1}{2}}$	0	$\frac{1}{2}$

8.3 Algoritmo de Verificación SCTL-MUS

El objetivo del algoritmo de verificación es obtener el grado de satisfacción de un requisito SCTL en un estado de un sistema cuyo comportamiento se modela mediante un modelo de estados subespecificados.

El algoritmo de verificación recibe como entradas el estado E_h del grafo MUS en el que se desea verificar el requisito, y el requisito SCTL en notación inversa $\overline{\mathcal{R}}$, a verificar. El algoritmo devuelve el valor correspondiente a la función *satisfacción*, $\models (\mathcal{R}, E_0)$, definida en el capítulo 7.

Para poder realizar la verificación de requisitos recursivos sin entrar en un bucle infinito se almacena, en cada estado del grafo MUS, una lista con información relativa a cada requisito que ha sido verificado sobre dicho estado, y el resultado de la verificación. De esta manera, cada vez que se verifique un requisito, se explorará dicha lista, y si existiera una entrada en ella para el requisito a verificar, se devolverá el resultado almacenado. En caso contrario, se procederá a la verificación propiamente dicha del requisito sobre el estado formulado.

Según la sintaxis SCTL definida, es posible que un requisito definido forme parte de otro utilizando simplemente su identificador precedido del símbolo $\uparrow$. Esto permite especificar requisitos con dependencias cruzadas. El algoritmo de verificación sólo despliega dicho requisito –sustituye el identificador por el requisito SCTL– si es estrictamente necesario. Esta operación se simboliza en el algoritmo por: **Insertar Requisito (Identificador)**.

Además, el algoritmo identifica todos los subrequisitos[1] que forman el requisito a verificar, *aprendiendo* el resultado de la verificación de cada uno de ellos en los estados correspondientes.

Cabe destacar que el algoritmo no necesita comprobar una condición de salida, ya que sólo se generan las llamadas necesarias. Esto es debido a que sólo se hace una nueva llamada cada vez que se detecta un nuevo subrequisito.

[1] Ver algoritmos B.3 y B.4.

En el pseudocódigo del algoritmo 8.1 se realizan llamadas recursivas innecesarias a dicho algoritmo. Esto es debido a las definiciones de las funciones *producto* y *suma* en el álgebra IPM, que se corresponden con obtener el elemento menor y mayor respectivamente. Por ello, para evaluar la operación suma ($\vee$) de varios elementos, sólo es necesario evaluar dicha operación hasta que uno de los elementos obtenido sea el mayor (1). Lo mismo ocurre con la operación producto $\wedge$ y el elemento menor (0). El algoritmo de verificación propuesto tiene en cuenta dichas características, aunque éstas no se han plasmado en el algoritmo 8.1 para simplificar su pseudocódigo.

Sin embargo, aunque es posible devolver un resultado sin tener que evaluar la totalidad de requisitos unidos por los operadores lógicos, la aplicación del algoritmo de verificación al resto de requisitos permite *aprender* el grado de satisfacción de dichos requisitos, ya que el resultado de la verificación se almacena en el grafo, tal y como se ha explicado anteriormente.

8.3.1 Descripción

A continuación, se describe con detalle cada uno de los pasos realizados por el algoritmo 8.1. Para facilitar el seguimiento del pseudocódigo de dicho algoritmo, cada explicación se acompaña del paso (entre corchetes) correspondiente al algoritmo 8.1:

[1] En primer lugar, tal y como se explicó en la sección anterior, el algoritmo comprueba si ya se ha realizado la verificación del requisito $\mathcal{R}$ en el estado formulado E_h. En caso afirmativo, devuelve el resultado almacenado en dicho estado.

[2] Si el requisito es un requisito atómico, se continúa en el paso **[8][a][i]**.

En caso contrario, se procede a la verificación del requisito propiamente dicha. La verificación se basa en la interpretación del primer *item*[2] del requisito –$\overline{\mathcal{R}}[0]$–.

[3] Si se corresponde con el símbolo $\uparrow$, se procede a desplegar el requisito cuyo identificador se encuentra en el siguiente *item* del requisito –$\overline{\mathcal{R}}[1]$–, obteniendo así un nuevo requisito $\mathcal{R}'$.

A continuación, se invoca de nuevo al algoritmo de verificación con el nuevo requisito obtenido $\mathcal{R}'$ y el mismo estado E_h.

[4] Si se corresponde con el operador lógico $\neg$, el resultado de la verificación se obtiene mediante la aplicación de la operación *complementación*, al resultado obtenido por el algoritmo de verificación aplicado sobre el requisito $\mathcal{R}$ sin el operador $\neg$, y el mismo estado E_h.

[5] En caso contrario, el requisito $\mathcal{R}$ está compuesto por dos subrequisitos $\mathcal{R}_{sub_1}$ y $\mathcal{R}_{sub_2}$, que se obtienen mediante una llamada al algoritmo de partición B.2. Dichos requisitos pueden estar unidos por un operador lógico o por un operador temporal.

[6] Si están unidos por el operador lógico *And*, el resultado de la verificación se corresponde con la aplicación de la operación *producto*, al resultado devuelto por el algoritmo de verificación sobre cada uno de los dos subrequisitos y el mismo estado E_h.

[2] Ver algoritmo B.1.

Algoritmo 8.1 Algoritmo de Verificación ($\overline{\mathcal{R}}$, E_h)

[1] Si ya ha existe $\models (\mathcal{R}, E_h)$, devolver dicho resultado;
[2] Si $\overline{\mathcal{R}}$ es atómico, ir al paso **[8][a][i]**;
[3] Si $\overline{\mathcal{R}}[0] = \uparrow$:
  [a] $\overline{\mathcal{R}}' =$ **Insertar Requisito** $(\overline{\mathcal{R}}[1])$;
  [b] $resultado =$ **Algoritmo de Verificación** $(\overline{\mathcal{R}}', E_h)$;
  [c] Devolver $resultado$;
[4] Si $\overline{\mathcal{R}}[0] = \neg$:
  [a] $resultado = \neg$ **Algoritmo de Verificación** $(\overline{\mathcal{R}}[1], E_h)$;
  [b] Devolver $resultado$;
[5] $\{\overline{\mathcal{R}}_{sub_1}, \overline{\mathcal{R}}_{sub_2}\} =$ **Algoritmo de Partición** $(\overline{\mathcal{R}})$;
[6] Si $\overline{\mathcal{R}}[0] = \wedge$:
  [a] $resultado =$ **Algoritmo de Verificación** $(\overline{\mathcal{R}}_{sub_1}, E_h) \wedge$
  Algoritmo de Verificación $(\overline{\mathcal{R}}_{sub_2}, E_h)$;
  [b] Devolver $resultado$;
[7] Si $\overline{\mathcal{R}}[0] = \vee$:
  [a] $resultado =$ **Algoritmo de Verificación** $(\overline{\mathcal{R}}_{sub_1}, E_h) \vee$
  Algoritmo de Verificación $(\overline{\mathcal{R}}_{sub_2}, E_h)$;
  [b] Devolver $resultado$;
[8] Si $\overline{\mathcal{R}}[0] \in \Theta$:
  [a] Si $\overline{\mathcal{R}} = \bigoplus\ true\ [\neg]a_i$ –es un requisito atómico–:
    [i] $res\_premisa = 1$;
    [ii] $\{E_{aplic}\} = \perp (\mathcal{R}, E_h) =$ **Algoritmo de Aplicabilidad** $(\overline{\mathcal{R}}, E_h)$;
    [iii] $\forall\ E_j \in \{E_{aplic}\},\ res\_consec = res\_consec \wedge \vdash (\mathcal{S}_{\mathcal{R}}(a_i, E_j), d[a_i][j][E_{sub}])$;
    [iv] Ir al paso f;
  [b] $res\_premisa =$ **Algoritmo de Verificación** $(\overline{\mathcal{R}}_{sub_1}, E_h)$;
  [c] Si $res\_premisa \leq \widehat{\frac{1}{2}}$, Devolver $\widehat{\frac{1}{2}}$;
  [d] $\{E_{aplic}\} = \perp (R, E_h) \equiv$ **Algoritmo de Aplicabilidad** $(\overline{\mathcal{R}}, E_h)$;
  [e] $\forall\ E_j \in \{E_{aplic}\},\ res\_consec = res\_consec \wedge$ **Algoritmo de Verificación** $(\overline{\mathcal{R}}_{sub_2}, E_j)$;
  [f] Devolver $(res\_premisa \rightarrow res\_consec)$;

[7] En caso de estar unidos por el operador lógico *Or*, el algoritmo se comporta de manera similar al caso anterior, pero aplicando la operación *suma*.

[8] En otro caso, los dos subrequisitos están unidos por un operador temporal. Es decir, el requisito consta de una premisa y una consecuencia. En primer lugar, se obtiene el grado de satisfacción de la premisa –*res_premisa*–. En caso de verificar un requisito atómico, dicho grado es *true*, mientras que en caso contrario, el resultado se obtiene mediante una llamada al algoritmo de verificación con la premisa –$\mathcal{R}_{sub_1}$–, y el mismo estado E_h donde

se especifica el requisito $\mathcal{R}$. Si el grado de satisfacción de la premisa es mayor que $\widehat{\frac{1}{2}}$, se obtienen los estados de aplicabilidad donde debe satisfacerse la consecuencia. Para ello, se realiza una llamada al algoritmo de aplicabilidad 6.3. Una vez obtenidos los estados de aplicabilidad, se obtiene el grado de satisfacción de la consecuencia –*res_consec*–. Ver los pasos **[8][a][iii]** y **[8][e]**.

[8][f] El resultado de la verificación se obtiene aplicando la operación *causal* (ver definición 7.12) al grado de satisfacción de la premisa y al grado de satisfacción de la consecuencia.

Obsérvese que para obtener la especificación de una acción a_i en un estado E_j del modelo de estados subespecificados, basta con obtener el valor del elemento $d[a_i][j][n+1]$ correspondiente a la matriz de adyacencia que representa el grafo subespecificado de dicha acción.[3]

Por otra parte, el algoritmo 8.1 obtiene el grado de satisfacción de un requisito SCTL aunque éste no se corresponda con una traza del requisito en forma normal positiva. Esto es debido a que es el propio algoritmo de verificación el que extrae la traza del requisito verificada en cada ejecución del mismo, lo que permite obtener una especificación más precisa en el caso de requisitos unidos por el operador lógico $\vee$. De esta manera, se evita repetir las mismas iteraciones del algoritmo para las distintas trazas del requisito.

8.3.2 Ejemplo de Aplicación

A continuación, se describe un ejemplo ilustrativo del funcionamiento del algoritmo de verificación. En la figura 8.2 se muestra el grafo MUS del prototipo actual del sistema sobre el que se desea realizar la verificación y, a continuación, el requisito a verificar sobre el nodo raíz de dicho grafo (para simplificar la notación, los requisitos atómicos del tipo $true \Rightarrow a$ se han sustituido por a):

req A is $((a \wedge b) \Rightarrow \bigcirc r) \Rightarrow (d \wedge \uparrow B)$ **endreq**	**req B is** $a \Rightarrow \neg c$ **endreq**

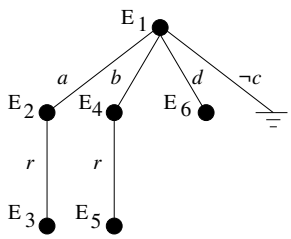


Figura 8.2: Grafo MUS del sistema.

En el ejemplo 8.2 se muestra el requisito a verificar en cada recursión del algoritmo, así como el número del estado del grafo MUS en el que se está verificando dicho requisito. En 12 recursiones

[3] Ver capítulo 4

se obtiene el resultado final. Cada recursión se acompaña de un número de orden. En este ejemplo todas las recursiones del algoritmo devuelven el valor $true$.

Ejemplo 8.2 Ejemplo del Algoritmo de Verificación.

Recursión	$\overline{\mathcal{R}}$	$\mathbf{E}_h$	**Resultado**
[1]	$\Rightarrow\Rightarrow \bigcirc \wedge abr \wedge d \uparrow B$	E_1	Recursión[2] $\rightarrow$ Recursión[8]
[2]	$\Rightarrow \bigcirc \wedge abr$	E_1	Recursión[3] $\rightarrow$((Recursión[6] $\wedge$ Recursión[7])
[3]	$\wedge ab$	E_1	Recursión[4] $\wedge$ Recursión[5]
[4]	a	E_1	$true$
[5]	b	E_1	$true$
[6]	r	E_2	$true$
[7]	r	E_4	$true$
[8]	$\wedge d \uparrow B$	E_1	Recursión[9] $\wedge$ Recursión[10]
[9]	d	E_1	$true$
[10]	$\uparrow B \Longrightarrow a\neg c$	E_1	Recursión[11] $\rightarrow$ Recursión[12]
[11]	a	E_1	$true$
[12]	$\neg c$	E_1	$true$

Parte IV

Fase de Diseño

Capítulo 9

Síntesis Incremental

9.1 Introducción

Mediante el algoritmo de verificación 8.1 se puede determinar el grado de satisfacción de un requisito SCTL en un estado de un sistema representado según el modelo de estados subespecificados. Además, mediante el algoritmo de traducción SCTL-MUS 6.6, se puede representar el comportamiento de un requisito SCTL mediante un grafo MUS.

Sin embargo, hasta ahora no se ha definido ningún mecanismo para obtener un modelo de estados de un sistema que debe satisfacer un conjunto de requisitos SCTL. Este es el objetivo del **algoritmo de síntesis**, que partiendo de un sistema que satisface un conjunto de requisitos SCTL $\{\mathcal{R}_1, \ldots, \mathcal{R}_s\}$ y cuyo comportamiento se expresa mediante un grafo MUS $\mathcal{M}$, sintetiza un grafo MUS $\mathcal{M}'$ que satisface, además, un nuevo requisito SCTL $\mathcal{R}_{s+1}$.

Esta tarea no es siempre posible, ya que, el requisito $\mathcal{R}_{s+1}$ puede ser inconsistente con un subconjunto de los requisitos que satisface el sistema representado por el grafo MUS $\mathcal{M}$. En este caso, el algoritmo de síntesis actúa como algoritmo detector de inconsistencias, mostrando el subconjunto de requisitos inconsistentes. Por el contrario, si es posible obtener un nuevo grafo $\mathcal{M}'$ que satisfaga la totalidad de requisitos especificados, éste se obtiene a través de una evolución del grafo original $\mathcal{M}$. A continuación, se resumen las posibles evoluciones de un grafo MUS, en función del resultado de la verificación de un nuevo requisito SCTL:

- Si el resultado es *verdadero*, el sistema no evoluciona, ya que satisface el nuevo requisito especificado.
- Si el resultado es *falso*, el sistema especificado contiene requisitos inconsistentes.
- Si el resultado es *subespecificado*, el sistema evoluciona dependiendo del grado de subespecificación obtenido:
  - Si el requisito está *totalmente subespecificado*, se puede modificar el sistema actual –perdiendo la subespecificación correspondiente a dicho requisito–, de manera que se sintetice un nuevo grafo MUS consistente con la totalidad de requisitos especificados.

- Si el requisito está *parcialmente subespecificado* y no puede ser *falso*, se actúa como en el caso anterior.
- Si el requisito está *parcialmente subespecificado* y no puede ser *verdadero*, no es posible sintetizar un modelo de estados MUS consistente con el conjunto total de requisitos especificados.

- En caso contrario, –el resultado de la verificación es *contradictorio*– no se satisface la condición de aplicabilidad del requisito, por lo que no se modifica el grafo MUS del sistema. Estos requisitos no se satisfacen, pero no imponen restricciones al sistema –no son aplicables–, por lo que el sistema no evoluciona.

9.2 Descripción General del Proceso de Síntesis Incremental

A continuación, se describen los pasos que deberá seguir el algoritmo de síntesis cuando se especifica un nuevo requisito $\mathcal{R}_{s+1} = \mathcal{P} \bigoplus \mathcal{C}$ en un estado E_j del grafo MUS $\mathcal{M}$ del sistema:

1. Comprobar si el nuevo requisito $\mathcal{R}_{s+1}$ ya se satisface en el estado E_j del grafo MUS del sistema. Para ello, bastará con hacer una serie de llamadas al algoritmo de verificación:

   Llamar al algoritmo de verificación con el requisito $\mathcal{P}$, premisa del requisito $\mathcal{R}_{s+1}$, y el estado E_j, donde dicho requisito debe satisfacerse. En función del resultado de esta verificación, se tienen tres situaciones diferentes:

   (a) $(\mathbf{0}, \frac{\mathbf{1}}{\mathbf{4}}, \widehat{\frac{\mathbf{1}}{\mathbf{2}}})$: El requisito $\mathcal{R}_{s+1}$ no se incumple en el estado E_j del grafo MUS $\mathcal{M}$, es decir, nunca podrá satisfacerse y nunca podrá no satisfacerse en dicho estado. Ir al paso [5].

   (b) $(\frac{\mathbf{1}}{\mathbf{2}}, \frac{\mathbf{3}}{\mathbf{4}})$: Llamar al algoritmo de verificación con el requisito $\mathcal{C}$, consecuencia del requisito $\mathcal{R}_{s+1}$, y los estados de aplicabilidad del requisito especificado. El resultado de dicha verificación será la operación *producto* de cada resultado, según la definición 7.13. En función del resultado de esta verificación, se obtienen cuatro situaciones diferentes:

      i. $(\mathbf{0}, \frac{\mathbf{1}}{\mathbf{4}})$: El requisito $\mathcal{R}_{s+1}$ está parcialmente subespecificado en el estado E_j, pero nunca podrá satisfacerse en dicho estado. Ir al paso [5].

      ii. $(\widehat{\frac{\mathbf{1}}{\mathbf{2}}})$: El requisito $\mathcal{R}_{s+1}$ está parcialmente subespecificado, pero no se incumple en el estado E_j del grafo MUS $\mathcal{M}$. Es decir, nunca podrá satisfacerse y nunca podrá no satisfacerse en dicho estado. Ir al paso [5].

      iii. $(\frac{\mathbf{1}}{\mathbf{2}})$: El requisito $\mathcal{R}_{s+1}$ está totalmente subespecificado en el estado E_j. En el caso de que el resultado del paso **(b)** sea $\frac{3}{4}$, la premisa está parcialmente subespecificada, de manera que no podrá no satisfacerse. Ir al paso [5].

      iv. $(\frac{\mathbf{3}}{\mathbf{4}}, \mathbf{1})$: El requisito $\mathcal{R}_{s+1}$ está parcialmente subespecificado en el estado E_j y nunca podrá no satisfacerse en dicho estado. Ir al paso [5].

(c) $(\mathbf{1})$: Llamar al algoritmo de verificación con el requisito $\mathcal{C}$, consecuencia del requisito $\mathcal{R}_{s+1}$, y sus estados de aplicabilidad, de manera similar al paso [1][b]. En función del resultado de esta verificación, se tienen seis situaciones diferentes:

i. $(\mathbf{0})$: El requisito $\mathcal{R}_{s+1}$ no se satisface en el estado E_j. El conjunto de requisitos $\{\mathcal{R}_1, ..., \mathcal{R}_s, \mathcal{R}_{s+1}\}$ son inconsistentes. Ir al paso [4].

ii. $(\frac{\mathbf{1}}{\mathbf{4}})$: La consecuencia del requisito $\mathcal{R}_{s+1}$ está parcialmente subespecificada en el estado E_j. Si se pierde la subespecificación el requisito $\mathcal{R}_{s+1}$ no se satisfará en el estado E_j del grafo MUS $\mathcal{M}$. Por tanto, el requisito $\mathcal{R}_{s+1}$ sólo puede no satisfacerse. Ir al paso [4].

iii. $(\widehat{\frac{\mathbf{1}}{\mathbf{2}}})$: La consecuencia del requisito $\mathcal{R}_{s+1}$ no se incumple en el estado E_j. Es decir, el requisito $\mathcal{R}_{s+1}$ nunca podrá satisfacerse y nunca podrá no satisfacerse en dicho estado. Ir al paso [5].

iv. $(\frac{\mathbf{1}}{\mathbf{2}})$: La consecuencia del requisito $\mathcal{R}_{s+1}$ está totalmente subespecificada en el estado E_j. Dado que la premisa se satisface, el requisito podrá satisfacerse o no, en función del resultado de la pérdida de subespecificación. Ir al paso [2].

v. $(\frac{\mathbf{3}}{\mathbf{4}})$: La consecuencia del requisito $\mathcal{R}_{s+1}$ está parcialmente subespecificada. El requisito $\mathcal{R}_{s+1}$ puede satisfacerse en dicho estado si se pierde la subespecificación de la consecuencia. Ir al paso [2].

vi. $(\mathbf{1})$: El requisito $\mathcal{R}_{s+1}$ se satisface en el estado E_j del grafo MUS $\mathcal{M}$. Ir al paso [3].

2. Sintetizar un nuevo grafo MUS $\mathcal{M}'$ que contenga el comportamiento del nuevo requisito especificado. Para ello, se solapará el grafo actual del sistema $\mathcal{M}$ con el grafo MUS $\mathcal{M}_{\mathcal{R}_{s+1}}$ correspondiente al requisito $\mathcal{R}_{s+1}$. Ir al paso [5].

3. El sistema $\mathcal{M}'$ satisface el conjunto de requisitos $\{\mathcal{R}_1, \ ... \ , \mathcal{R}_s, \mathcal{R}_{s+1}\}$. Ir al paso [5].

4. No es posible sintetizar un sistema que satisfaga el conjunto de requisitos especificados $\{\mathcal{R}_1, \ ... \ , \mathcal{R}_s, \ \mathcal{R}_{s+1}\}$.

5. Fin de la iteración del proceso de síntesis incremental.

9.3 Algoritmo de Síntesis: Una Primera Aproximación

9.3.1 Comparación con el Algoritmo de Traducción

El algoritmo de traducción SCTL-MUS expuesto en el capítulo 6 puede interpretarse como un algoritmo de síntesis. Dicho algoritmo traduce la especificación de un requisito SCTL a un modelo de estados subespecificados que representa el comportamiento del mismo. En dicha traducción se parte de un modelo de estados totalmente subespecificado al que se le van añadiendo estados a medida que es necesario.

El algoritmo de síntesis debe añadir el comportamiento de un requisito SCTL a un modelo de estados en fase de desarrollo. Esto no supone ningún inconveniente, ya que, el algoritmo de traducción funciona exactamente igual en dicho caso. Es más, cuando se traducen requisitos SCTL unidos por el operador lógico *And*, y se aborda la traducción del segundo requisito unido por dicho operador, el modelo de estados no está subespecificado, ya que contiene el comportamiento del primer requisito.

Por tanto, si se quiere añadir un nuevo requisito SCTL en un estado E_h de un modelo de estados obtenido anteriormente, bastaría con obtener la traducción del nuevo requisito en dicho estado E_h. Sin embargo, la obtención de un algoritmo de síntesis a partir del algoritmo de traducción no es tan inmediata debido a varios motivos:

- El modelo de estados correspondiente a la traducción de un requisito SCTL muestra el comportamiento general de dicho requisito. Sin embargo, dicho comportamiento se particulariza para cada sistema. Por ejemplo, el modelo de estados del requisito atómico correspondiente al operador temporal *Antes* contiene un único estado anterior al estado inicial donde se especifica el requisito. Por el contrario, si este modelo de estados se particulariza en un sistema en concreto, es posible que aparezcan un número indeterminado de estados anteriores o estados de aplicabilidad del requisito.

  En la figura 9.1 se pone de manifiesto lo expuesto anteriormente. El modelo de estados de la izquierda muestra el grafo MUS del requisito atómico $\mathcal{R}_{at} = true \Rightarrow \odot a$ con un único estado de aplicabilidad, ya que ha sido obtenido por el algoritmo de traducción. Este requisito se particulariza en el modelo de estados de la derecha, en el que existen dos estados de aplicabilidad (anteriores) al estado E_h donde se especifica dicho requisito.

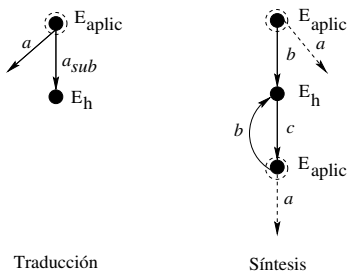


Figura 9.1: Grafo MUS del requisito $\mathcal{R}_{at} = true \Rightarrow \odot a$

- El algoritmo de traducción SCTL-MUS crea nuevos estados en el sistema cuando no existen estados de aplicabilidad. En principio, la creación de nuevos estados cada vez que esto sea necesario, permite sintetizar un sistema con el mayor grado de libertad, es decir, con el mayor grado de subespecificación posible, de manera que muestre únicamente el comportamiento especificado por el usuario en el requisito SCTL. Esto, unido a que en la síntesis es posible identificar un requisito SCTL como una invarianza o como una finalidad puede suponer la síntesis de sistemas con infinitos estados. Por ejemplo, si se espe-

cifican como invarianzas los requisitos SCTL: $\mathcal{R}_1 = (true \Rightarrow a) \Rightarrow \bigcirc(true \Rightarrow b)$ y $\mathcal{R}_2 = (true \Rightarrow b) \Rightarrow \bigcirc(true \Rightarrow a)$. Ver figura 9.2.

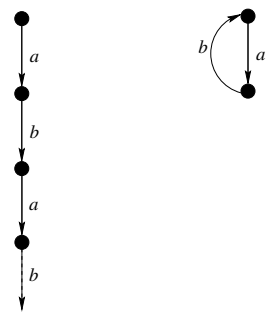


Figura 9.2: Sistema con infinitos estados.

La solución propuesta en el capítulo 6 permite especificar que en un estado de un grafo MUS, éste se comporta según un requisito SCTL, lo que permite desplegar el grafo sólo cuando sea necesario. Sin embargo, en la síntesis de un sistema determinado, es posible, además, reutilizar estados ya existentes en el sistema –creando bucles en el grafo–, siempre que el grafo resultante satisfaga la totalidad de requisitos especificados.

- La filosofía de cada algoritmo es distinta. Mientras que el algoritmo de traducción pretende mostrar el comportamiento del requisito SCTL que representa, el algoritmo de síntesis tiene como objetivo sintetizar un sistema que satisfaga un conjunto de requisitos SCTL. Es decir, dado un sistema que satisface la premisa de un requisito SCTL, hacer que se satisfaga su consecuencia en todos los estados de aplicabilidad de dicho requisito.

Es necesario, por tanto, modificar el algoritmo de traducción y adaptarlo a las características del algoritmo de síntesis. En primer lugar, en vez de crear nuevos estados cuando sea necesario, se optará por buscar algún estado, ya existente en el sistema, que satisfaga las condiciones del nuevo estado impuestas por el requisito. Sin embargo, la utilización de estados ya creados, impone nuevas restricciones al sistema que no han sido especificadas por el usuario, o mejor dicho, que están subespecificadas. En la siguiente sección se describen las posibles pérdidas de subespecificación introducidas en un grafo MUS, debido a la reutilización de estados en el proceso de síntesis.

9.3.2 Reutilización de Estados

La figura 9.3 muestra un grafo MUS $\mathcal{M}_{sin}$, en cuyo estado inicial E_0 se ha especificado el requisito $\mathcal{R}_{sin}$. Su premisa se satisface en el estado inicial, pero la consecuencia está subespecificada, ya que está subespecificado el estado siguiente al estado inicial según la evolución de la acción a –$E^{\{a,0\}} = \frac{1}{2}$–.

req $\mathcal{R}_{sin}$ **is**
$(true \Rightarrow a) \Rightarrow \bigcirc (true \Rightarrow b)$
endreq

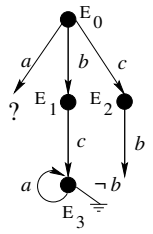


Figura 9.3: Grafo MUS $\mathcal{M}_{sin}$

La síntesis del requisito $\mathcal{R}_{sin}$ conlleva una pérdida de subespecificación, ya que hay que especificar un nuevo estado del sistema $E^{\{a,0\}}$. La primera solución, mostrada en la figura 9.4(a), parte por crear un nuevo estado donde añadir la consecuencia, perdiendo la subespecificación del estado asociado $E^{\{a,0\}}$ y la subespecificación de la acción b en el nuevo estado creado. Ésta es la solución adoptada por el algoritmo de traducción.

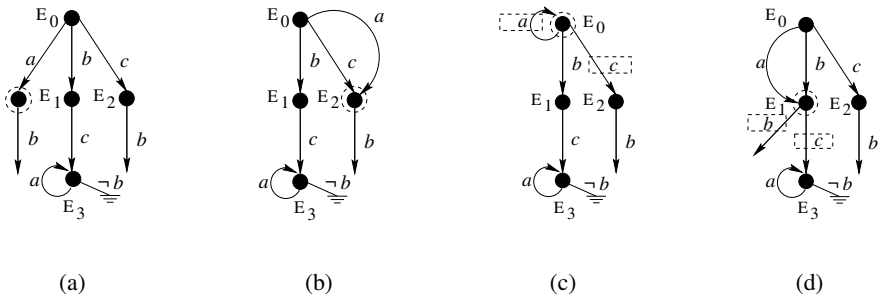


Figura 9.4: Pérdidas de subespecificación debidas a la reutilización de estados.

Sin embargo, tal y como se explicó anteriormente, es necesario reutilizar estados ya existentes en el sistema para evitar la síntesis de sistemas con infinitos estados. En la figura 9.4(b) se reutiliza el estado E_2. Esta solución conlleva una única pérdida de subespecificación, debido a que la evolución a través de las acciones a y c, a partir del estado inicial E_0, se realiza al mismo estado E_2 –$E^{\{a,0\}} = E^{\{c,0\}} = E_2$–. Esta restricción, que no ha sida especificada por el usuario, supone una pérdida de subespecificación.

En la figura 9.4(c) se reutiliza el estado E_0. Esto supone varias pérdidas de subespecificación. En primer lugar, el estado E_0 es anterior a sí mismo, y dicha evolución no ha sido especificada por el usuario. En segundo lugar, el requisito a satisfacer sólo impone que el estado siguiente a la evolución de la acción a contenga la acción b especificada como posible. Sin embargo, el estado E_0 contiene, además, especificadas como posibles las acciones a y c.

Finalmente, en la figura 9.4(d) se reutiliza el estado E_1. En este caso, la satisfacción del requisito supone una pérdida de subespecificación en el estado E_1, ya que, en dicho estado la acción

b estaba subespecificada, y la satisfacción del requisito impone que dicha acción se especifique como posible en dicho estado E_1. Además, en el estado E_1 está especificada la acción c como posible, mientras que en el requisito $\mathcal{R}_{sin}$ dicha acción está subespecificada.

A pesar de las pérdidas de subespecificación que impone la reutilización de estados, la solución pasa por evitar la creación de nuevos estados, ya que esto puede conllevar una síntesis divergente del sistema. De esta manera, cada vez que sea necesaria la creación de un nuevo estado, se explorarán los estados existentes en el sistema, eligiendo uno que sea compatible con las restricciones impuestas por el requisito SCTL a sintetizar. Dicha elección debe priorizarse según algún criterio, ya que, pueden existir varios estados compatibles con el nuevo estado a crear. En ese caso, y según lo expuesto anteriormente, se deberá elegir el estado compatible que permita perder la mínima subespecificación posible, de manera que el sistema sintetizado se ajuste al máximo a las especificaciones del usuario.

Una vez elegido dicho estado se continúa con la síntesis del sistema, si bien se deberá tener en cuenta la pérdida de subespecificación añadida en dicho estado. De esta manera, si en futuras especificaciones se llegase a una síntesis inconsistente, se volverá al paso donde se eligió el estado compatible, optando por el siguiente estado compatible según el criterio de mínima pérdida de subespecificación. Este proceso se repetirá hasta que finalmente no existan estados compatibles, en cuyo caso se creará un nuevo estado –que constituye la máxima pérdida de subespecificación–. Si no fuera posible la síntesis, se puede concluir que los requisitos especificados son incompatibles o inconsistentes.

9.3.3 Pseudocódigo del Algoritmo

El algoritmo 9.1 muestra el pseudocódigo del **algoritmo de síntesis** obtenido a partir del algoritmo de traducción SCTL-MUS 6.6. La principal diferencia entre uno y otro reside en buscar estados compatibles dentro del propio modelo de estados, en vez de crear uno nuevo cada vez que sea necesario.

A continuación, se describe el funcionamiento básico de dicho algoritmo:

1. En primer lugar, se comprueba si el requisito a sintetizar es un requisito atómico. En caso contrario, se hace una llamada recursiva al algoritmo de síntesis con el primer subrequisito obtenido por el algoritmo de partición. Dicho subrequisito se corresponderá con la premisa del requisito, o con un requisito unido por el operador lógico *And* con el segundo subrequisito. En cualquiera de los dos casos, si no es posible sintetizar dicho requisito, se devuelve ERROR.

2. Si se sintetiza con éxito el primer subrequisito, se comprueba el tipo del operador que une los dos subrequisitos que forman el requisito SCTL a sintetizar. Si dicho operador es el operador lógico *And* se realiza una nueva llamada recursiva al algoritmo de síntesis con el segundo subrequisito. Si no es posible realizar dicha síntesis, se va al paso **[8]**. En caso contrario, la síntesis se ha realizado con éxito y se devuelve OK.

Algoritmo 9.1 Algoritmo de Síntesis SCTL-MUS($\overline{\mathcal{R}}, E_h$)

[1] Si $\overline{\mathcal{R}} = \bigoplus true[\neg]a_i$ –es un requisito atómico–, ir al paso **[6]**:

[2] $\{\overline{\mathcal{R}}^1_{sub}, \overline{\mathcal{R}}^2_{sub}\}$ = **Algoritmo de Partición** $(\overline{\mathcal{R}})$;

[3] $resultado$ = **Algoritmo de Síntesis SCTL-MUS** $(\overline{\mathcal{R}}^1_{sub}, E_h)$;

[4] Si $resultado$ = ERROR, ir al paso **[8]**;

[5] Si $\overline{\mathcal{R}}[0] = \wedge$:

- **[a]** $resultado$ = **Algoritmo de Síntesis SCTL-MUS** $(\overline{\mathcal{R}}^2_{sub}, E_h)$;
- **[b]** Si $resultado$ = ERROR, ir al paso **[8]**; en caso contrario, ir al paso **[7]**;

[6] Si $\overline{\mathcal{R}}[0] \in \Theta$:

- **[a]** $\{E_{aplic}\} = \perp (\overline{\mathcal{R}}, E_h)$;
- **[b]** Si $\bigoplus = \Rightarrow \bigodot$ y $\{E_{aplic}\} = \emptyset$:
  - **[i]** $\{E_{comp}\} = \{E_j \in \mathcal{E}_{\mathcal{M}} : d[a_{sub}][j][h] \neq false\}$;
  - **[ii]** Si $\overline{\mathcal{R}}$ es atómico, $\{E_{comp}\} = \{E_j \in \{E_{comp}\} : d[a_i][j][E_{sub}] \bigcup \mathcal{S}_{\mathcal{R}}(a_i, E_j) \in \Psi\}$;
  - **[iii] Decisión**: Elegir un $E_j \in \{E_{comp}\}$ por el que no se haya decidido antes;
    - **[A]** $d[a_{sub}][j][h] = 1$; $\{E_{aplic}\} = \{E_j\}$;
  - **[iv]** En caso contrario, **última decisión**:
    - **[A]** Nuevo estado E_{n+1};
    - **[B]** $d[a_{sub}][n+1][h] = 1$; $\{E_{aplic}\} = \{E_{n+1}\}$;
  - **[v]** Si ya se había tomado la última decisión, devolver ERROR:
- **[c]** Si $\bigoplus = \Rightarrow \bigcirc, \forall a_k \in \{\Lambda_{\mathcal{R}}\}$ / $E^{\{a_k,h\}} = \frac{1}{2}$:
  - **[i]** $\{E_{comp}\} = \{E_j \in \mathcal{E}_{\mathcal{M}} : d[a_k][h][j] \neq false\}$;
  - **[ii]** Si $\overline{\mathcal{R}}$ es atómico, $\{E_{comp}\} = \{E_j \in \{E_{comp}\} : d[a_i][j][E_{sub}] \bigcup \mathcal{S}_{\mathcal{R}}(a_i, E_j) \in \Psi\}$;
  - **[iii] Decisión**: Elegir un $E_j \in \{E_{comp}\}$ por el que no se haya decidido antes;
    - **[A]** $d[a_k][h][j] = 1$; $E^{\{a_k,h\}} = E_j$;
  - **[iv]** En caso contrario, **última decisión**:
    - **[A]** Nuevo estado E_{n+1};
    - **[B]** $d[a_k][h][n+1] = 1$; $E^{\{a_k,h\}} = E_{n+1}$; $\{E_{aplic}\} = \{E_{aplic}\} \bigcup E_{n+1}$;
  - **[v]** Si ya se había tomado la última decisión, devolver ERROR:
- **[d]** $\forall E_j \in \{E_{aplic}\}$:
  - **[i]** Si $\overline{\mathcal{R}}$ es atómico y $d[a_i][j][E_{sub}] = d[a_i][j][E_{sub}] \bigcup \mathcal{S}_{\mathcal{R}}(a_i, E_j) \notin \Psi$:
    deshacer los cambios y devolver ERROR;
  - **[ii]** Si $\overline{\mathcal{R}}$ no es atómico:
    - **[A]** $resultado$ = **Algoritmo de síntesis SCTL-MUS** $(\overline{\mathcal{R}}^2_{sub}, E_j)$;
    - **[B]** Si $resultado$ = ERROR, ir al paso **[8]**:

[7] Devolver OK;

[8] Si se tomó alguna decisión antes de la llamada que devolvió ERROR:

- **[a]** Deshacer los cambios realizados en dicha decisión;
- **[b]** Tomar una nueva decisión, paso **[6][b][iii]** o **[6][c][iii]**;

[9] En caso contrario, devolver ERROR;

3. Si el operador es temporal, o si el requisito era atómico, se calculan los estados de aplicabilidad de dicho requisito, con el fin de sintetizar la consecuencia (el segundo subrequisito) en dichos estados. En el caso del operador temporal *A la vez* el único estado de aplicabilidad es el mismo estado E_h, por lo que se pasa directamente al paso **[6][d]**.

4. Si el operador temporal es *Antes* y existen estados de aplicabilidad, se pasa también al estado **[6][d]**. En caso contrario, es necesario buscar un estado de aplicabilidad para satisfacer el requisito. Para ello, tal y como se expuso anteriormente, se buscan estados compatibles dentro del propio sistema, siendo la creación de un nuevo estado la última opción a elegir. La condición de compatibilidad consiste en que el estado elegido pueda ser anterior al estado E_h donde se está sintetizando el requisito. Si además el requisito es atómico, dicho estado debe tener una especificación compatible de la acción expresada en dicho requisito.

5. El funcionamiento del algoritmo es similar para el operador temporal *Después*, salvo que es necesario buscar un estado –donde aplicar la consecuencia– para todos los posibles estados de aplicabilidad, independientemente de que estos estuvieran subespecificados. Esto permite mantener en el sistema la satisfacción de dichos requisitos ante futuras especificaciones.

6. Finalmente, en el paso **[6][d]** se sintetiza la consecuencia del requisito en todos los estados de aplicabilidad. Algunos de dichos estados pueden haberse decidido en los pasos anteriores. Ver los pasos **[6][b][iii]** y **[6][c][iii]**.

7. Si el requisito es atómico, basta con añadir en dichos estados la especificación de la acción expresada en el requisito. En caso de no poder realizar dicha operación, se devuelve ERROR. Hay que tener en cuenta que los estados de aplicabilidad "decididos" nunca pueden causar este error, ya que, dicha condición se verifica antes de su elección en los pasos **[6][b][ii]** y **[6][c][ii]**.

8. Si el requisito no es atómico, la síntesis se obtiene mediante llamadas recursivas al algoritmo de síntesis con el segundo subrequisito (la consecuencia) y cada uno de los estados de aplicabilidad.

9. Si alguna llamada recursiva al algoritmo de síntesis devuelve ERROR, hay que comprobar si dicho error puede haber sido producido por alguna decisión tomada en el presente algoritmo, en cuyo caso, se opta por una nueva decisión. Si no se ha tomado ninguna decisión anterior a la llamada que devolvió el error, o si ya se han agotado todas las decisiones posibles, se devuelve ERROR. De esta manera, el error se propaga hacia decisiones anteriores, hasta que o bien se resuelve o se agotan dichas decisiones, en cuyo caso, el requisito especificado sería incompatible con el sistema actual.

10. Aunque en el algoritmo 9.1 no se especifica, en cada estado se mantiene información relativa a los requisitos sintetizados en dicho estado. De esta manera, se evita sintetizar en un estado varias veces un mismo requisito.

Cabe destacar, que el algoritmo es independiente del criterio adoptado para la elección de uno u otro estado compatible. De hecho, podría añadirse la creación de un nuevo estado al conjunto $\{E_{comp}\}$ e ir eligiendo uno u otro hasta agotar todas las decisiones. Esto permite una flexibilidad en el algoritmo de síntesis, que consiste en determinar cuál es el criterio de decisión adoptado por el algoritmo en cada momento. Dicho criterio podría ser tomado por el usuario en tiempo de diseño o automatizarse según una relación de orden cualquiera.

9.3.4 Ejemplo de Aplicación

En la figura 9.5 se muestra el grafo MUS $\mathcal{M}_{int}$ correspondiente a un sistema en una etapa intermedia del diseño. En dicha etapa se produce la especificación de dos nuevos requisitos, formulados ambos como invarianzas:

req $\mathcal{R}_1$ is
$(true \Rightarrow a) \wedge (true \Rightarrow \odot b) \Rightarrow \bigcirc\, (true \Rightarrow b) \Rightarrow (true \Rightarrow \neg c)$
endreq

req $\mathcal{R}_2$ is
$(true \Rightarrow \neg c) \Rightarrow \bigcirc\, (true \Rightarrow \neg a)$
endreq

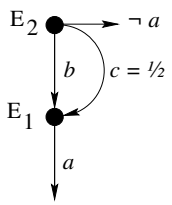


Figura 9.5: Grafo MUS $\mathcal{M}_{int}$ de un sistema en una fase intermedia del diseño.

El primer paso para la inclusión de estos nuevos requisitos corresponde al algoritmo de verificación, el cual comprueba que la premisa de $\mathcal{R}_1$ –$(true \Rightarrow a) \wedge (true \Rightarrow \odot b)$– se satisface en el estado E_1; y su consecuencia está subespecificada –$E^{\{a,1\}} = \frac{1}{2}$–, cumpliéndose así una de las condiciones de aplicabilidad del algoritmo de síntesis (ver la sección 9.2). Sin embargo, dicha premisa no se satisface en el estado E_2 –$E_2[a] = 0$–, por lo que no se aplica el algoritmo de síntesis en dicho estado.

El siguiente paso corresponde, por tanto, al algoritmo de síntesis, el cual se invoca con el requisito $\mathcal{R}_1$ y el estado E_1. La aplicación de dicho algoritmo supone una toma de decisión para asignar un estado siguiente al estado E_1 según la evolución de la acción a. En la figura 9.6 se muestra el resultado del algoritmo tomando como decisión el estado E_2. Esta decisión supone una pérdida de subespecificación para $\mathcal{R}_1$, debido a que en el estado E_2 la acción a está especificada

como negada, mientras que en la consecuencia del requisito $\mathcal{R}_1$ –$(true \Rightarrow b) \Rightarrow (true \Rightarrow \neg c)$–, dicha acción está subespecificada. De la misma manera, supone una pérdida de subespecificación para el estado E_2, ya que, en él la acción c estaba subespecificada, y la síntesis del requisito $\mathcal{R}_1$ hace que se especifique como no posible.

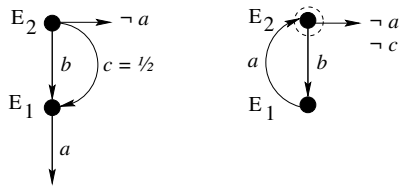


Figura 9.6: Síntesis de $\mathcal{R}_1$. Decisión I.

La síntesis de $\mathcal{R}_1$ se ha completado, ya que, no existe ningún otro estado donde se satisfaga su premisa. Se pasa, por tanto, a la síntesis del requisito $\mathcal{R}_2$. Su premisa se satisface en el estado E_2, pero no se satisface su consecuencia, ya que, en su estado siguiente –E_1–, la acción a está especificada como posible (figura 9.7), mientras que en la consecuencia de $\mathcal{R}_2$ se especifica como no posible.

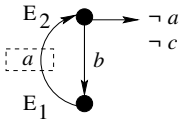


Figura 9.7: $\mathcal{R}_2$ no se satisface en E_2.

Sin embargo, es posible sintetizar un sistema que satisfaga ambos requisitos. El error parte por la decisión tomada en la síntesis de $\mathcal{R}_1$. Es necesario, por tanto, volver a sintetizar $\mathcal{R}_1$ tomando una nueva decisión. En la figura 9.8 se muestra el resultado del algoritmo de síntesis correspondiente al requisito $\mathcal{R}_1$ y al estado E_1, tomando una nueva decisión: $E^{\{a,1\}} = E_1$.

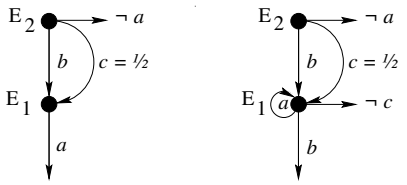


Figura 9.8: Síntesis de $\mathcal{R}_1$. Decisión II.

Una vez sintetizado $\mathcal{R}_1$, se vuelve a verificar $\mathcal{R}_2$. En E_2 no se satisface su premisa, por lo que no es aplicable el algoritmo de síntesis. En E_1 sí se satisface su premisa, pero de nuevo no se satisface su consecuencia, ya que, la acción a se especifica como posible en su estado asociado, que en este caso es también el estado E_1 (figura 9.9).

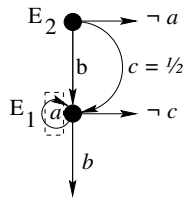


Figura 9.9: $\mathcal{R}_2$ no se satisface en E_1.

Aún queda una última decisión en la síntesis de $\mathcal{R}_1$, que se corresponde con la creación de un nuevo estado. En la figura 9.10 se muestra esta opción.

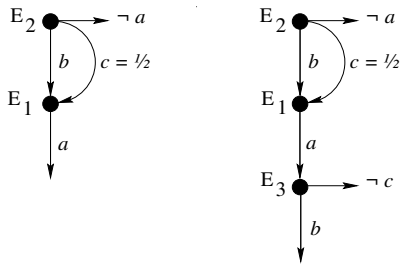


Figura 9.10: Síntesis de $\mathcal{R}_1$. Decisión III.

Finalmente, se vuelve a verificar $\mathcal{R}_2$ en cada estado del sistema. En E_2 no se satisface la premisa, por lo que no es aplicable el algoritmo de síntesis. Lo mismo ocurre con E_1. Por el contrario, en E_3 se satisface la premisa. Si la consecuencia no se satisface, o si no fuera posible sintetizarla perdiendo subespecificación, los requisitos especificados serían inconsistentes, ya que, se han agotado las decisiones para la síntesis de $\mathcal{R}_1$. Sin embargo, la consecuencia de $\mathcal{R}_2$ está subespecificada, debido a que están subespecificados los estados siguientes a E_3, y es posible perder dicha subespecificación de manera que se satisfaga la consecuencia de $\mathcal{R}_2$.

Esto conlleva una nueva decisión respecto a estos estados, eligiendo cualquier estado compatible con la especificación de la consecuencia de $\mathcal{R}_2$ –$\neg a$–. Las posibles decisiones son:

1. Reutilizar el estado E_2, tal y como se muestra en la figura 9.11(a).

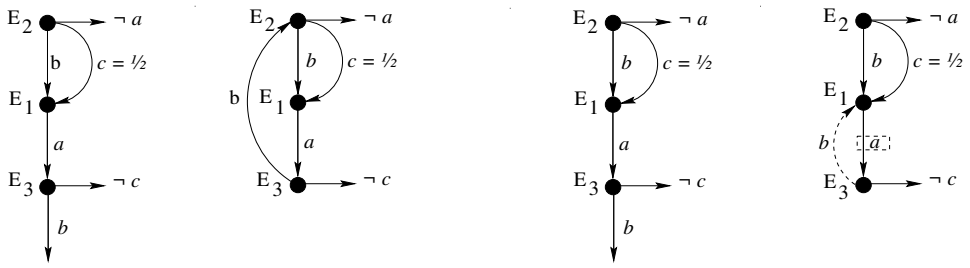
2. Reutilizar el estado E_1. Esto no es posible, ya que, en dicho estado la acción a está especificada como posible. Figura 9.11(b).

3. Reutilizar el estado E_3. Lo que conlleva una pérdida de subespecificación de la acción a en dicho estado. Figura 9.12(a).

4. La creación de un nuevo estado E_4, tal y como se muestra en la figura 9.12(b).

En las figuras 9.11 y 9.12 se muestra la decisión para el estado siguiente al E_3 respecto a la evolución mediante la acción b. De manera similar, se adoptaría la decisión para el estado

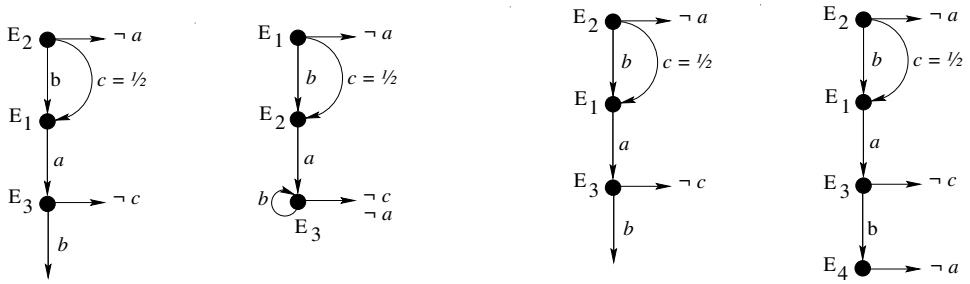
siguiente según la acción a, que está subespecificada en dicho estado, obteniendo 9 sistemas diferentes producto de combinar cada una de las dos decisiones, ya que, cada una de ellas ofrece tres posibilidades diferentes.



(a) Decisión I

(b) Decisión no válida

Figura 9.11: Síntesis de $\mathcal{R}_2$ I.



(a) Decisión II

(b) Decisión III

Figura 9.12: Síntesis de $\mathcal{R}_2$ II.

9.4 Conclusiones

9.4.1 Pérdida de Subespecificación

A continuación, se evalúan cada una de las posibles pérdidas de subespecificación causadas por la reutilización de estados expuesta en la sección 9.3.2.

- En primer lugar, la reutilización de un estado implica la creación de un nuevo enlace entre dos estados, por lo que uno de ellos tendrá que satisfacer las restricciones de estado anterior del primero, y el otro las restricciones de estado siguiente. Estas restricciones no las ha especificado el usuario, ya que, el enlace entre los dos estados (un bucle) se ha creado artificialmente.

- El estado compatible puede contener subespecificadas algunas de las acciones especificadas

por el usuario en el nuevo estado, lo que conlleva una pérdida de subespecificación en las acciones del estado compatible elegido.

- El estado compatible puede contener especificadas más acciones de las que el usuario ha especificado para el nuevo estado, lo que conlleva una pérdida de subespecificación en el requisito especificado por el usuario.

Una vez enumeradas las posibles pérdidas de subespecificación, se muestran los criterios adoptados para la elección de un estado compatible con la mínima pérdida de subespecificación:

1. Mínima pérdida de subespecificación en las acciones del estado compatible. Es decir, se evita añadir, en lo posible, nuevas especificaciones de acciones en los estados ya especificados en el sistema.

2. Mínima pérdida de subespecificación en el requisito especificado. Según este criterio, se elegirá el estado compatible que añada el menor número de restricciones al requisito especificado por el usuario.

En este sentido, crear un nuevo estado será la última opción elegida, ya que, impone la pérdida de subespecificación en dicho estado de todas las acciones especificadas en el requisito a sintetizar. En el ejemplo de la figura 9.4 las decisiones se tomarían en el siguiente orden: (b), (c), (d) y (a).

9.4.2 Valoración del Algoritmo Obtenido

El algoritmo de síntesis 9.1 expuesto permite sintetizar un modelo de estados subespecificados que satisface un conjunto de requisitos SCTL. Dicha síntesis tiene una relación directa con la técnica de especificación utilizada, ya que, necesita la especificación del requisito en SCTL para poder ir sintetizando el modelo de estados correspondiente, es decir, tiene una **dependencia de SCTL**. Dicha dependencia está causada por la manera en la que se ha obtenido el algoritmo de síntesis, ya que, éste se basa en el algoritmo de traducción SCTL-MUS, que naturalmente depende de SCTL.

La síntesis de varios sistemas que contengan subrequisitos comunes es totalmente independiente. Es decir, en cada una de ellas se aplica el algoritmo de síntesis particularizando sobre el sistema el requisito a satisfacer, sin obtener ningún grado de **reutilización** por el hecho de haber sintetizado dicho requisito en otros sistemas.

Mediante el algoritmo de síntesis 9.1 se modifica el grafo que representa el comportamiento del sistema especificado hasta que satisface un nuevo requisito SCTL. Sin embargo, dicho sistema puede verse de nuevo modificado por la especificación de un nuevo requisito, Naturalmente, el primer requisito debe seguir satisfaciéndose, si bien esto puede no ser cierto debido a que el último requisito puede haber originado la aparición de nuevos estados de aplicabilidad para el primer requisito. Dichos estados no satisfacen el primer requisito, debido a que cuando se le aplicó el algoritmo de síntesis, dichos estados no existían.

El algoritmo de síntesis 9.1 se obtiene a partir del algoritmo de traducción SCTL-MUS, en el cual, no existe la posibilidad de que aparezcan nuevos estados de aplicabilidad, tal y como se explica a continuación:

1. En el caso del operador *Después*, el requisito se aplica tanto en los estados siguientes existentes como en los estados siguientes subespecificados, creando dichos estados si es necesario, con el fin de evitar tener que evaluar de nuevo dicho requisito.

2. En el caso del operador *Antes*, sólo es posible obtener un único estado anterior a cada uno de los estados del requisito. Esto es debido a que el modelo de estados correspondiente a un requisito SCTL muestra el comportamiento de éste sin introducir ninguna restricción. Es decir, nunca realiza bucles, y por tanto un estado sólo puede tener un único estado anterior. De esta manera, cuando se traduce un requisito que hace referencia a un estado anterior, se aplica a dicho estado, creándolo si éste no existiese. La traducción de requisitos recursivos se soluciona gracias a que dichos requisitos expresan su recursividad en SCTL, por lo que es detectada por el algoritmo de traducción, tal y como se pone de manifiesto en el capítulo 6.

En el algoritmo de síntesis 9.1 se sigue realizando la síntesis para los estados siguientes subespecificados. Sin embargo, se realizan bucles, producto de la reutilización de estados, lo que conlleva la aparición de nuevos estados anteriores al estado en el que se ha realizado el bucle. Dichos estados pueden ser estados de aplicabilidad de requisitos previamente sintetizados, sobre los que no se ha sintetizado dicho requisito, lo que obliga a repetir la verificación y síntesis de todos los requisitos cada vez que se aplica el algoritmo de síntesis a un nuevo requisito, con los consiguientes costes y pérdida de **eficiencia**.

Resumiendo, el algoritmo de síntesis 9.1 obtenido muestra tres deficiencias principales:

1. Dependencia de SCTL.

2. No ofrece ninguna posibilidad de reutilización.

3. Conlleva la evaluación continua de los requisitos anteriormente sintetizados.

En el siguiente capítulo se abordan cada uno de estos problemas, obteniendo distintos algoritmos de síntesis, evaluando las ventajas e inconvenientes de cada uno de ellos.

Capítulo 10

Reutilización en el Proceso de Síntesis Incremental

10.1 Algoritmo de Síntesis: En Busca de Reutilización

Para poder reutilizar la síntesis de un mismo requisito en varios sistemas, es necesario aislar el proceso de traducción del de síntesis. La solución que se plantea es la siguiente:

- En primer lugar, obtener un grafo MUS a partir del requisito SCTL especificado, según el algoritmo de traducción SCTL-MUS.
- Para añadir dicho requisito a un sistema modelado mediante un grafo MUS, es necesario que parte de los dos grafos coincidan. Esto es debido a la semántica de las proposiciones de los requisitos SCTL: *"si se satisface la premisa, se debe satisfacer la consecuencia en los estados de aplicabilidad"*. Por tanto, el grafo MUS correspondiente al requisito y el correspondiente al sistema deben coincidir, al menos, en la parte que representa el comportamiento de la premisa.
- Para obtener un sistema que satisfaga el nuevo requisito, basta con solapar el resto del grafo del requisito en el grafo del sistema. Dicho solapamiento no tiene por qué ser único, obteniéndose familias de sistemas que satisfacen el mismo conjunto de requisitos. Dichos sistemas se corresponderían con distintas decisiones tomadas durante la síntesis, al igual que las decisiones sobre estados compatibles tomadas en el algoritmo 9.1.

10.1.1 Solapamiento de Estados

El nuevo algoritmo de síntesis propuesto debe, por tanto, solapar cada uno de los estados del requisito en algún estado del árbol del sistema. Las especificaciones correspondientes a la premisa deben coincidir en el grafo del sistema, mientras que las correspondientes a la consecuencia deben añadirse –si fuera necesario– a dicho grafo, satisfaciendo así el nuevo requisito especificado.

Sin embargo, la correspondencia entre los estados a solapar no tiene por qué ser de uno a uno. Es decir, los estados de aplicabilidad del requisito no tienen por qué coincidir en el requisito y en el grafo del sistema. Por ejemplo, en el grafo MUS de un requisito atómico con el operador $Antes$, sólo existe un estado anterior; mientras que en un sistema particular pueden existir varios estados anteriores donde aplicar la consecuencia de dicho requisito [1]. Por ello, es necesario solapar cada estado de aplicabilidad del requisito en todos los estados de aplicabilidad del grafo del sistema.

Dado que en el requisito pueden existir varios estados de aplicabilidad, bastaría con elegir cualquiera de ellos para solaparlo en todos los estados de aplicabilidad del grafo del sistema, ya que, la consecuencia se aplica por igual a cada uno de los estados de aplicabilidad. Sin embargo, esto no es posible debido a que un estado puede contener información relativa a varios de los subrequisitos que forman el requisito original. Es decir, puede ser un estado de aplicabilidad de varios subrequisitos que forman el requisito a sintetizar. Para obtener únicamente las restricciones impuestas por un subrequisito en particular, basta con calcular la intersección de todos sus estados de aplicabilidad.

Definición 10.1. Se define la operación **intersección** entre dos estados E_j, E_k de un grafo MUS $\mathcal{M}$ como un vector de especificaciones de acciones $\{\mathcal{S}_a\} = \{\mathcal{S}(a_1), ..., \mathcal{S}(a_m) : a_i \in \Lambda, \mathcal{S}(a_i) \in \Psi$, tal que $\mathcal{S}(a_i) = E_j[a_i]$, si $E_j(a_i) = E_k[a_i]$, y $\mathcal{S}(a_i) = \frac{1}{2}$ en caso contrario$\}$.

La figura 10.1 muestra el grafo MUS correspondiente al requisito $\mathcal{R}_{int} = \mathcal{R}_{int_1} \wedge \mathcal{R}_{int_2}$:

req $\mathcal{R}_{int_1}$ **is**
$(true \Rightarrow \odot\, b) \Rightarrow \bigcirc\, (true \Rightarrow b)$
endreq

req $\mathcal{R}_{int_2}$ **is**
$(true \Rightarrow a) \Rightarrow \bigcirc(true \Rightarrow \neg c)$
endreq

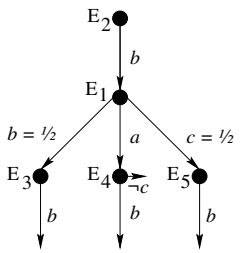


Figura 10.1: Grafo MUS del requisito R_{int}.

Para obtener las restricciones impuestas por el requisito $\mathcal{R}_{int_1}$, es necesario realizar la intersección entre sus estados de aplicabilidad –E_3, E_4 y E_5–, ya que, el estado E_4 contiene, además, la restricción sobre la acción c impuesta por $\mathcal{R}_{int_2}$.

$$E_3 \cap E_4 \cap E_5 \ = \{\mathcal{S}_a(a) = \tfrac{1}{2}, \mathcal{S}_a(b) = 1, \mathcal{S}_a(c) = \tfrac{1}{2}\}$$

[1] Ver figura 9.1

Para realizar el solapamiento de dos estados, basta con aplicar la operación *unión* (ver definición 7.11), a las filas y columnas del grafo MUS correspondientes a dichos estados. En la figura 10.2 se muestra el grafo MUS resultante después de unir el estado E_2 al estado E_1 del grafo MUS de la figura 10.1. Dicha operación impone la unión de los estados E_1 y E_3, ya que, tanto en E_2 como en E_1 está especificado el estado siguiente según la acción b. En el primer caso, el estado siguiente es E_1, mientras que en el segundo, el estado siguiente es E_3 –$E^{\{b,2\}} = E_1$ y $E^{\{b,1\}} = E_3$–. Es decir, la unión de los estados, implica, además, la unión de sus estados siguientes.

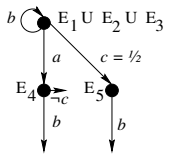


Figura 10.2: Unión de dos estados.

10.1.2 Pseudocódigo del Algoritmo

El algoritmo 10.1 muestra el pseudocódigo del nuevo algoritmo de síntesis. Dicho algoritmo recibe como parámetros el requisito SCTL $\mathcal{R}$ a sintetizar y el estado E_h del grafo del sistema sobre el que se solapa la intersección de un conjunto de estados de aplicabilidad del requisito $\{E_h^{\mathcal{R}}\}$.

El pseudocódigo del algoritmo de síntesis 10.1 es similar al algoritmo 9.1. La diferencia básica entre uno y otro reside en la forma que ambos tratan a los requisitos SCTL a sintetizar. El algoritmo 9.1 recorre los requisitos extrayendo la información que éstos especifican. Por el contrario, algoritmo 10.1 utiliza la especificación SCTL únicamente para obtener los estados de aplicabilidad correspondientes al grafo MUS del sistema donde solapar cada uno de los estados del grafo MUS del requisito SCTL –obtenido previamente mediante el algoritmo de traducción SCTL-MUS–.

Por tanto, mientras que en el algoritmo 9.1 se unen las especificaciones de las acciones expresadas en SCTL (paso **[6][d][i]**), en el algoritmo 10.1 se unen estados del grafo MUS del requisito a sintetizar, y del grafo MUS del sistema actual (paso **[1]**). El resto del algoritmo 10.1 es idéntico al algoritmo 9.1. Se toman decisiones y se almacena en cada estado la información relativa a los estados con los que se ha unido (solapado) para evitar repetir dichos solapamientos.

10.1.3 Ejemplo de Aplicación

A continuación, se aplica el algoritmo 10.1 a la síntesis de los requisitos $\mathcal{R}_1$ y $\mathcal{R}_2$ realizada en la sección 9.3.4. En la figura 10.3 se muestran los grafos MUS correspondientes al sistema en la fase actual del diseño y a los requisitos $\mathcal{R}_1$ y $\mathcal{R}_2$ respectivamente. Los dos últimos han sido obtenidos mediante el algoritmo de traducción SCTL-MUS 6.6.

Algoritmo 10.1 Algoritmo de Síntesis SCTL-MUS II ($\overline{\mathcal{R}}, E_h, \{E_h^{\mathcal{R}}\} = \{E_{h_1}^{\mathcal{R}}, ..., E_{h_l}^{\mathcal{R}}\}$)

[1] Si $E_h = E_h \bigcup \{\bigcap_{i=1}^{i=l} E_{h_i}^{\mathcal{R}}\} \notin \Psi^m$, deshacer los cambios y devolver ERROR;

[2] Si $\overline{\mathcal{R}} = \bigoplus true[\neg]a_i$ –es un requisito atómico–, ir al paso **[7]**:

[3] $\{\overline{\mathcal{R}}_{sub}^1, \overline{\mathcal{R}}_{sub}^2\}$ = **Algoritmo de Partición** $(\overline{\mathcal{R}})$;

[4] $resultado$ = **Algoritmo de Síntesis SCTL-MUS II** $(\overline{\mathcal{R}}_{sub}^1, E_h, \{E_h^{\mathcal{R}}\})$;

[5] Si $resultado$ = ERROR, ir al paso **[8]**;

[6] Si $\overline{\mathcal{R}}[0] = \wedge$:

[a] $resultado$ = **Algoritmo de Síntesis SCTL-MUS II** $(\overline{\mathcal{R}}_{sub}^2, E_h, \{E_h^{\mathcal{R}}\})$;

[b] Si $resultado$ = ERROR, ir al paso **[8]**; en caso contrario, ir al paso **[7]**;

[7] Si $\overline{\mathcal{R}}[0] \in \Theta$:

[a] $\{E_{aplic}\} = \perp (\overline{\mathcal{R}}, E_h)$, $\{E_{aplic}^R\} = \sum_{i=i}^{i=l} \perp (\overline{\mathcal{R}}, E_{h_i}^R)$;

[b] Si $\bigoplus = \Rightarrow \bigodot$ y $\{E_{aplic}\} = \emptyset$:

[i] $\{E_{comp}\} = \{E_j \in \mathcal{E}_{\mathcal{M}} : d[a_{sub}][j][h] \neq false\}$;

[ii] Si $\overline{\mathcal{R}}$ es atómico, $\{E_{comp}\} = \{E_j \in \{E_{comp}\} : d[a_i][j][E_{sub}] \bigcup \mathcal{S}_{\mathcal{R}}(a_i, E_j) \in \Psi\}$;

[iii] Decisión: Elegir un $E_j \in \{E_{comp}\}$ por el que no se haya decidido antes;

[A] $d[a_{sub}][j][h] = 1$; $\{E_{aplic}\} = \{E_j\}$;

[iv] En caso contrario, **última decisión**:

[A] Nuevo estado E_{n+1};

[B] $d[a_{sub}][n+1][h] = 1$; $\{E_{aplic}\} = \{E_{n+1}\}$;

[v] Si ya se había tomado la última decisión, devolver ERROR:

[c] Si $\bigoplus = \Rightarrow \bigcirc$, $\forall a_k \in \{\Lambda_{\mathcal{R}}\}$ / $E^{\{a_k,h\}} = \frac{1}{2}$:

[i] $\{E_{comp}\} = \{E_j \in \mathcal{E}_{\mathcal{M}} : d[a_k][h][j] \neq false\}$;

[ii] Si $\overline{\mathcal{R}}$ es atómico, $\{E_{comp}\} = \{E_j \in \{E_{comp}\} : d[a_i][j][E_{sub}] \bigcup \mathcal{S}_{\mathcal{R}}(a_i, E_j) \in \Psi\}$;

[iii] Decisión: Elegir un $E_j \in \{E_{comp}\}$ por el que no se haya decidido antes;

[A] $d[a_k][h][j] = 1$; $E^{\{a_k,h\}} = E_j$; $\{E_{aplic}\} = \{E_{aplic}\} \bigcup E_{n+1}$;

[iv] En caso contrario, **última decisión**:

[A] Nuevo estado E_{n+1};

[B] $d[a_k][h][n+1] = 1$; $E^{\{a_k,h\}} = E_{n+1}$;

[v] Si ya se había tomado la última decisión, devolver ERROR:

[d] $\forall E_j \in \{E_{aplic}\}$:

[i] Si $\overline{\mathcal{R}}$ es atómico y $E_h = E_h \bigcup \{\bigcap_{i=1}^{i=l} E_{h_i}^{\mathcal{R}}\} \notin \Psi^m$:

deshacer los cambios y devolver ERROR;

[ii] Si $\overline{\mathcal{R}}$ no es atómico:

[A] $resultado$ = **Algoritmo de síntesis SCTL-MUS II** $(\overline{\mathcal{R}}_{sub}^2, E_j, \{E_{aplic}^{\mathcal{R}}\})$;

[B] Si $resultado$ = ERROR, ir al paso **[8]**:

[8] Devolver OK;

[9] Si se tomó alguna decisión antes de la llamada que devolvió ERROR:

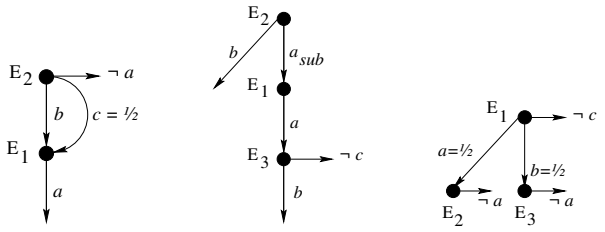
[a] Deshacer los cambios realizados en dicha decisión;

[b] Tomar una nueva decisión, paso **[7][b][iii]** o **[7][c][iii]**;

[10] En caso contrario, devolver ERROR;

req $\mathcal{R}_1$ is
$(true \Rightarrow a) \wedge (true \Rightarrow \odot b) \Rightarrow \bigcirc (true \Rightarrow b) \Rightarrow (true \Rightarrow \neg c)$
endreq

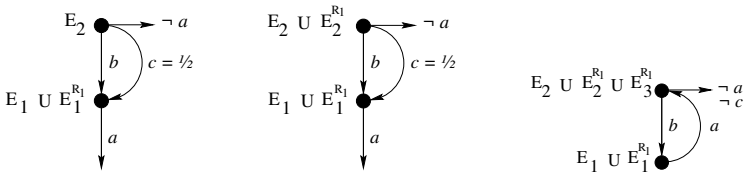
req $\mathcal{R}_2$ is
$(true \Rightarrow \neg c) \Rightarrow \bigcirc (true \Rightarrow \neg a)$
endreq



(a) $\mathcal{M}$. (b) $\mathcal{M}_{\mathcal{R}_1}$. (c) $\mathcal{M}_{\mathcal{R}_2}$.

Figura 10.3: Grafos MUS del sistema y de los requisitos $\mathcal{R}_1$ y $\mathcal{R}_2$.

En la figura 10.4 se muestra, paso por paso, la síntesis del requisito $\mathcal{R}_1$, utilizando el algoritmo 10.1 y tomando la primera decisión $-E^{\{a,1\}} = E_2-$. El resultado final (ver figura 10.4(c)) coincide con el obtenido utilizando el algoritmo 9.1 de la figura 9.6.



(a) Paso I (b) Paso II (c) Paso III

Figura 10.4: Síntesis de $\mathcal{R}_1$ mediante el Algoritmo 10.1. Decisión I.

Al igual que en la sección 9.3.4 la síntesis del requisito $\mathcal{R}_2$ es incompatible con el grafo MUS de la figura 10.4(c). En este caso, el algoritmo 10.1 detecta dicha inconsistencia debido a que no es posible realizar la unión de los estados E_1 y $E_2^{\mathcal{R}_2}$.

En la figura 10.5 se muestra el modelo de estados obtenido cuando se elige la segunda decisión $-E^{\{a,1\}} = E_1-$ en la síntesis del requisito $\mathcal{R}_1$. De nuevo, dicho modelo de estados coincide con el obtenido por el algoritmo 9.1 de la figura 9.8. En esta ocasión el algoritmo de síntesis 10.1 detecta la inconsistencia de dicho modelo de estados con el requisito $\mathcal{R}_2$ debido a que no es posible realizar la unión de los estados E_1 y $E_2^{\mathcal{R}_2}$.

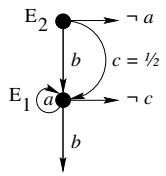


Figura 10.5: Síntesis de $\mathcal{R}_1$. Decisión II.

Finalmente, en la figura 10.6 se muestra el grafo MUS obtenido tras sintetizar $\mathcal{R}_1$ tomando la tercera decisión $-E^{\{a,1\}} = E_3-$. El modelo de estados obtenido coincide con el de la figura 9.10, que se obtuvo mediante el algoritmo 9.1. A partir de dicho modelo de estados es posible sintetizar $\mathcal{R}_2$, siendo necesaria una nueva decisión que ya se explicó en la sección 9.3.4, y que coincide en su totalidad al aplicar el nuevo algoritmo de síntesis.

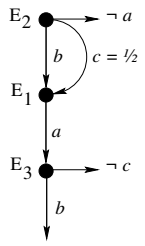


Figura 10.6: Síntesis de $\mathcal{R}_1$. Decisión III.

10.2 Algoritmo de Síntesis: Aumentando la Reutilización

En el algoritmo 10.1 se consigue cierto grado de reutilización, ya que, parte de la síntesis está implícita en el modelo de estados que representa el requisito SCTL, que se calcula una única vez mediante el algoritmo de traducción; reutilizándola, por tanto, cada vez que el requisito correspondiente se especifica en un sistema en concreto. Sin embargo, es posible aumentar considerablemente dicho grado de reutilización.

En el algoritmo 10.1, al igual que en el algoritmo 9.1, se buscan estados ya existentes en el sistema para poder realizar bucles, sintetizando así sistemas con el menor número de estados posible. Estos bucles podrían haberse realizado previamente en el grafo MUS del requisito, siempre que la elección del estado del sistema donde se va a realizar el bucle, coincida con un estado donde se sintetizó algún estado del requisito. En ese caso, dicho bucle podría haberse realizado una única vez en el grafo del requisito, evitando dicha búsqueda en todos los sistemas donde se realice la síntesis del mismo.

Sea $\mathcal{M}_R$ el grafo MUS correspondiente al requisito SCTL $\mathcal{R}$ obtenido mediante el algoritmo de traducción SCTL-MUS. A continuación, se detalla la estrategia a seguir para obtener un mayor grado de reutilización en la síntesis del requisito $\mathcal{R}$, según la idea expuesta:

- Obtener un conjunto de grafos MUS $\{\mathcal{M}^{\mathcal{R}}\} = \{\mathcal{M}_1^{\mathcal{R}}, ..., \mathcal{M}_r^{R}\}$, correspondientes a representaciones de dicho requisito donde, en vez de crear nuevos estados, se han reutilizado estados existentes similarmente a como se hace en los algoritmos de síntesis 9.1 y 10.1. Dichos grafos MUS se ordenan según el número de estados que contienen. $\mathcal{M}_1^{\mathcal{R}}$ contiene el menor numero de estados, y $\mathcal{M}_r^{\mathcal{R}}$ el máximo, por lo que se corresponderá con $\mathcal{M}_R$, el modelo de estados obtenido por el algoritmo de traducción SCTL-MUS 6.6. Es necesario, por tanto, el desarrollo de un nuevo algoritmo, denominado **algoritmo de reducción de estados**, que obtenga el conjunto de grafos MUS $\{\mathcal{M}^{\mathcal{R}}\}$. Dicho algoritmo se describe en la sección 10.2.1.

- El algoritmo de síntesis debe seguir aplicando el mismo criterio de búsqueda de estados compatibles, ya que, en los grafos del requisito sólo se recogen los bucles a estados del requisito. Sin embargo, dicha búsqueda se debe reducir a los estados del sistema donde no ha sido solapado ningún estado correspondiente al requisito, ya que, éstos están implícitos en alguno de los modelos de estados obtenidos por el algoritmo de reducción.

- Para sintetizar un requisito SCTL $\mathcal{R}$, se sintetizará primero el modelo de estados $\mathcal{M}_1^{\mathcal{R}}$, aplicándose el mismo algoritmo de síntesis 10.1. Si no fuera posible la síntesis, se intentaría con el siguiente modelo de estados, y así sucesivamente hasta llegar al modelo de estados $\mathcal{M}^{R}$.

- En el proceso de síntesis, hay que almacenar –además de las decisiones de estado tomadas en el algoritmo 10.1– el grafo MUS sintetizado, obteniendo así nuevas decisiones de diseño.

De esta manera, se consigue el máximo grado de reutilización, ya que, se almacena en el requisito toda la información común a su síntesis. Dicha información consta del propio modelo de estados y de todas las posibles decisiones comunes a todos los sistemas donde sintetizar dicho requisito.

10.2.1 Algoritmo de Reducción de Estados

El algoritmo de reducción de estados obtiene, a partir de un grafo MUS $\mathcal{M}_{\mathcal{R}}$ un conjunto de grafos MUS que se calculan mediante la unión de los estados del grafo inicial. Existirán tantos grafos como posibilidades diferentes en la unión de estados, si bien, no todas serán posibles, ya que pueden existir estados incompatibles que no pueden ser unidos. Por tanto, el algoritmo de reducción de estados consiste en aplicar la operación unión entre estados, por lo que, para su explicación se va a utilizar un ejemplo ilustrativo de dicho algoritmo.

La figura 10.7 muestra el grafo MUS correspondiente al requisito SCTL $\mathcal{R}_{red}$, que se muestra a continuación:

req $\mathcal{R}_{red}$ **is**

$$(true \Rightarrow \bigodot b) \Rightarrow \bigcirc\ ((true \Rightarrow b) \Rightarrow ((true \Rightarrow \neg c) \wedge (true \Rightarrow \bigodot a)))$$

endreq

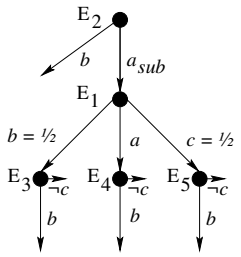
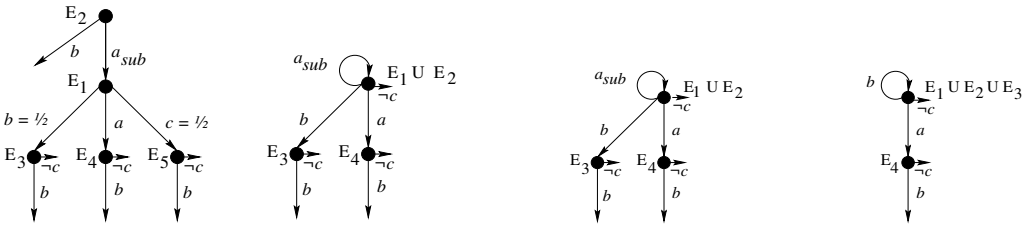


Figura 10.7: Grafo MUS $\mathcal{M}_1^{\mathcal{R}_{red}}$.

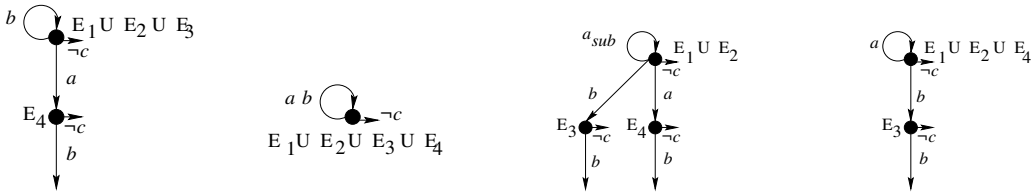
En la figura 10.8(a) se muestra el grafo MUS obtenido después de unir los estados E_1 y E_2. Esta unión supone que el estado E_1 sea un estado siguiente a sí mismo. Es decir, supone un bucle en el estado E_1 a través de una acción. Debido a que, en cualquier caso los estados siguientes a E_1 tienen especificada la acción c como no posible, dicha especificación se añade al estado E_1, eliminándose así el estado E_5.



(a) $\mathcal{M}_2^{\mathcal{R}_{red}}$ (b) $\mathcal{M}_3^{\mathcal{R}_{red}}$

Figura 10.8: Grafos MUS $\mathcal{M}_2^{\mathcal{R}_{red}}$ y $\mathcal{M}_3^{\mathcal{R}_{red}}$.

En la figura 10.8(b) se muestra el grafo MUS obtenido después de unir los estados E_1, E_2 y E_3. Se elimina la subespecificación de la acción que realiza el bucle en el estado E_1, porque ya existe una acción, en este caso la acción b, que realiza dicho bucle.



(a) $\mathcal{M}_4^{\mathcal{R}_{red}}$ (b) $\mathcal{M}_5^{\mathcal{R}_{red}}$

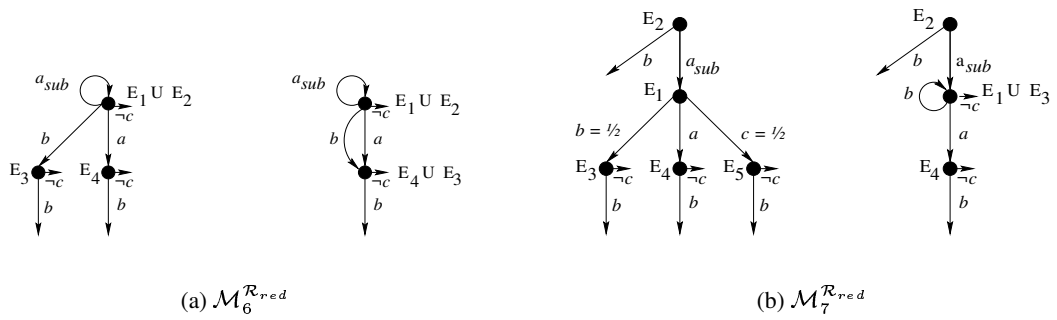
Figura 10.9: Grafos MUS $\mathcal{M}_4^{\mathcal{R}_{red}}$ y $\mathcal{M}_5^{\mathcal{R}_{red}}$.

En la figura 10.9(a) se muestra el grafo MUS obtenido después de unir el estado E_4 a los anteriores. El sistema resultante es la representación mínima del requisito SCTL $\mathcal{R}_1$, ya que, contiene un único estado. En la figura 10.9(b) se muestra el grafo MUS obtenido después de unir el estado E_4 al estado E_1 del grafo MUS $\mathcal{M}_2^{\mathcal{R}_{red}}$. A partir del grafo $\mathcal{M}_5^{\mathcal{R}_{red}}$ es posible obtener de

nuevo el grafo $\mathcal{M}_4^{\mathcal{R}_{red}}$, uniendo el estado E_3 al estado E_1.

Partiendo de nuevo del grafo inicial $\mathcal{M}_1^{\mathcal{R}_{red}}$, no es posible obtener un nuevo grafo uniendo el estado E_5 al estado E_1, ya que ambos estados son incompatibles. En E_1 se especifica E_5 como estado siguiente según la acción c –$E^{\{c,1\}} = E_5$–, mientras que dicha acción está especificada como no posible en E_5 –$E_5[c] = 0$–, por lo que no es posible unir ambos estados.

En la figura 10.10(a) se muestra el grafo MUS resultante de unir los estados E_3 y E_4 del grafo MUS $\mathcal{M}_2^{\mathcal{R}_{red}}$. Para obtener un nuevo grafo MUS, se vuelve al grafo MUS inicial $\mathcal{M}_1^{\mathcal{R}_{red}}$ y se une el estado E_3 al estado E_1, tal y como se muestra en la figura 10.10(b). Igual que en el grafo $\mathcal{M}_2^{\mathcal{R}_{red}}$, se suprime el estado E_5. Siguiendo con el proceso anterior, se obtiene el resto de grafos MUS correspondientes al requisito $\mathcal{R}_{red}$.



(a) $\mathcal{M}_6^{\mathcal{R}_{red}}$ (b) $\mathcal{M}_7^{\mathcal{R}_{red}}$

Figura 10.10: Grafos MUS $\mathcal{M}_6^{\mathcal{R}_{red}}$ y $\mathcal{M}_7^{\mathcal{R}_{red}}$.

Se han obtenido 24 grafos MUS diferentes representando al requisito $\mathcal{R}_{red}$. El apéndice A contiene la totalidad de grafos MUS obtenidos por el algoritmo de reducción. Hay que significar que el cálculo de dichos modelos no es tan costoso como parece *a priori*. En primer lugar, dicho cálculo se realizaría implícitamente en cada sistema donde se sintetiza el requisito $\mathcal{R}_{red}$; incluso varias veces dentro del mismo sistema, si el requisito ha de verificarse en varios estados del mismo. En segundo lugar, el algoritmo que calcula dichos grafos es muy simple, ya que, parte del grafo MUS obtenido por el algoritmo 6.6 de traducción SCTL-MUS, y une los distintos estados del mismo. Además, no es necesario generar todas las posibilidades, ya que, existen estados incompatibles que no pueden unirse formando un único estado.

A continuación, se muestra la lista de grafos MUS correspondientes al requisito $\mathcal{R}_{red}$, ordenada de menor a mayor, según el número de estados. El último lugar se corresponde con el grafo MUS $\mathcal{M}_1^{\mathcal{R}_{red}}$ de la figura 10.7, que se corresponde con el grafo MUS obtenido por el algoritmo 6.6 de traducción SCTL-MUS. Se ha suprimido el grafo MUS $\mathcal{M}_6^{\mathcal{R}_{red}}$, debido a que es equivalente al grafo $\mathcal{M}_4^{\mathcal{R}_{red}}$, ya que, en el estado $E_1 \bigcup E_2$ debe haber un bucle a sí mismo, y las dos acciones que pueden realizar ese bucle tienen asociado el estado $E_3 \bigcup E_4$, por lo que supone que se forme un único estado como en el grafo MUS $\mathcal{M}_4^{\mathcal{R}_{red}}$.

$$\{\mathcal{M}^{\mathcal{R}_{red}}\} = \{4, 3, 5, 8, 9, 11, 14, 2, 7, 10, 13, 15, 16, 18, 19, 21, 23, 12, 17, 20, 22, 24, 1\}$$

El nuevo algoritmo de síntesis que se propone consiste, por tanto, en aplicar el mismo algorit-

mo de síntesis 10.1 al conjunto de grafos MUS previamente calculados mediante el algoritmo de reducción de estados.

10.2.2 Ejemplo de Aplicación

A continuación, se repite el ejemplo de síntesis realizada en la sección 9.3.4, utilizando el algoritmo de síntesis 10.1 y la obtención previa de los grafos MUS correspondientes a los requisitos a sintetizar. En la figura 10.11 se muestra el grafo MUS del sistema en la fase actual del diseño, y a continuación, los requisitos a sintetizar.

req $\mathcal{R}_1$ is
$(true \Rightarrow a) \wedge (true \Rightarrow \odot\, b) \Rightarrow \bigcirc\, (true \Rightarrow b) \Rightarrow (true \Rightarrow \neg c)$
endreq

req $\mathcal{R}_2$ is
$(true \Rightarrow \neg c) \Rightarrow \bigcirc\, (true \Rightarrow \neg a)$
endreq

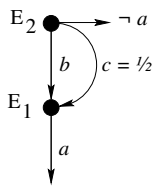


Figura 10.11: Grafo MUS $\mathcal{M}_{int}$ de un sistema en una fase intermedia del diseño.

Los grafos MUS correspondientes a $\mathcal{R}_2$ se muestran en la figura 10.12. Los correspondientes a $\mathcal{R}_1$ se muestran en la figura 10.13.

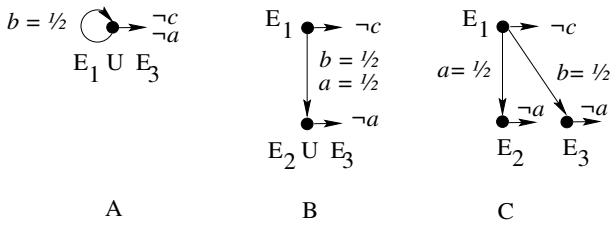


Figura 10.12: Grafos MUS $\{\mathcal{M}^{\mathcal{R}_2}\}$.

Para la síntesis de ambos requisitos, primero se sintetiza $\mathcal{R}_1$, eligiendo en primer lugar el grafo MUS $\mathcal{M}_A^{\mathcal{R}_1}$. Con dicho modelo de estados no es posible sintetizar $\mathcal{R}_2$, sea cual sea el grafo MUS de dicho requisito. Este proceso se repite hasta que es posible la síntesis de los dos requisitos. Esto se produce con el grafo MUS $\mathcal{M}_E^{\mathcal{R}_1}$, y el grafo MUS $\mathcal{M}_A^{\mathcal{R}_2}$. En la figura 10.14 se muestra el

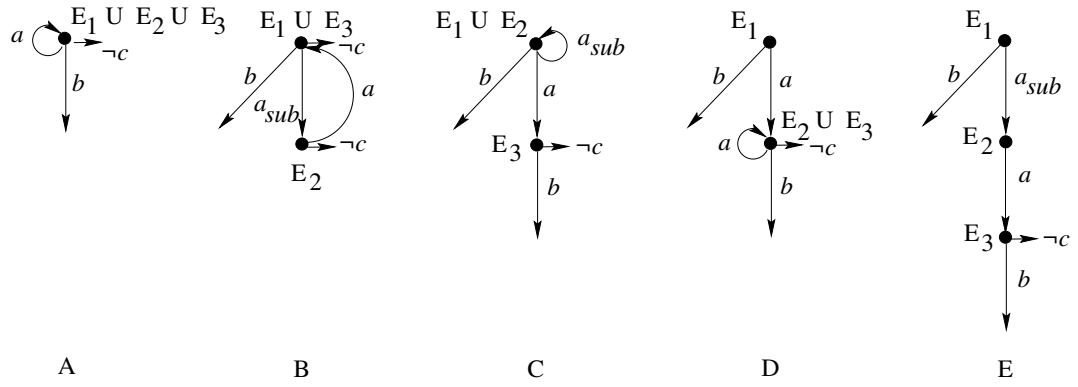


Figura 10.13: Grafos MUS $\{\mathcal{M}^{\mathcal{R}_1}\}$.

grafo MUS obtenido. Obsérvese que el resultado es el mismo que el obtenido por el algoritmo 9.1, mostrado en la figura 9.12, obteniéndose la solución con el menor número de estados.

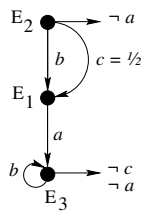


Figura 10.14: Resultado del Algoritmo de Síntesis III.

10.3 Algoritmo de Síntesis: En Busca de Eficiencia

Los algoritmos de síntesis 9.1 y 10.1 muestran una clara deficiencia derivada de la necesidad de evaluar de nuevo los requisitos previamente sintetizados. Esto se debe, tal y como se explicó en las secciones anteriores, a que la síntesis de un nuevo requisito puede conllevar la aparición de nuevos estados. Estos estados pueden ser estados de aplicabilidad de algún requisito previamente sintetizado. Naturalmente, estos nuevos estados no tienen porqué satisfacer la consecuencia impuesta por dichos requisitos, por lo que deben ser evaluados de nuevo y sintetizados si fuera necesario.

Para reducir el coste computacional que esto conlleva, se deberá obtener un algoritmo de síntesis que no necesite sintetizar más que una vez los requisitos en el proceso de síntesis. Para ello, es necesario poder mantener información adicional en los grafos MUS, de manera que, a partir de dicha información, sea posible introducir nuevos estados pero sujetos a las restricciones impuestas por los requisitos previamente sintetizados.

La solución pasa por mantener, en el grafo MUS, información acerca de las restricciones a los estados anteriores de uno dado, ya que para los estados siguientes esta información ya se

mantiene. Sin embargo, existe una diferencia importante entre ambos, derivada de la limitación en el número de los estados siguientes, ya que, en el peor de los casos, habrá uno por cada acción del conjunto Λ. Sin embargo, los estados anteriores, no están acotados, por no estarlo los estados del sistema. Es, por tanto, imposible almacenar a *priori* dichos estados anteriores, tal y como se hace con los estados siguientes. La solución propuesta crea un nuevo grafo MUS para cada estado que representa las restricciones impuestas para sus estados anteriores. Cada vez que aparezca un nuevo estado, deberá solaparse –basta con aplicar el algoritmo de síntesis 10.1– el modelo de estados que almacena las restricciones de sus estados anteriores en el nuevo estado anterior.

Naturalmente, dicha solución impone una mayor carga, tanto en el almacenamiento como en la síntesis, ya que, el número de estados del sistema, como mínimo, se duplica. De todas maneras, los estados correspondientes a los estados anteriores sólo deben almacenar las restricciones de los estados anteriores, por lo que sólo deberán ser considerados en el algoritmo de aplicabilidad cuando se evalúe el operador *Antes*.

Por tanto, el nuevo algoritmo de síntesis, que permite sintetizar una única vez los requisitos SCTL, impone las siguientes modificaciones sobre los algoritmos 9.1 y 10.1:

1. Cada vez que se cree un nuevo estado, se creará un grafo MUS representando las restricciones que deben satisfacer los estados anteriores a dicho estado. Inicialmente, será un grafo MUS totalmente subespecificado.

2. Cuando se evalúe un requisito con el operador temporal *Antes*, se considerarán dichos estados como estados de aplicabilidad, sintetizando en ellos los requisitos correspondientes.

3. Cuando se realice un bucle hacia un estado ya existente del sistema, el estado origen del bucle se solapará al grafo MUS que almacena las restricciones de estado anterior correspondiente al estado destino del bucle. Dicho solapamiento será una nueva condición de estado compatible.

De esta manera, se obtienen dos nuevos algoritmos, con las mismas características que los algoritmos 9.1 y 10.1 salvo que no necesitan evaluar los requisitos SCTL previamente sintetizados. A dichos algoritmos se les ha denominado algoritmos de síntesis I (II) **modificados**.

10.4 Algoritmo de Síntesis: Independencia de SCTL

Todos los algoritmos de síntesis vistos hasta ahora dependen del formalismo utilizado en la captura de requisitos, en este caso SCTL. El algoritmo 9.1 depende obviamente de SCTL, ya que, utiliza el algoritmo de traducción SCTL-MUS. El algoritmo 10.1 necesita obtener los estados de aplicabilidad del requisito SCTL en el grafo MUS del requisito y en el del sistema sobre el que se está sintetizando dicho requisito.

Un algoritmo de síntesis independiente de SCTL se podría aplicar a partir, únicamente, de grafos MUS; pudiendo utilizar cualquier otra técnica de especificación para obtener dichos modelos

de estados. De la misma manera, este algoritmo podría utilizarse para realizar verificación a nivel de grafos MUS. Esta posibilidad se apunta en la sección 10.5.

El algoritmo de síntesis 10.2 se basa únicamente en los grafos MUS y en sus características. Dicho algoritmo va solapando cada uno de los estados del requisito en los estados del sistema. Los estados de aplicabilidad que puedan aparecer en el sistema y que no pueden ser reflejados en el grafo MUS del requisito, es decir, la particularización del requisito en el grafo del sistema, se obtiene utilizando un algoritmo de traducción SCTL-MUS **modificado**, en el que se introducen las restricciones de estados anteriores, tal y como se explicó en la sección 10.3 para el algoritmo de síntesis.

10.4.1 Pseudocódigo del Algoritmo

El pseudocódigo del nuevo algoritmo de síntesis (ver algoritmo 10.2), recibe como parámetros de entrada los estados iniciales del grafo MUS del sistema $E_h^{\mathcal{M}}$ y el del requisito a sintetizar $E_h^{\mathcal{R}}$. Los estados compatibles se calculan, de manera similar a cómo lo hacia el algoritmo 10.1, utilizando la notación $E_i \sim E_j$ para indicar que ambos estados son compatibles, simplificando así la notación.

Además, es necesario representar, en el grafo MUS del requisito, toda la información relativa a su comportamiento, ya que no se utiliza la especificación SCTL. Por ello, el grafo MUS del requisito contiene las restricciones de estados anteriores, lo que hace que se modifique la operación unión entre estados, que debe, además, realizar la unión del estado anterior al grafo de restricciones correspondiente. Esta operación se denota por $\bigcup_{res}$.

El resto del algoritmo es similar a los anteriores, salvo en la manera de calcular estados de aplicabilidad. Al no disponer de la especificación SCTL, se solapan todos los estados del grafo MUS del requisito, recorriendo los estados siguientes al inicial –ver paso [3]– y los estados anteriores al mismo –ver paso [4]–, en cada iteración del algoritmo.

El algoritmo 10.2 puede modificarse, al igual que los algoritmos 9.1 y 10.1, obteniendo así un algoritmo de síntesis independiente de SCTL, con o sin evaluación de requisitos previamente sintetizados.

10.5 Conclusiones de los Algoritmos de Síntesis Obtenidos

Inicialmente, se obtuvo un algoritmo de síntesis similar al algoritmo de traducción. Dicho algoritmo se modificó según las características propias de la síntesis incremental, expuestas en la sección 9.3.1. Las principales características del algoritmo 9.1 obtenido son:

- Recorre los requisitos de manera similar al algoritmo de traducción, a partir de su especificación SCTL.

Algoritmo 10.2 Algoritmo de Síntesis III $(\{E_h^{\mathcal{M}}\}, \{E_h^{\mathcal{R}}\})$

[1] Si ya están solapados $(E_h^{\mathcal{M}}, E_h^{\mathcal{R}}\}$ devolver OK;
[2] Si $E_h^{\mathcal{M}} = E_h^{\mathcal{M}} \cup_{res} E_h^{\mathcal{R}} \notin \Psi^m$ devolver ERROR;
[3] Para toda $a_i \in \Lambda$, $E_j^{\mathcal{R}} \in \mathcal{M}_{\mathcal{R}}$ tal que $d^{\mathcal{R}}[a_i][E_h^{\mathcal{R}}][E_j^{\mathcal{R}}] = 1$ y $E_j^{\mathcal{R}}$ no ha sido solapado:
  [a] Si $\exists E_j^{\mathcal{M}}$ tal que $d^{\mathcal{M}}[a_i][E_h^{\mathcal{M}}][E_j^{\mathcal{M}}] = 1$, **Algoritmo de Síntesis III $(E_j^{\mathcal{M}}, E_j^{\mathcal{R}})$**;
  [b] En caso contrario:
    [i] $\{E_{comp}\} = \{E_j^{\mathcal{M}} \in \mathcal{M} : E_j^{\mathcal{M}} \sim E_j^{\mathcal{R}}\}$ ordenados por mínima subespecificación;
    [ii] Para todo $E_j^{\mathcal{M}} \in \{E_{comp}\}$: **Algoritmo de Síntesis III $(E_j^{\mathcal{M}}, E_j^{\mathcal{R}})$**;
    [iii] Si se devuelve OK, ir al paso **[c]**;
    [iv] Si quedan estados compatibles volver al paso **[ii]**;
    [v] En caso contrario:
      [A] Nuevo estado $E_{n+1}^{\mathcal{M}}$;
      [B] $d^{\mathcal{M}}[a_i][E_h^{\mathcal{M}}][E_{n+1}^{\mathcal{M}}] = 1.$;
      [C] Si **Algoritmo de Síntesis III $(E_{n+1}^{\mathcal{M}}, E_j^{\mathcal{R}})$** devolver ERROR;
  [c] Volver al paso **[3]**;
[4] Para todo $E_j^{\mathcal{R}}$ tal que $d^{\mathcal{R}}[a_{sub}][E_j^{\mathcal{R}}][E_h^{\mathcal{R}}] = 1$ y no haya sido solapado:
  [a] $\{E_{comp}\} = \{E_j^{\mathcal{M}} \in \mathcal{M} : E_j^{\mathcal{M}} \sim E_j^{\mathcal{R}}\}$ ordenados por mínima subespecificación;
  [b] Para todo $E_j^{\mathcal{M}} \in \{E_{comp}\}$: **Algoritmo de Síntesis III $(E_j^{\mathcal{M}}, E_j^{\mathcal{R}})$**;
  [c] Si se devuelve OK, ir al paso **[4]**;
  [d] Si quedan estados compatibles volver al paso **[4] [b]**;
  [e] En caso contrario:
    [i] Nuevo estado $E_{n+1}^{\mathcal{M}}$;
    [ii] $d^{\mathcal{M}}[a_{sub}][E_{n+1}^{\mathcal{M}}][E_h^{\mathcal{M}}] = 1$. Anotar restricciones estado anterior;
    [iii] Si **Algoritmo de Síntesis III $(E_{n+1}^{\mathcal{M}}, E_j^{\mathcal{R}})$** devolver ERROR;
  [f] Volver al paso **[4]**;
[5] Devolver OK;

- Busca estados compatibles, lo que permite sintetizar el requisito con el menor número de modificaciones en el sistema.
- La utilización de estados compatibles impone tomas de decisión que pueden rehacerse a lo largo del proceso de síntesis. Es necesario, por tanto, almacenar dichas tomas de decisión.
- Si no es posible encontrar un estado compatible, se crea un nuevo estado, lo que supone la máxima pérdida de subespecificación.
- El algoritmo detecta inconsistencias en los requisitos especificados. Esto sucede cuando no es capaz de tomar una decisión para sintetizar el requisito especificado.

La última característica indicada, sugiere que es posible utilizar el algoritmo de síntesis como algoritmo de verificación. El resultado sería *verdadero* si es posible sintetizar el requisito, y *falso*

en caso contrario. Una de las líneas futuras de este trabajo es avanzar en un mecanismo de verificación basado en los algoritmos de síntesis propuestos, de manera que pueda, además, aportar información de grados de satisfacción intermedios, como el algoritmo de verificación propuesto en el capítulo 8.

El algoritmo 10.1 propone una nueva estrategia en la síntesis de requisitos SCTL, con el objetivo de poder reutilizar la síntesis de un mismo requisito en distintos sistemas:

- Utiliza el grafo MUS del requisito SCTL obtenido por el algoritmo de traducción. Dicho grafo contiene la información común a todas las síntesis de dicho requisito, y por tanto puede reutilizarse.

- La síntesis se basa en recorrer el grafo MUS del requisito y el del sistema. Ambos deben encajar en la parte de la premisa –condición de aplicabilidad–, y deben solaparse en la parte de la consecuencia. La síntesis de la premisa se realiza una única vez por el algoritmo de traducción, mientras que la síntesis de la consecuencia se particulariza para cada sistema en concreto.

- Para ello, utiliza la unión entre estados. Cuando la unión de dos estados da como resultado un elemento externo del conjunto Ψ, dicha unión no es posible, detectando así inconsistencias entre los requisitos especificados.

- Este algoritmo sigue utilizando las tomas de decisión al igual que el algoritmo anterior.

A partir de dicho algoritmo, se propone una variación del mismo que permita un mayor grado de reutilización:

- Se propone calcular, previamente, un conjunto de grafos MUS que representan un requisito SCTL. La diferencia entre ellos se basa en la particularización que sufren dichos requisitos al ser sintetizados sobre un sistema en concreto. La utilización en la síntesis de estados compatibles –bucles–, puede realizarse previamente si el estado compatible elegido es un estado en el que se solapa el requisito.

- La solución propuesta consiste en utilizar el mismo algoritmo 10.1 pero con los diferentes grafos MUS del requisito SCTL. En primer lugar, se intentará sintetizar el grafo MUS con el menor número de estados, hasta que finalmente se intentará con el grafo MUS obtenido por el algoritmo de traducción –máxima pérdida de subespecificación–.

- La reutilización obtenida aumenta considerablemente con el caso anterior, ya que los bucles a estados del grafo MUS del requisito se realizan una única vez.

- Sigue siendo necesario la búsqueda de estados compatibles en el grafo MUS del sistema, ya que pueden existir estados del mismo donde no se solapa ningún estado del requisito, y por tanto, dichos bucles no se contemplarían en los distintos grafos MUS del mismo.

A continuación, se estudia la posibilidad de hacer más eficientes los algoritmos anteriores. Para ello, se propone almacenar la información necesaria en el grafo MUS del sistema para evitar la evaluación de requisitos SCTL previamente sintetizados:

- La utilización de estados compatibles hace que aparezcan en el sistema nuevos estados de aplicabilidad para requisitos previamente especificados. Esto es debido a que pueden aparecer estados anteriores cada vez que se realiza un bucle en el sistema, lo que supondría evaluar de nuevo dichos requisitos.

- Este problema no existe para los estados siguientes, ya que, sólo puede existir uno por cada acción posible, por lo que dicha información se almacena en cada estado del sistema –utilizando estados representantes para que sea más eficiente–.

- La solución propuesta es almacenar, para cada estado del sistema, el conjunto de restricciones de sus estados anteriores. Dichas restricciones no se limitan a un estado, por lo que se almacena un grafo MUS para cada estado anterior. Estas restricciones deberán ser tenidas en cuenta al calcular los estados compatibles.

Finalmente, se explora la posibilidad de aislar el proceso de síntesis del formalismo SCTL utilizado para la captura de requisitos. Para ello, se propone un nuevo algoritmo 10.2 que solapa el comportamiento de dos grafos MUS, el correspondiente al requisito especificado y el del sistema:

- El grafo MUS del requisito puede haber sido obtenido por cualquier método ya que no contiene ninguna información ajena a las especificaciones MUS.

- La diferencia fundamental con respecto al algoritmo 10.1 reside en que no puede utilizar la información del requisito SCTL para calcular los estados de aplicabilidad.

- Para solucionar ese problema, se recorre el grafo MUS completo del requisito, hasta que se solapan todos sus estados en algún estado del grafo MUS del sistema.

- La solución propuesta necesita que el grafo MUS contenga toda la información de los estados de aplicabilidad del requisito, por lo que se utiliza el grafo MUS del requisito con las restricciones de los estados anteriores, tal y como se explicó en el sección anterior.

10.6 Algoritmo de Traducción SCTL-MUS: En Busca de Reutilización

En las secciones anteriores se ha desarrollado un algoritmo de síntesis que, finalmente, es capaz de sintetizar un conjunto de requisitos expresados mediante grafos MUS, independientemente del origen de dichos grafos. Esto proporciona una gran flexibilidad al trabajo desarrollado, ya que permite utilizar técnicas diferentes a SCTL para obtener y capturar los requisitos del sistema.

Sin embargo, en el trabajo de esta tesis se ha optado por utilizar un formalismo con las características de SCTL, por lo que, en esta sección se estudia la posibilidad de mejorar el algoritmo de traducción SCTL-MUS 6.6 expuesto en el capítulo 6.

Dicho algoritmo obtiene un grafo MUS a partir de un requisito SCTL. Para ello, utiliza un algoritmo auxiliar que se corresponde con la traducción de requisitos atómicos. Sin embargo, a pesar de dicho algoritmo auxiliar, cada vez que se afronta la traducción de un nuevo requisito SCTL, no se reutiliza nada de las traducciones de dichos requisitos atómicos o de otros requisitos que lo componen.

Esta es la razón por la que se plantea obtener un nuevo algoritmo de traducción, orientado principalmente a la reutilización de sistemas que comparten requisitos o subrequisitos componentes de los primeros. La idea surge del nuevo algoritmo de síntesis 10.2. Dicho algoritmo parte de dos grafos MUS, y los "solapa" en uno solo, de manera que el resultado es un nuevo grafo MUS que satisface en el estado especificado un nuevo requisito representado en MUS. Para ello, basta con indicar al algoritmo cuál es el estado donde se desea solapar el nuevo requisito, y cuál es el estado inicial del grafo MUS que representa dicho requisito.

Para poder utilizar esta filosofía en la traducción de requisitos SCTL a MUS, bastaría con obtener inicialmente los tres modelos de estados subespecificados correspondientes a los requisitos atómicos –de hecho ya se han obtenido en el capítulo 6–, y solaparlos en los estados indicados por los operadores lógicos o temporales que los unen formando el requisito SCTL a traducir.

Es decir, si dos requisitos atómicos están unidos por el operador lógico And, para obtener el grafo MUS correspondiente al nuevo requisito, bastará con solapar los modelos de estados de los requisitos atómicos en los dos estados iniciales de los mismos. Si estuviesen unidos por un operador temporal, habría que solapar el requisito consecuencia (en su nodo inicial) en los estados de aplicabilidad correspondientes sobre el requisito premisa.

De esta manera, no sólo se podría reutilizar la traducción de los requisitos SCTL más simples, sino que la reutilización puede extenderse a requisitos SCTL más complejos, ya que éstos tendrán un grafo MUS con un estado inicial, que se puede solapar en los estados que indique el operador que une dicho requisito con cualquier otro, formando así requisitos más complejos.

El grado de reutilización obtenido es altísimo, tal y como se muestra a continuación. Dado un requisito SCTL, dicho requisito se descompone en todos y cada uno de los requisitos componentes. Se halla el grafo MUS de cada uno de ellos, de manera que si un requisito contiene otros requisitos más simples –siempre es así, salvo en los requisitos atómicos–, el grafo MUS se obtiene a partir del grafo MUS de los primeros. Una vez obtenido el grafo MUS de dicho requisito, éste puede solaparse en cualquier grafo MUS sin tener que volver a realizar ningún cálculo sobre el requisito SCTL, ni sobre el grafo MUS correspondiente. A medida que el sistema se enriquece con la especificación de requisitos SCTL, la obtención de los modelos de estados será inmediata, ya que el sistema habrá "aprendido" el grafo MUS correspondiente a los requisitos que lo componen.

10.6.1 Algoritmo de Traducción SCTL-MUS II

En la sección anterior se ha justificado la realización de un nuevo algoritmo de traducción, basado en el algoritmo de síntesis independiente de SCTL. Sin embargo, la utilización de dicho algoritmo difiere en un pequeño detalle. Mientras que para la obtención de un nuevo grafo MUS consistente con un nuevo requisito se utiliza el criterio de mínima pérdida de subespecificación, lo que permite sintetizar los sistemas más simples posibles –menor número de estados–; en la síntesis de un grafo MUS que represente el comportamiento de un requisito SCTL es necesario utilizar el criterio contrario, es decir, el criterio de máxima subespecificación.

Esto es así debido a que el grafo MUS obtenido debe representar el comportamiento del requisito SCTL especificado sin ninguna otra restricción. En el momento de solaparlo a un sistema en concreto será cuando se particularice su comportamiento, de manera que suponga los mínimos cambios posibles en dicho sistema; pero el grafo MUS correspondiente al requisito debe conservar toda la expresividad dada por SCTL.

De esta manera, en el algoritmo de síntesis, en vez de buscar nodos compatibles en el grafo del sistema, siempre que sea necesario se creará un nuevo estado, que es equivalente a un criterio de pérdida de máxima subespecificación, ya que, ésta es la decisión tomada cuando no existe ningún estado compatible en el que solapar de manera consistente el requisito.

Finalmente, es necesario introducir las restricciones en el grafo MUS del requisito para poder independizar el algoritmo de síntesis de SCTL, tal y como se explicó en la sección 10.4. Para ello, basta con introducir las restricciones en los modelos de estados de los requisitos atómicos, ya que, el resto se sintetizan a partir de éstos mediante el algoritmo de síntesis, donde se van anotando dichas restricciones.

Algoritmo 10.3 Algoritmo de Traducción SCTL-MUS II ($\overline{\mathcal{R}} = \{\overline{\mathcal{R}}[0], ..., \overline{\mathcal{R}}[n]\}, E_h$)

[1] Si ya se ha traducido $\overline{\mathcal{R}}$ en el estado E_h de $\mathcal{M}$, devolver OK;
[2] Si ya ha sido calculado $\mathcal{M}_{\mathcal{R}}$:
  [a] Algoritmo de Síntesis III $(\boldsymbol{E_h, E_1^{\mathcal{R}}, max_{subesp}})$;
  [b] Fin;
[3] $\{\overline{\mathcal{R}}_{sub_1}, \overline{\mathcal{R}}_{sub_2}\}$ = **Algoritmo de Partición** $(\overline{\boldsymbol{\mathcal{R}}})$;
[4] Algoritmo de Traducción SCTL-MUS II $(\overline{\boldsymbol{\mathcal{R}}}_{\boldsymbol{sub_1}}, \mathbf{E_h})$;
[5] Si $\overline{\mathcal{R}}[0] = \wedge$:
  [a] Algoritmo de Traducción SCTL-MUS II $(\overline{\boldsymbol{\mathcal{R}}}_{\boldsymbol{sub_2}}, \mathbf{E_h})$;
  [b] Fin;
[6] Si $\overline{R}[0] \in \Theta$:
  [a] $\{E_{aplic}\}$ = **Algoritmo de Aplicabilidad** $(\overline{\boldsymbol{\mathcal{R}}}, \mathbf{E_h})$;
  [b] $\{E_{aplic}\} = \{E_{aplic}\} \bigcup$ **Algoritmo de Aplicabilidad Potencial** $(\overline{\boldsymbol{\mathcal{R}}}, \mathbf{E_h})$;
  [c] $\forall E_j \in \{E_{aplic}\}$, **Algoritmo de Traducción SCTL-MUS II** $(\overline{\boldsymbol{\mathcal{R}}}_{\boldsymbol{sub_2}}, \mathbf{E_j})$;
  [d] Fin;

10.7 Traducción MUS E-LOTOS

Según el modelo de desarrollo software propuesto en el capítulo 3, una vez que el prototipo MUS del sistema satisface los requisitos SCTL especificados por el usuario, se pasa a una nueva etapa –denominada de refinamiento–, en la que se transforma la arquitectura inicial del sistema mediante el entorno transformacional LIRA.

La versión actual de LIRA aplica las transformaciones a especificaciones E-LOTOS, por lo que es necesario traducir el prototipo MUS a una especificación E-LOTOS. La traducción de los grafos MUS a E-LOTOS básico expandido[2] es inmediata a partir de su representación gráfica.

Sin embargo, es necesario eliminar previamente los elementos subespecificados de los grafos MUS, ya que, éstos no pueden expresarse en E-LOTOS. Para ello, se define una fase de **pérdida de subespecificación**, en la que se transforma un grafo MUS a un grafo totalmente especificado. A continuación, se enumeran las distintas pérdidas de subespecificación que deben realizarse en un grafo MUS para poder obtener una especificación en E-LOTOS:

1. Todas las acciones subespecificadas se especificarán como no posibles, obteniendo así, el sistema más simple posible.

2. Todos los arcos a estados subespecificados se transformarán en arcos especificados hacia un estado de parada.

3. Los estados representantes se convertirán en estados reales del sistema, cambiando el valor de los arcos que llegan a dichos estados, de $\frac{3}{4}$ a 1.

4. A los arcos cuya acción asociada está subespecificada, se les asignará una acción especificada como posible en el estado origen de dicho arco.

En el ejemplo 10.1 se muestra la especificación E-LOTOS del grafo MUS $\mathcal{M}$ de la figura 10.15.

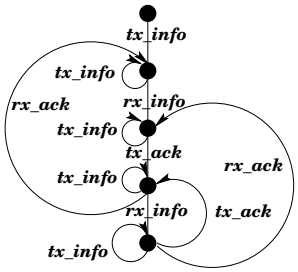


Figura 10.15: Grafo MUS $\mathcal{M}$ de un proceso Tx_Rx.

[2]E-LOTOS básico es un subconjunto de E-LOTOS en el que se ha eliminado el paso de valores. E-LOTOS básico expandido es un subconjunto de E-LOTOS básico en el que sólo se utiliza el prefijo de acción, la elección no determinista y la recursión.

Ejemplo 10.1 Ejemplo de Traducción MUS E-LOTOS.

```
process Tx_Rx [tx_info, rx_info, tx_ack, rx_ack] : noexit :=
    tx_info; S_1
where
    process S_1 [tx_info, rx_info] : noexit :=
        tx_info; S_1
        []
        rx_info; S_2
    where
        process S_2 [tx_info, tx_ack] : noexit :=
            tx_info; S_2
            []
            tx_ack; S_3
        where
            process S_3 [tx_info, rx_info, rx_ack] : noexit :=
                tx_info; S_3
                []
                rx_info; S_4
                []
                rx_ack; S_1
            where
                process S_4 [tx_info, tx_qck, rx_ack] : noexit :=
                    tx_info; S_4
                    []
                    tx_ack; S_3
                    []
                    rx_ack; S_2
                endproc
            endproc
        endproc
    endproc
endproc
```

Capítulo 11

Diseño de la Arquitectura

11.1 Introducción

El comportamiento de un sistema concurrente puede descomponerse en dos partes: una parte **Funcional** que contiene las acciones relativas a la manipulación de datos y las que realizan la función encomendada a cada uno de los procesos del sistema, sin tener en cuenta su relación con el resto de procesos; y una segunda parte, denominada de **Sincronización**, que contiene las restricciones necesarias en el ordenamiento temporal de las acciones que realizan los diferentes procesos concurrentes que componen el sistema.

La descomposición de un sistema en estas dos partes permite concentrar la complejidad de la concurrencia en la parte de sincronización. Esta parte suele ser bastante pequeña, en comparación con la parte funcional de cada uno de los procesos del sistema, si bien, generalmente, es más compleja.

Un sistema concurrente genérico estará formado por un conjunto de procesos $\mathcal{S} = \{\mathcal{P}_1, ..., \mathcal{P}_s\}$, y un conjunto de requisitos que expresan las restricciones en la ejecución concurrente de los mismos. Según la metodología SCTL-MUS expuesta, cada proceso $\mathcal{P}_i$ se sintetiza mediante la especificación incremental del conjunto de requisitos SCTL que debe satisfacer. Falta, por tanto, completar la metodología con un mecanismo que permita **expresar** y **sintetizar** la sincronización entre cada uno de los procesos.

Para ello, es necesario definir operadores arquitectónicos que permitan expresar la arquitectura del sistema, identificando la sincronización entre cada uno de los procesos componentes. En este trabajo, se optó por utilizar los operadores arquitectónicos definidos en E-LOTOS [ISO89b]: *entrelazamiento*, *sincronización total* y *sincronización parcial*. Esta decisión se basa, tal y como se explicó en el capítulo 3, en la integración de la metodología propuesta con el entorno transformacional LIRA, así como en la inmediata traducción del formalismo MUS a E-LOTOS.

Por tanto, la arquitectura de un sistema concurrente genérico podrá expresarse mediante el operador arquitectónico *sincronización parcial*: $\mathcal{S} = \mathcal{P}_1 |[\Lambda_{sinc}]| \mathcal{P}_2$, siendo Λ_{sinc} el conjunto de acciones de sincronización. Sin embargo, las características anteriormente citadas, sugieren

independizar la parte de sincronización de la parte funcional, aislando así la complejidad de la concurrencia. Una posible solución consiste en combinar los operadores *entrelazamiento* y *sincronización parcial*: $\mathcal{S} = (\mathcal{P}_1 ||| \mathcal{P}_2) || [\Lambda_{sinc}] || \mathcal{Y}$, siendo $\mathcal{Y}$ un proceso que contiene las restricciones impuestas por la sincronización de los procesos $\mathcal{P}_1$ y $\mathcal{P}_2$.

La solución propuesta, se basa, por tanto, en desarrollar algoritmos que permitan la síntesis automática de procesos de sincronización a partir del conjunto de requisitos que expresan dichas restricciones; de manera que en la arquitectura de un sistema concurrente queden perfectamente identificadas las partes funcionales (procesos componentes) y la parte de sincronización (proceso sincronizador).

11.1.1 Requisitos de Sincronización

Dado un sistema $\mathcal{S}$ compuesto por dos procesos $\mathcal{P}_1$ y $\mathcal{P}_2$ –la extensión a un número mayor de procesos es inmediata–, se puede expresar la evolución concurrente de ambos procesos mediante el operador arquitectónico *entrelazamiento*[1] $\mathcal{P}_1 \mid\mid\mid \mathcal{P}_2$. Por tanto, si el sistema carece de restricciones de sincronización, el sistema resultante podría expresarse por $\mathcal{S}_{ent} = \mathcal{P}_1 \mid\mid\mid \mathcal{P}_2$, obteniendo una expresión del sistema con estructura.

Sin embargo, pueden existir comportamientos de $\mathcal{S}_{ent}$ no permitidos, debido a restricciones en la evolución concurrente de los dos procesos que lo componen. En primer lugar, se va a estudiar la naturaleza de estas restricciones, a fin de obtener la influencia de las mismas en el proceso entrelazamiento $\mathcal{S}_{ent}$.

La parte de sincronización de un sistema concurrente no contiene componentes funcionales del mismo, es decir, no aporta nuevos comportamientos al sistema, sino todo lo contrario; los limita prohibiendo ciertas evoluciones. Por tanto, si se especifican requisitos de sincronización sobre los procesos $\mathcal{P}_1$ y $\mathcal{P}_2$, dichos requisitos **sólo** pueden afectar al proceso $\mathcal{S}_{ent}$ eliminando comportamientos del mismo, y nunca añadiéndolos.

En la metodología SCTL-MUS, cada proceso se modela mediante un conjunto de requisitos SCTL que satisface, y un grafo MUS que representa su comportamiento. A partir de los grafos MUS de los procesos $\mathcal{P}_1$ y $\mathcal{P}_2$, es posible obtener –de manera automática– un grafo MUS del proceso entrelazamiento $\mathcal{S}_{ent}$. Según lo expuesto en el párrafo anterior, la especificación de requisitos de sincronización sólo puede suponer la eliminación de comportamientos en dicho proceso. Es decir, sólo puede suponer la eliminación de alguna de las ramas del grafo que representa su comportamiento.

La especificación de requisitos de sincronización consiste, por tanto, en especificar las ramas del proceso entrelazamiento que deben ser eliminadas. Para ello, basta con especificar que una acción $a_i \in \Lambda'$ debe ser *no posible* en un conjunto de estados E_{aplic} del grafo MUS de $\mathcal{S}_{ent}$. Es posible, por tanto, utilizar requisitos SCTL para especificar las restricciones o requisitos de

[1] $\mathcal{P}_1 ||| \mathcal{P}_2$: $\dfrac{\mathcal{P}_1 \xrightarrow{a} \mathcal{P}_1'}{\mathcal{P}_1 ||| \mathcal{P}_2 \xrightarrow{a} \mathcal{P}_1' ||| \mathcal{P}_2}$ $\dfrac{\mathcal{P}_2 \xrightarrow{a} \mathcal{P}_2'}{\mathcal{P}_1 ||| \mathcal{P}_2 \xrightarrow{a} \mathcal{P}_1 ||| \mathcal{P}_2'}$

sincronización, ya que permiten expresar que una (o varias) acciones no deben ser posibles en un conjunto de estados de aplicabilidad de su operador temporal.

Por tanto, la especificación de un sistema concurrente $\mathcal{S}$ mediante la metodología SCTL-MUS, se realizará en tres etapas:

1. Especificación de cada uno de los procesos componentes del sistema, de los que tiene conocimiento el usuario o diseñador.

2. Especificación de un conjunto de requisitos de sincronización –$\{\mathcal{R}_{sinc}\}$–. Dichos requisitos sólo pueden restringir la evolución concurrente de los procesos anteriormente especificados.

3. Síntesis del sistema resultante $\mathcal{S}$ que coincide con el proceso entrelazamiento $\mathcal{S}_{ent}$ salvo en las ramas eliminadas por los requisitos de sincronización.

11.1.2 Síntesis de un Proceso Sincronizador

La obtención del sistema resultante $\mathcal{S}$ puede realizarse a partir del proceso $\mathcal{S}_{ent}$ y la definición de algoritmos que permitan eliminar las ramas prohibidas por los requisitos de sincronización:

$$\mathcal{S} = \mathcal{S}_{ent} + \textbf{Algoritmo de Eliminación de Ramas } (\{\mathcal{R}_{sinc}\})$$

A partir del proceso $\mathcal{S}$ es posible realizar tareas de análisis de integración, ya que se representa, en un único proceso, el comportamiento del sistema especificado. Sin embargo, no es posible realizar un análisis parcial, ya que no es posible expresar dicho proceso a partir de sus procesos componentes $\mathcal{P}_1$ y $\mathcal{P}_2$, tal y como sucedía con el proceso entrelazamiento –$\mathcal{S}_{ent} = \mathcal{P}_1 \,|||\, \mathcal{P}_2$–. Se ha perdido, por tanto, la información de estructura del sistema especificado.

La solución ideal sería poder expresar el proceso resultante $\mathcal{S}$ como un proceso compuesto por los procesos $\mathcal{P}_1$, $\mathcal{P}_2$ y un nuevo proceso, denominado **proceso sincronizador** –$\mathcal{Y}$–, que restringiera la evolución concurrente de ambos según el conjunto de requisitos de sincronización $\mathcal{R}_{sin}$. Para ello, se puede utilizar el operador arquitectónico *sincronización total*[2] tal y como se muestra a continuación:

$$\mathcal{S} = \mathcal{S}_{ent} \,||\, \mathcal{Y} \;=\; (\mathcal{P}_1 \,|||\, \mathcal{P}_2) \,||\, \mathcal{Y}$$

Una primera solución es identificar el proceso sincronizador con el proceso resultante, ya que: $\mathcal{S} = \mathcal{S}_{ent} \,||\, \mathcal{S}$. Esta solución permite obtener el sistema final $\mathcal{S}$ con estructura, en función de sus procesos componentes. Sin embargo, no parece la solución más adecuada, ya que supone que toda la información del sistema resultante es de sincronización.

El problema anterior surge por realizar la sincronización total con el proceso sincronizador. Dicha sincronización puede reducirse únicamente a las acciones que intervienen en la sincronización –Λ_{sinc}–, ya que, el proceso sincronizador sólo debe expresar el comportamiento restringido

[2] $\mathcal{P}_1||\mathcal{P}_2$: $\frac{\mathcal{P}_1 \xrightarrow{a} \mathcal{P}_1',\ \mathcal{P}_2 \xrightarrow{a} \mathcal{P}_2'}{\mathcal{P}_1||\mathcal{P}_2 \xrightarrow{a} \mathcal{P}_1'||\mathcal{P}_2'}$

de la evolución concurrente de los procesos involucrados. La solución propuesta utiliza el operador arquitectónico *sincronización parcial*[3] para obtener el proceso sincronizador:

$$\mathcal{S} = \mathcal{S}_{ent} \mid [\Lambda_{sinc}] \mid \mathcal{Y} \;=\; (\mathcal{P}_1 \mid\mid\mid \mathcal{P}_2) \mid [\Lambda_{sinc}] \mid \mathcal{Y}$$

En la siguiente sección se propone un algoritmo, denominado **algoritmo de sincronización**, que obtiene el conjunto de acciones de sincronización Λ_{sinc} y el proceso sincronizador $\mathcal{Y}$. Esto permite, por una parte, obtener una especificación con estructura del sistema especificado; y por otra, realizar un análisis de integración sobre el sistema resultante $\mathcal{S}$, y un análisis parcial sobre sus componentes: los procesos $\mathcal{P}_1$, $\mathcal{P}_2$ y el proceso sincronizador $\mathcal{Y}$. Obsérvese que el proceso sincronizador se sintetiza automáticamente a partir de los requisitos de sincronización y los procesos concurrentes especificados.

11.2 Algoritmo de Sincronización

La síntesis del proceso global $\mathcal{S}$ es trivial a partir del proceso entrelazamiento $\mathcal{S}_{ent} = \mathcal{P}_1 \mid\mid\mid \mathcal{P}_2$ y de los requisitos de sincronización $\mathcal{R}_{sinc}$; ya que, es igual a dicho proceso entrelazamiento salvo los arcos prohibidos por $\mathcal{R}_{sinc}$. El algoritmo propuesto es un método incremental que parte de un procesos sincronizador inicial $\mathcal{Y}_{ini}$ al que se le van añadiendo los arcos necesarios para eliminar, del proceso entrelazamiento, los arcos prohibidos por los requisitos de sincronización. Por tanto, se debe elegir un proceso sincronizador inicial que no elimine ningún arco. La figura 11.1 muestra el proceso sincronizador $\mathcal{Y}_{ini}$, que consiste en un único estado en el que todas las acciones son posibles, siendo su evolución un bucle hacia el mismo estado. La sincronización de dicho proceso con cualquier otro no elimina ningún comportamiento del sistema original:

$$\mathcal{S} = \mathcal{S}_{ent} \mid\mid \mathcal{Y}_{ini} = \mathcal{S}_{ent}$$

Figura 11.1: Proceso sincronizador inicial $\mathcal{Y}_{ini}$.

A continuación, se describe el proceso incremental mediante el que se van añadiendo nuevos arcos al sincronizador inicial hasta obtener el proceso sincronizador buscado. Dicho proceso se aplica sobre las acciones de sincronización, que se subespecifican en todos los estados del proceso sincronizador. La pérdida de subespecificación de dichas acciones permite eliminar o no las evoluciones a través de dichas acciones. El conjunto inicial de acciones de sincronización se corresponde con las acciones de los arcos prohibidos por los requisitos de sincronización.

Sea a_j una acción de sincronización, y sea c_{ij} un arco prohibido de $\mathcal{S}_{ent}$ a través de la acción a_j. Para eliminar dicho arco, es necesario añadir al proceso sincronizador el conjunto de caminos

[3] $\mathcal{P}_1 \mid [\Lambda] \mid \mathcal{P}_2$: $\dfrac{\mathcal{P}_1 \xrightarrow{a} \mathcal{P}_1',\ a \notin \Lambda}{\mathcal{P}_1 \mid [\Lambda] \mid \mathcal{P}_2 \xrightarrow{a} \mathcal{P}_1' \mid [\Lambda] \mid \mathcal{P}_2}$ $\dfrac{\mathcal{P}_2 \xrightarrow{a} \mid \mathcal{P}_2',\ a \notin \Lambda}{\mathcal{P}_1 \mid [\Lambda] \mid \mathcal{P}_2 \xrightarrow{a} \mathcal{P}_1 \mid [\Lambda] \mid \mathcal{P}_2'}$
$\dfrac{\mathcal{P}_1 \xrightarrow{a} \mathcal{P}_1',\ \mathcal{P}_2 \xrightarrow{a} \mathcal{P}_2',\ a \in \Lambda}{\mathcal{P}_1 \mid [\Lambda] \mid \mathcal{P}_2 \xrightarrow{a} \mathcal{P}_1' \mid [\Lambda] \mid \mathcal{P}_2'}$

al arco c_{ij} del proceso resultante $\mathcal{S}$[4]. Dichos caminos coinciden con los del proceso entrelazamiento salvo en el último estado, en el que la acción a_j está prohibida ($falsa$). Además, el resto de caminos del proceso $\mathcal{S}$ cuyo último arco es una evolución a través de la acción a_j deben añadirse también al proceso sincronizador. De lo contrario, dichos caminos serían también eliminados. Sin embargo, puede suceder que alguno de dichos caminos no pueda ser añadido al proceso sincronizador, debido a que ambos contienen arcos inconsistentes. Esto supone la aparición de una nueva acción de sincronización (una acción perteneciente al camino que no se ha podido añadir), lo que permite añadir dicho camino mediante la creación de nuevos estados en el proceso sincronizador.

Por ejemplo, sea $\Lambda = \{a, b, c, d\}$ el conjunto de acciones de un proceso $\mathcal{S}_{ent}$. El proceso sincronizador $\mathcal{Y}_1$, mostrado en la figura 11.2(a), elimina la rama de un proceso $\mathcal{S}_{ent}$ que empieza con la acción c. En la figura 11.2(b) se muestra el grafo MUS correspondiente a un proceso sincronizador $\mathcal{Y}_2$ que, además de eliminar la rama anterior, permite que la acción c sea posible después de evolucionar a través de la acción a –$\Lambda_{sinc} = \{a, c\}$–.

Además, dado que las acciones b y d no afectan a las ramas prohibidas ni a las ramas permitidas por los procesos $\mathcal{Y}_1$ e $\mathcal{Y}_2$ –se permiten todas las evoluciones a través de dichas acciones–, es posible eliminarlas del proceso sincronizador y del operador arquitectónico. De esta manera, se obtienen dos procesos sincronizadores más simples –$\mathcal{Y}_1'$ e $\mathcal{Y}_2'$– que combinados con el operador arquitectónico *sincronización parcial*, permiten sintetizar los mismos procesos resultantes (ver figuras 11.2(c) y (d)).

$$\mathcal{S}_{ent} \,||\, \mathcal{Y}_1 \;=\; \mathcal{S}_{ent} \,|[a]|\, \mathcal{Y}_1' \qquad\qquad \mathcal{S}_{ent} \,||\, \mathcal{Y}_2 \;=\; \mathcal{S}_{ent} \,|[a,c]|\, \mathcal{Y}_2'$$

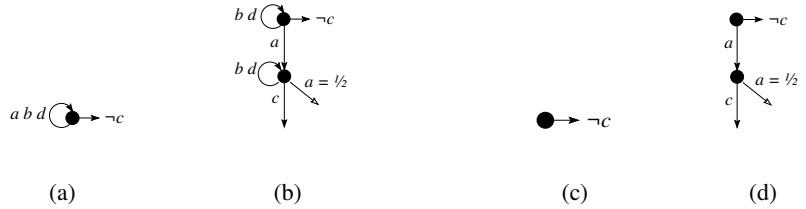


Figura 11.2: Procesos sincronizadores $\mathcal{Y}_1$, $\mathcal{Y}_2$, $\mathcal{Y}_1'$, $\mathcal{Y}_2'$.

11.2.1 Pseudocódigo

El algoritmo 11.1 muestra el pseudocódigo del algoritmo de sincronización. Dicho algoritmo recibe como parámetros los procesos componentes $\mathcal{P}_1$ y $\mathcal{P}_2$, y el conjunto de requisitos de sincronización $\mathcal{R}_{sinc}$. A continuación, se describen los pasos más relevantes del algoritmo 11.1:

[**1**] Se obtiene el proceso resultante y el conjunto inicial de acciones de sincronización.

[**3**] Las acciones de sincronización se subespecifican en el proceso sincronizador inicial.

[4] $p_{c_{ij}}$: conjunto de caminos compuestos por los arcos necesarios para evolucionar desde el estado inicial al arco c_{ij}.

Algoritmo 11.1 Algoritmo de Sincronización I ($\mathcal{P}_1, \mathcal{P}_2, \{\mathcal{R}_{sinc}\}$)

[1] Obtener $\mathcal{S}$ y Λ_{sinc} a partir de $\mathcal{S}_{int} = \mathcal{P}_1 \,|||\, \mathcal{P}_2$ y $\{\mathcal{R}_{sinc}\}$;
[2] $\mathcal{Y} = \mathcal{Y}_{ini}$;
[3] Para toda $a_i \in \Lambda_{sinc}$: $E_0[a_i] = \frac{1}{2}$;
[4] Repetir, para toda $a_i \in \Lambda_{sinc}$:
  [a] Repetir, para todo $p \in \{p^{\mathcal{S}}_{a_i}\}$:
    [i] Si $p \notin \mathcal{Y}$:
      [A] Si es posible: $\mathcal{Y} = \mathcal{Y} \cup p$;
      [B] En caso contrario:
        [I] Elegir $a_k \notin \Lambda_{sinc}$ / $a_k \in p$;
        [II] $\Lambda_{sinc} = \Lambda_{sinc} \cup a_k$;
        [III] Volver al paso [2];
  [b] Fin Repetir;
[5] Fin Repetir;
[6] Para toda $a_i \in \Lambda_{sinc}$ y para todo $E_j \in \mathcal{Y}$ / $E_j[a_i] = \frac{1}{2}$:
  [a] $E_j[a_i] = 0$;
[7] Para toda $a_i \notin \Lambda_{sinc}$ y para todo $E_j \in \mathcal{Y}$:
  [a] $E_j[a_i] = 0$;
[8] Devolver $\mathcal{Y}$ y Λ_{sinc};

Notación:
$\{p^{\mathcal{S}}_{a_i}\}$: Conjunto de caminos (ordenados de menor a mayor según el número de arcos) de $\mathcal{S}$ cuyo último arco –incluyendo los arcos prohibidos– evoluciona a través de la acción a_i.

[**4**] Todos los caminos de $\mathcal{S}$ cuyo último arco contiene alguna acción de sincronización (incluyendo los arcos prohibidos), se añaden al proceso sincronizador. Esta operación puede causar que alguna acción se añada al conjunto de acciones de sincronización (ver paso [4][a][i][B][II]). En este caso, el algoritmo vuelve al paso [2].

[**6**] El proceso sincronizador se simplifica especificando como *falsas* las acciones subespecificadas.

[**7**] El proceso sincronizador se simplifica eliminando las acciones que no son de sincronización.

El algoritmo 11.1 presenta un claro defecto debido al alto coste computacional que supone el cálculo de todos los caminos en los que están involucradas las acciones de sincronización. Este problema se acentúa en sistemas con bucles, en los que el número de caminos aumenta exponencialmente. Sin embargo, es posible resolver este problema eliminando los bucles del sistema resultante $\mathcal{S}$, ya que estos pueden incorporarse una vez obtenido el proceso sincronizador. Esto implica que las acciones involucradas en dichos bucles pasen a formar parte del conjunto de acciones de sincronización. Es necesario, por tanto, mantener un mapa de estados que relacione

cada estado del proceso resultante $\mathcal{S}$ con un estado del proceso sincronizador; lo que permitirá la incorporación de los bucles eliminados de $\mathcal{S}$. Inicialmente, todos los estados de $\mathcal{S}$ se corresponden con el único estado del sincronizador inicial $\mathcal{Y}_{ini}$. El algoritmo 11.2 incluye estas consideraciones (ver pasos [2], [5], [6][ii] y [8]).

Algoritmo 11.2 Algoritmo de Sincronización II $(\mathcal{P}_1, \mathcal{P}_2, \{\mathcal{R}_{sinc}\})$

[1] Obtener $\mathcal{S}$ y Λ_{sinc} a partir de $\mathcal{S}_{int} = \mathcal{P}_1 \,|||\, \mathcal{P}_2$ y $\{\mathcal{R}_{sinc}\}$;
[2] $\mathcal{S}' = \mathcal{S}$; Eliminar los bucles de $\mathcal{S}$;
[3] $\mathcal{Y} = \mathcal{Y}_{ini}$;
[4] Para toda $a_i \in \Lambda_{sinc}$: $E_0[a_i] = \frac{1}{2}$;
[5] Desde $i = 0$ hasta $N_{\mathcal{S}'}$: $map[i] = E_0$;
[6] Repetir para toda $a_i \in \Lambda_{sinc}$:
  [a] Repetir para todo $p \in \{p_{a_i}^{\mathcal{S}'}\}$:
    [i] Si $p \notin \mathcal{Y}$:
      [A] Si es posible: $\mathcal{Y} = \mathcal{Y} \cup p$;
      [B] En caso contrario:
        [I] Elegir $a_k \notin \Lambda_{sinc}$ / $a_k \in p$;
        [II] $\Lambda_{sinc} = \Lambda_{sinc} \cup a_k$;
        [III] Volver al paso [2];
    [ii] Actualizar map;
  [b] Fin Repetir;
[7] Fin Repetir;
[8] Añadir a $\mathcal{Y}$ los bucles de $\mathcal{S}$; Actualizar Λ_{sinc};
[9] Para toda $a_i \in \Lambda_{sinc}$ y para todo $E_j \in \mathcal{Y}$ / $E_j[a_i] = \frac{1}{2}$:
  [a] $E_j[a_i] = 0$;
[10] Para toda $a_i \notin \Lambda_{sinc}$ y para todo $E_j \in \mathcal{Y}$:
  [a] $E_j[a_i] = 0$;
[11] Devolver $\mathcal{Y}$ y Λ_{sinc};

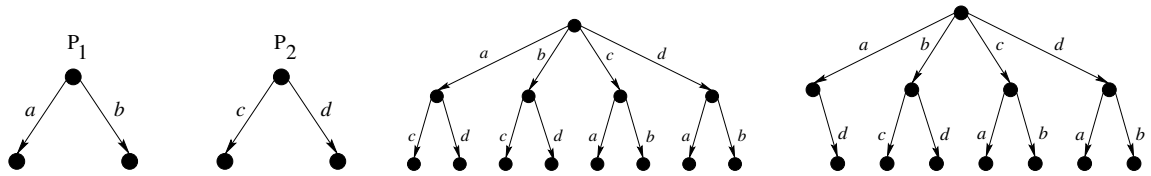
Notación:
$N_{\mathcal{S}'}$: Número de estados de $\mathcal{S}'$.
$\{p_{a_i}^{\mathcal{S}'}\}$: Conjunto de caminos (ordenados de menor a mayor según el número de arcos) de $\mathcal{S}'$ cuyo último arco –incluyendo los arcos prohibidos– evoluciona a través de la acción a_i.
map: Mapa de estados que relaciona los estados del proceso resultante $\mathcal{S}$ con los estados del proceso sincronizador $\mathcal{Y}$.

11.2.2 Ejemplos

En la figura 11.3(a) se muestran los grafos MUS de dos procesos $\mathcal{P}_1$ y $\mathcal{P}_2$. La figura 11.3(b) muestra el grafo MUS correspondiente al proceso entrelazamiento de ambos, $\mathcal{S}_{ent}$. En la figu-

ra 11.3(c) se muestra el grafo MUS correspondiente al proceso $\mathcal{S}_1$, resultado de eliminar, en el proceso $\mathcal{S}_{ent}$, las ramas que no cumplen el requisito de sincronización $\mathcal{R}_{sin_1}$, que se muestra a continuación.

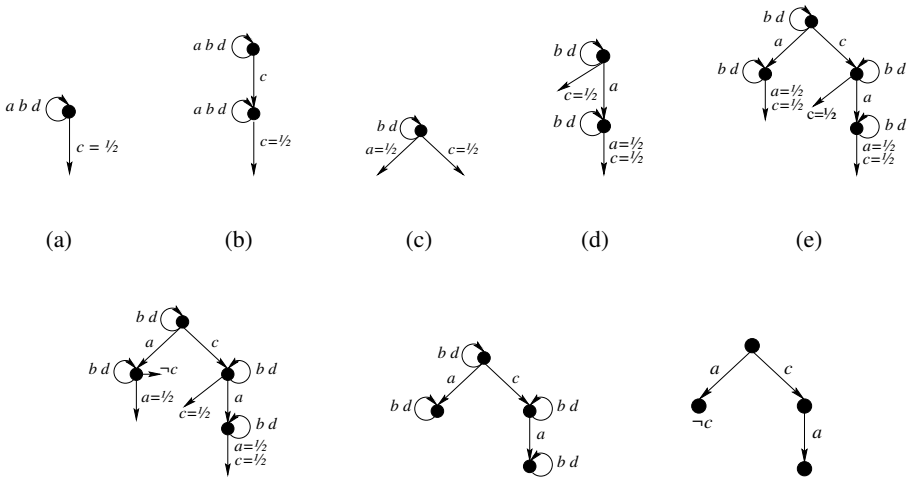
req $\mathcal{R}_{sin_1}$ **is**
$(true \Rightarrow a) \Rightarrow \bigcirc(true \Rightarrow \neg c)$
endreq



(a) $\mathcal{P}_1$ y $\mathcal{P}_2$ (b) $\mathcal{S}_{ent}$ (c) $\mathcal{S}_1$

Figura 11.3: Grafos MUS de los procesos $\mathcal{P}_1$, $\mathcal{P}_2$, $\mathcal{S}_{ent}$ y $\mathcal{S}_1$.

Para obtener el proceso sincronizador se aplica el algoritmo 11.2 descrito en la sección anterior. En la figura 11.4 se muestra, paso por paso, la obtención de dicho proceso sincronizador $\mathcal{Y}_1'$, lo que permite expresar de manera estructurada el sistema especificado. $\mathcal{S}_1 = (\mathcal{P}_1 \,|||\, \mathcal{P}_2) \,|[a,c]|\, \mathcal{Y}_1'$. El ejemplo 11.1 resume los principales pasos de la síntesis del proceso sincronizador, incluyendo el número del paso en el que se encuentra el algoritmo 11.2.



(a) (b) (c) (d) (e)

(f) (g) $\mathcal{Y}_1$ (h) $\mathcal{Y}_1'$

Figura 11.4: Síntesis del Proceso Sincronizador $\mathcal{Y}$.

Ejemplo 11.1 Ejemplo del Algoritmo de Sincronización II.

[1] Obtener $\mathcal{S}_1$ (ver figura 11.3(c)) y $\Lambda_{sinc_1} = \{c\}$;
[4] Inicializar $\mathcal{Y}_1$ (ver figura 11.4(a));
[6][a] Obtener $\{p_c^{\mathcal{S}_1}\}$ (ver figuras 11.5(a)(b)(c));
[6][a][i][A] $\mathcal{Y}_1 \cup p_{c_1}^{\mathcal{S}_1}$ (ver figura 11.4(b));
[6][a][i][B] $p_{c_2}^{\mathcal{S}_1}$ es inconsistente con $\mathcal{Y}_1$;
[6][a][i][B][ii] $\Lambda_{sinc_1} = \{a, c\}$;
[4] Inicializar $\mathcal{Y}_1$ (ver figura 11.4(c));
[6][a] Obtener $\{p_a^{\mathcal{S}_1}\}$ (ver figuras 11.5(d)(e)(f));
[6][a][i][A] $\mathcal{Y}_1 \cup p_{a_1}^{\mathcal{S}_1}$ (ver figura 11.4(d));
[6][a][i][A] $\mathcal{Y}_1 \cup p_{a_2}^{\mathcal{S}_1}$ (ver figura 11.4(e));
[6][a][i] $p_{a_3}^{\mathcal{S}_1}$ existe en $\mathcal{Y}_1$;
[6][a][i] $p_{c_1}^{\mathcal{S}_1}$ existe en $\mathcal{Y}_1$;
[6][a][i][A] $\mathcal{Y}_1 \cup p_{c_2}^{\mathcal{S}_1}$ (ver figura 11.4(f));
[6][a][i] $p_{c_3}^{\mathcal{S}_1}$ existe en $\mathcal{Y}_1$;
[9] Eliminar subespecificación (ver figura 11.4(g));
[10] Eliminar $a_i \notin \Lambda_{sinc_1}$ (ver figura 11.4(h));
[11] Devolver $\mathcal{Y}_1'$ y $\Lambda_{sinc_1} = \{a, c\}$;

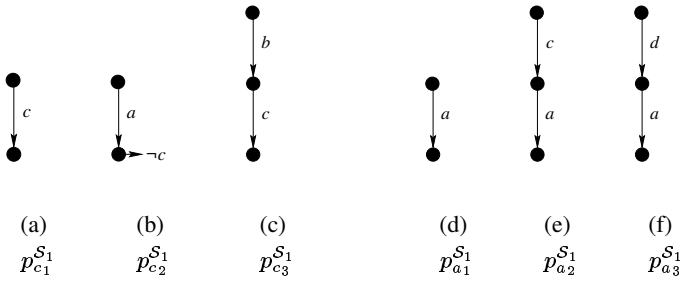


Figura 11.5: $\{p_c^{\mathcal{S}_1}\}$ y $\{p_a^{\mathcal{S}_1}\}$.

A continuación, se muestra la síntesis de un nuevo proceso sincronizador $\mathcal{Y}_2$, que representa las restricciones impuestas por el requisito $\mathcal{R}_{sin_2}$. La figura 11.6 muestra el proceso resultante $\mathcal{S}_2$, después de eliminar las ramas prohibidas del proceso $\mathcal{S}_{ent}$.

req $\mathcal{R}_{sin_2}$ **is**
$(true \Rightarrow a) \Rightarrow (true \Rightarrow \neg c)$
endreq

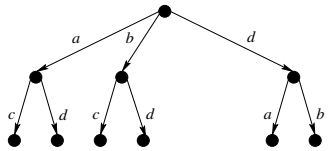


Figura 11.6: Grafo MUS del proceso resultante $\mathcal{S}_2$.

En la figura 11.7 se muestra cómo se obtiene un proceso sincronizador $\mathcal{Y}_2'$, tal que:

$$\mathcal{S}_2 = (\mathcal{P}_1 \mid\mid\mid \mathcal{P}_2) \mid[a, b, c]\mid \mathcal{Y}_2'.$$

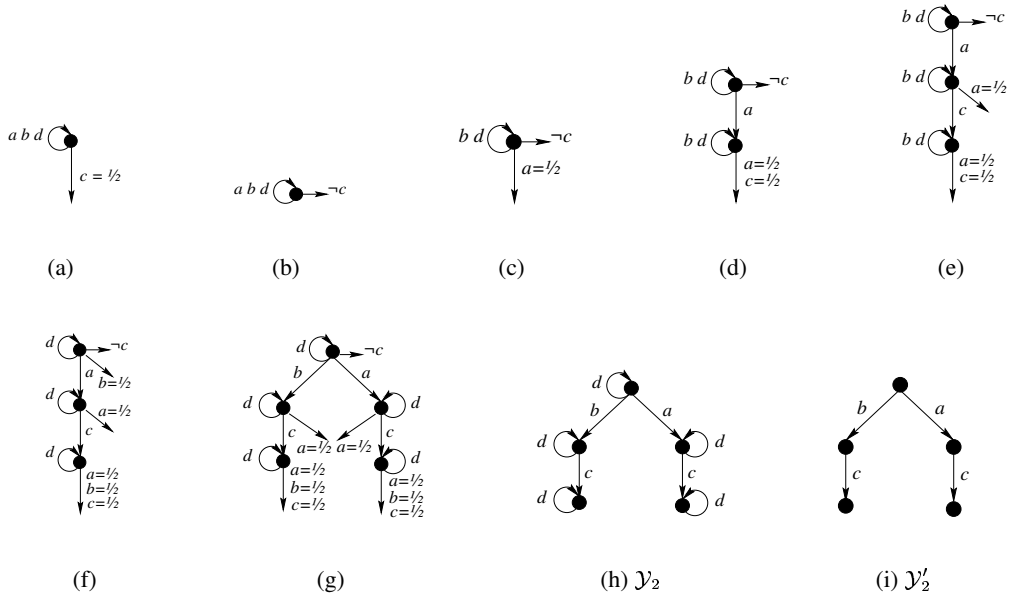


Figura 11.7: Síntesis del proceso sincronizador $\mathcal{Y}_2$.

Finalmente, en la figura 11.8 se muestra el grafo MUS $\mathcal{S}_3$, obtenido al aplicar al proceso $\mathcal{S}_{ent}$ el requisito de sincronización $\mathcal{R}_3$. La figura 11.9 muestra el grafo MUS del proceso sincronizador $\mathcal{Y}_3$: $\mathcal{S}_3 = (\mathcal{P}_1 \mid\mid\mid \mathcal{P}_2) \mid[a, b, c]\mid \mathcal{Y}_3'$.

req $\mathcal{R}_{sin_3}$ **is**
$(true \Rightarrow d) \wedge (true \Rightarrow \bigodot a) \Rightarrow (true \Rightarrow \neg c)$
endreq

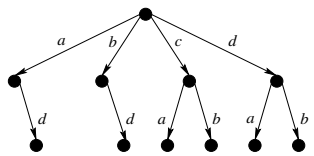
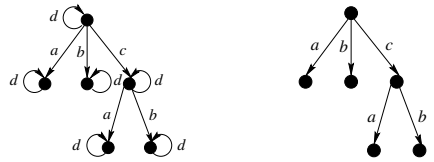


Figura 11.8: Grafo MUS del proceso resultante $\mathcal{S}_3$.



(a) $\mathcal{Y}_3$ (b) $\mathcal{Y}'_3$

Figura 11.9: Síntesis del proceso sincronizador $\mathcal{Y}_3$.

11.3 Procesos Sincronizadores Mínimos

11.3.1 Introducción

La síntesis de un proceso sincronizador $\mathcal{Y}'$ permite expresar el sistema final de forma estructurada, en función de sus procesos componentes: $\mathcal{S} = \mathcal{S}_{ent} \; |[\Lambda_{sin}]| \; \mathcal{Y}'$. La sincronización parcial de dos procesos obtiene, como resultado, un proceso que consiste en el entrelazamiento de dichos procesos salvo las ramas de ambos que no se sincronizan. Sin embargo, si uno de los procesos sólo contiene acciones de sincronización, es posible asegurar que el resultado es un subconjunto del otro proceso. Es decir, si $\forall a_i \in \mathcal{Y}', a_i \in \Lambda_{sin}$, entonces: $\mathcal{S} \subseteq \mathcal{S}_{ent}$.

Los procesos sincronizadores obtenidos por el algoritmo de sincronización 11.1 cumplen la condición anterior, ya que las acciones de sincronización se obtienen, precisamente, de las acciones involucradas en los procesos sincronizadores sintetizados. Esta conclusión es la esperada, ya que los procesos sincronizadores sólo deben limitar el comportamiento concurrente de los procesos componentes, sin añadir comportamiento al sistema.

Por tanto, los procesos sincronizadores obtenidos en las secciones anteriores sólo eliminan las ramas prohibidas del proceso entrelazamiento, tal y como se había previsto. No obstante, dicha condición la cumplen todos los procesos sincronizadores cuyas acciones están incluidas en el conjunto Λ_{sin}. Los procesos sincronizadores obtenidos por el algoritmo 11.2 pueden reducirse, obteniendo procesos más simples (con menor número de estados y arcos). La figura 11.10 muestra los procesos sincronizadores mínimos correspondientes a los procesos sincronizadores de las figuras 11.4(h) y 11.9(b).

(a) $\mathcal{Y}'_{1_{min}}$ (b) $\mathcal{Y}'_{3_{min}}$

Figura 11.10: Procesos sincronizadores mínimos.

El objetivo planteado es, por tanto, la síntesis del proceso sincronizador **mínimo** $\mathcal{Y}_{min}$, siendo dicho proceso, el proceso sincronizador más simple –con menor número de estados y arcos– que

permite obtener de manera estructurada el proceso resultante, $\mathcal{S} = \mathcal{S}_{ent} \, ||[\Lambda_{sin}]|| \, \mathcal{Y}_{min}$.

11.3.2 Síntesis del Sincronizador Mínimo

Las conclusiones obtenidas en la sección anterior sugieren buscar una nueva estrategia para la síntesis de procesos sincronizadores mínimos. Dicho cambio de estrategia se basa en cambiar el punto de partida. En vez de partir de un proceso sincronizador $\mathcal{Y}_{ini}$ sobre el que se van añadiendo las ramas necesarias, se plantea la posibilidad de partir de un proceso $\mathcal{Y}_{max}$ e ir eliminando en éste las ramas que no son necesarias. Esta nueva estrategia tiene, como segundo objetivo, reducir el coste computacional del algoritmo 11.2 debido al cálculo de caminos en los que intervienen las acciones de sincronización.

La elección de $\mathcal{Y}_{max}$ es muy sencilla: $\mathcal{Y}_{max} = \mathcal{S}$, ya que, $\mathcal{S}$ es la "cota superior" del proceso sincronizador mínimo. El problema surge en cómo eliminar las ramas de $\mathcal{Y}_{max}$ para reducirlo hasta sintetizar el proceso sincronizador más simple. La solución propuesta se basa en la evolución del proceso $\mathcal{S}_{ent}$ y el reflejo de esta evolución en cada uno de los procesos componentes $\mathcal{P}_1$ y $\mathcal{P}_2$. Si la evolución del proceso entrelazamiento se corresponde con una evolución de $\mathcal{P}_1$ ($\mathcal{P}_2$ no evoluciona), el resultado sería el mismo que si se añade a $\mathcal{P}_2$ un bucle mediante la acción de evolución, y se sincronizaran ambos procesos.

La síntesis de un proceso sincronizador $\mathcal{Y}_{min}$ a partir de $\mathcal{Y}_{max}$ consiste en simplificar dicho proceso aplicando en éste el algoritmo de reducción de estados explicado en el capítulo 10. Tal y como se explicó en el capítulo 10, el algoritmo de reducción de estados obtiene un conjunto de grafos MUS a partir de uno dado. En este caso, el algoritmo finaliza cuando el grafo no puede reducirse más, obteniendo un proceso $\mathcal{Y}_{min}$ que satisface: $\mathcal{S} = \mathcal{S}_{ent}||\mathcal{Y}_{min}$. Además, dicho proceso puede simplificarse eliminando las acciones que no son de sincronización. Dichas acciones son aquellas que en todos los estados están subespecificadas, o bien forman un bucle al mismo estado: $\forall a_i \in \Lambda$, $a_i \notin \Lambda_{sync}$ sii $\forall E_j \in \mathcal{Y}, E_j \xrightarrow{a_i} E_j$ or $E_j[a_i] = \frac{1}{2}$.

11.3.3 Pseudocódigo

El algoritmo 11.3 muestra el pseudocódigo del algoritmo descrito. Dicho algoritmo recibe como parámetros los procesos componentes $\mathcal{P}_1$ y $\mathcal{P}_2$, y el conjunto de requisitos de sincronización $\mathcal{R}_{sinc}$. La aplicación de dicho algoritmo sin reducir al máximo el proceso sincronizador, permite obtener familias de procesos sincronizadores, lo que proporciona una relación entre los diferentes grafos MUS de un requisito SCTL.

A continuación, se describen los principales pasos del algoritmo 11.3:

[**1**] Obtener el proceso resultante $\mathcal{S}$.

[**2**] Inicializar el proceso sincronizador al proceso resultante $\mathcal{S}$.

[**3**] Cada arco de $\mathcal{Y}$ se reduce uniendo (si es posible) los estados que forman dicho arco.

Algoritmo 11.3 Algoritmo de Sincronización III $(\mathcal{P}_1, \mathcal{P}_2, \{\mathcal{R}_{sinc}\})$

[1] Obtener $\mathcal{S}$ a partir de $\mathcal{S}_{int} = \mathcal{P}_1 \,|||\, \mathcal{P}_2$ y $\{\mathcal{R}_{sinc}\}$;
[2] $\mathcal{Y} = \mathcal{Y}_{max} = \mathcal{S}$;
[3] Repetir para toda $a_i \in \Lambda$:
  [a] Para todo $E_j, E_k \in \mathcal{Y}$ / $E_j \xrightarrow{a_i} E_k$:
    [i] Unir Estados $(\boldsymbol{\mathcal{Y}}, \boldsymbol{E_j}, \boldsymbol{E_k})$;
[4] $\Lambda_{sync} = \Lambda$;
[5] Para toda $a_i \in \Lambda_{sinc}$ / $\forall E_j \in \mathcal{Y}, E_j \xrightarrow{a_i} E_j$ or $E_j[a_i] = \frac{1}{2}$:
  [a] $\Lambda_{sinc} = \Lambda_{sinc} - a_i$;
[6] Devolver $\mathcal{Y}$ y Λ_{sinc};

Algoritmo 11.4 Unir Estados $(\mathcal{Y}, E_1, E_2)$

[1] If $E_1 = E_2$ devolver($\mathcal{Y}$, ok);
[2] $\mathcal{Y}_{old} = \mathcal{Y}$;
[3] Repetir para toda $a_i \in \Lambda$:
  [a] Si $E_1[a_i] = 0$:
    [i] Si $E_2[a_i] = 1$ devolver($\mathcal{Y}_{old}$, $error$);
  [b] Si $E_1[a_i] = \frac{1}{2}$:
    [i] Si $E_1[a_i] \neq \frac{1}{2}$:
      [A] $E_1[a_i] = E_2[a_i]$;
      [B] Si $\exists E_j$ / $E_2 \xrightarrow{a_i} E_j$: $E_1 \xrightarrow{a_i} E_j$;
  [c] Si $E_1[a_i] = 1$ / $E_1 \xrightarrow{a_i} E_j$:
    [i] Si $E_2[a_i] = 0$ devolver($\mathcal{Y}_{old}$, $error$);
    [ii] Si $E_2[a_i] = 1$ / $E_2 \xrightarrow{a_i} E_k$:
      [A] $error$ = **Unir Estados $(\boldsymbol{\mathcal{Y}}, \boldsymbol{E_j}, \boldsymbol{E_k})$**;
      [B] Si $error$ devolver($\mathcal{Y}_{old}$, $error$);
[4] Fin Repetir;
[5] Para toda $a_i \in \Lambda$ y para todo $E_j \in \mathcal{Y}$ / $E_j \xrightarrow{a_i} E_2$:
  [a] $E_j \xrightarrow{a_i} E_1$;
[6] Eliminar E_2 de $\mathcal{Y}$;
[7] Devolver($\mathcal{Y}$, ok);

[**5**] Reducir el proceso sincronizador eliminando las acciones que no son de sincronización.

El algoritmo 11.4 muestra cómo se unen dos estados del grafo del proceso sincronizador[5]. Si los estados son inconsistentes, el algoritmo devuelve un código de error y no modifica el grafo del proceso sincronizador(ver los pasos [2], [3][a][i], [3][c][i] y [3][c][ii][B]).

11.3.4 Ejemplo

En esta sección se sintetiza el proceso sincronizador mínimo correspondiente al ejemplo 11.1. Para ello, se utiliza el algoritmo 11.3 descrito en la sección anterior. En la figura 11.11(a) se muestra el proceso sincronizador inicial $\mathcal{Y}_{1_{max}} = \mathcal{S}_1$. El ejemplo 11.2 describe los principales pasos seguidos por el algoritmo 11.3 para obtener el proceso sincronizador mínimo $\mathcal{Y}_{1_{min}}$, que se muestra en la figura 11.11 (b).

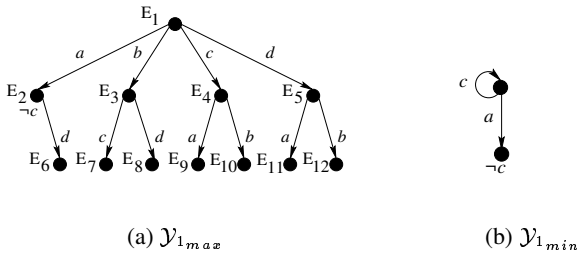


Figura 11.11: Procesos sincronizadores.

Ejemplo 11.2 Ejemplo del Algoritmo de Sincronización III.

[1] E_1 y E_2 no se pueden unir, ya que $E_2[c] = 0$ y $E_1[c] = 1$;
[2] Se unen E_1 y E_3. Esto implica que se unan los estados E_4 y E_7, y los estados E_5 y E_8 (ver la figura 11.12(a));
[3] Se unen $E_{4\cup 7}$ y $E_{1\cup 3}$. Esto implica que se unan los estados E_2 y E_9, y los estados $E_{1\cup 3}$ y E_{10} (ver la figura 11.12(b));
[4] Se unen $E_{1\cup 3\cup 4\cup 7\cup 10}$ y $E_{5\cup 8}$. Esto implica que se unan los estados $E_{2\cup 9}$ y E_{11}, y los estados $E_{1\cup 3\cup 4\cup 7\cup 10}$ y E_{12} (ver figure 11.12(c));
[5] Se unen $E_{2\cup 9\cup 11}$ y E_6 (ver figura 11.12(d));
[5] Se eliminan las acciones que no son de sincronización. $\Lambda_{sinc} = \{a, c\}$. $\mathcal{Y}_{1_{min}}$ se muestra en la figura 11.11(b);

[5]Una operación unión entre dos estados no es posible (ver paso [2][b]) si dichos estados no son compatibles, o si implica la unión de otros estados no compatibles, tal y como se explicó en el algoritmo de reducción de estados.

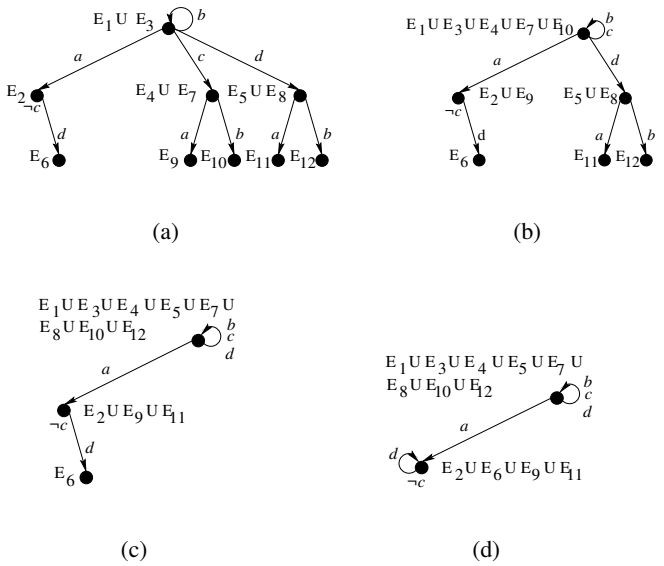


Figura 11.12: Síntesis del proceso sincronizador $\mathcal{Y}_{1_{min}}$.

11.4 Reutilización de Procesos Sincronizadores

En las secciones anteriores se han sintetizado diversos procesos de sincronización, en función de la especificación de diferentes requisitos de sincronización. Sin embargo, se ha mantenido el conjunto de procesos $\mathcal{P}_1$ y $\mathcal{P}_2$ sobre los que se especificaban las restricciones de evolución concurrente.

Se plantea obtener un proceso sincronizador que satisfaga las restricciones impuestas por dos conjuntos de requisitos $\{\mathcal{R}_{sin_1}\}$ y $\{\mathcal{R}_{sin_2}\}$. La primera solución es aplicar el algoritmo de sincronización con la unión de dichos requisitos, obteniendo un proceso sincronizador global $\mathcal{Y}_{glob}$. Sin embargo, si ya han sido sintetizado los procesos sincronizadores $\mathcal{Y}_1$ e $\mathcal{Y}_2$, sería muy interesante poder obtener el sincronizador global a partir de los anteriores:

- Por una parte, esto permitiría **reutilizar** la síntesis de dichos procesos, ya que, el proceso sincronizador debe contener la sincronización expresada por ambos procesos $\mathcal{Y}_1$ e $\mathcal{Y}_2$.
- Por otra, la síntesis del proceso sincronizador a partir de los sincronizadores componentes, sugiere la posibilidad de expresar el primero de manera estructurada en función de los mismos; exactamente igual que el proceso del sistema diseñado, con las consiguientes ventajas ya expresadas en la sección 11.1.

Sean $\mathcal{S}_1$ y $\mathcal{S}_2$ los procesos resultantes después de eliminar en el proceso entrelazamiento $\mathcal{S}_{ent} = \mathcal{P}_1 \,|||\, \mathcal{P}_2$, las restricciones de los conjuntos de requisitos $\mathcal{R}_{sin_1}$ y $\mathcal{R}_{sin_2}$, respectivamente; y sean $\mathcal{Y}_1$ e $\mathcal{Y}_2$ los procesos sincronizadores obtenidos por el algoritmo de sincronización, tales que:

$$\mathcal{S}_1 = (\mathcal{P}_1 \,|||\, \mathcal{P}_2) \,||\, \mathcal{Y}_1 = (\mathcal{P}_1 \,|||\, \mathcal{P}_2) \,|[\Lambda_{sin_1}]|\, \mathcal{Y}_1'$$

$$\mathcal{S}_2 = (\mathcal{P}_1 \,|||\, \mathcal{P}_2) \,||\, \mathcal{Y}_2 = (\mathcal{P}_1 \,|||\, \mathcal{P}_2) \,|[\Lambda_{sin_2}]|\, \mathcal{Y}'_2$$

Sea $\mathcal{S}_3$ el proceso resultante de eliminar en el proceso entrelazamiento las restricciones de los dos conjuntos de requisitos $\mathcal{R}_{sin_1}$ y $\mathcal{R}_{sin_2}$, y sea $\mathcal{Y}_3$ el proceso sincronizador obtenido por el algoritmo de sincronización:

$$\mathcal{S}_3 = (\mathcal{P}_1 \,|||\, \mathcal{P}_2) \,||\, \mathcal{Y}_3 = (\mathcal{P}_1 \,|||\, \mathcal{P}_2) \,|[\Lambda_{sin_3}]|\, \mathcal{Y}'_3$$

La solución propuesta se basa en la utilización de los procesos sincronizadores obtenidos antes de eliminar las acciones que no son de subespecificación. Esto permite obtener un proceso de sincronización $\mathcal{Y}_{glob}$ con las características deseadas. Para calcular dicho proceso de sincronización, basta con aplicar la siguiente fórmula:

$$\mathcal{Y}_{glob} = (\mathcal{Y}_1 \,||\, \mathcal{Y}_2)$$

A continuación, se demuestra que dicho proceso satisface que: $\mathcal{S}_3 = (\mathcal{P}_1 \,|||\, \mathcal{P}_2) \,||\, \mathcal{Y}_{glob}$, obteniendo así el proceso sincronizador buscado.

$\mathcal{S}_3 = \mathcal{S}_1 \,||\, \mathcal{S}_2$, ya que es el resultado de eliminar, en el proceso entrelazamiento $\mathcal{S}_{ent}$, las ramas eliminadas en la obtención de $\mathcal{S}_1$ y las correspondientes a $\mathcal{S}_2$. Por tanto, y sin más que sustituir en la expresión anterior, se obtiene:

$$\mathcal{S}_3 = (\mathcal{S}_{ent} \,||\, \mathcal{Y}_1) \,||\, (\mathcal{S}_{ent} \,||\, \mathcal{Y}_2)$$

Teniendo en cuenta que $\mathcal{S}_{ent} \,||\, \mathcal{S}_{ent} = \mathcal{S}_{ent}$, y aplicando las propiedades asociativa y conmutativa:

$$\mathcal{S}_3 = \mathcal{S}_{ent} \,||\, (\mathcal{Y}_1 \,||\, \mathcal{Y}_2) = \mathcal{S}_{ent} \,||\, \mathcal{Y}_{glob}$$

Además, es posible obtener un nuevo proceso $\mathcal{Y}'_{glob}$, y un conjunto de acciones de sincronización $\Lambda_{sin_{glob}}$, tales que:

$$\mathcal{S}_3 = \mathcal{S}_{ent} \,|[\Lambda_{sin_{glob}}]|\, \mathcal{Y}'_{glob}$$

Para llegar a esta conclusión, basta con fijarse en la naturaleza de los procesos $\mathcal{Y}_1$ e $\mathcal{Y}_2$ que forman el proceso de sincronización global. Cada uno de ellos, mantiene un conjunto de acciones $\Lambda - \Lambda_{sin_i}$ que forman bucles en todos los estados del mismo. Es posible, por tanto, eliminar de la sincronización aquellas acciones que en ambos procesos no pertenecen al conjunto de acciones de sincronización. Además, dichas acciones serán las que cumplen las características anteriores en el proceso $\mathcal{Y}_{glob}$ –son bucles en todos sus estados o están subespecificadas–. Por tanto, dichas acciones nunca sincronizan, por lo que es posible eliminarlas, obteniendo un nuevo proceso sincronizador $\mathcal{Y}'_{glob}$ que contiene únicamente acciones de sincronización, siendo dicho conjunto la unión de las acciones de sincronización de sus procesos componentes: $\Lambda_{sin_{glob}} = \Lambda_{sin_1} \bigcup \Lambda_{sin_2}$.

Por otra parte, utilizando el algoritmo de sincronización 11.3, es posible obtener el proceso sincronizador global desdoblando cada uno de los sincronizadores parciales hasta obtener en los dos la misma reducción de estados. Esto es equivalente a eliminar del proceso $\mathcal{S}_{ent}$ las ramas prohibidas por los dos sincronizadores parciales y aplicar la reducción de estados del algoritmo 11.3.

Este resultado permite, además, poder razonar sobre los distintos procesos sincronizadores,

ya que si el proceso sincronizador global coincidiera con alguno de sus procesos componentes, esto indicaría que dicho proceso no aporta ninguna restricción de sincronización adicional. Esto permite afirmar que los requisitos de sincronización correspondientes a dicho proceso, están **"contenidos"** en los requisitos de sincronización del primero, obteniendo así, una relación de orden entre dichos requisitos.

11.4.1 Ejemplo de Aplicación

La figura 11.13 muestra los grafos de los procesos sincronizadores correspondientes a los requisitos $\mathcal{R}_{sin_1} \wedge \mathcal{R}_{sin_2}$ y $\mathcal{R}_{sin_1} \wedge \mathcal{R}_{sin_2} \wedge \mathcal{R}_{sin_3}$, especificados en la sección 11.2.2. Dichos procesos se obtienen mediante la reutilización de los procesos $\mathcal{Y}_1$, $\mathcal{Y}_2$ e $\mathcal{Y}_3$ obtenidos en la sección 11.2.2. Además, se observa que $\mathcal{Y}_3 \mid\mid \mathcal{Y}_1 = \mathcal{Y}_3$, por lo que las restricciones impuestas por el requisito $\mathcal{R}_{sin_1}$ están incluidas en $\mathcal{R}_{sin_3}$, tal y como se deduce a partir de los grafos de los sistemas resultantes, ya que $\mathcal{S}_3 \subset \mathcal{S}_1$.

(a) $\mathcal{Y}_4$ (b) $\mathcal{Y}'_4$ (c) $\mathcal{Y}_5$ (d) $\mathcal{Y}'_5$

Figura 11.13: $\mathcal{Y}_4 = \mathcal{Y}_1 \mid\mid \mathcal{Y}_2, \;\; \mathcal{Y}_5 = \mathcal{Y}_4 \mid\mid \mathcal{Y}_3$.

Parte V

Mantenimiento

Capítulo 12

Mantenimiento de Especificaciones SCTL-MUS

La última fase del ciclo de desarrollo propuesto en el capítulo 3 es la fase de mantenimiento. El objetivo de dicha fase es mantener la operatividad del sistema desarrollado, subsanando errores y/o añadiendo nuevas funcionalidades al mismo. Para ello, es necesario almacenar información relativa al proceso de desarrollo, desde la fase de especificación hasta la fase de diseño.

El primer paso a realizar es, por tanto, la identificación del conjunto de entidades involucradas en una especificación SCTL-MUS, razonando sobre cuáles de éstas deben ser almacenadas y cuáles no. En segundo lugar, se deben idear los mecanismos de almacenamiento adecuados que permitan cumplir los objetivos de la fase de mantenimiento de una manera eficiente y estructurada.

En este capítulo se abordan dichos pasos desde varias perspectivas. Primero, desde un punto de vista global, en el que los sistemas diseñados se componen de un conjunto de unidades denominadas procesos. Finalmente, se aborda el mantenimiento a bajo nivel, definiendo estructuras que permitan almacenar de una manera eficiente tanto las fórmulas de los requisitos SCTL como sus grafos MUS.

12.1 Procesos

La metodología SCTL-MUS propuesta permite el diseño y desarrollo de sistemas distribuidos desde la fase inicial de especificación de requisitos. Un sistema distribuido $\mathcal{S}$ consta, en general, de un conjunto de procesos componentes $\mathcal{P}_1$, ..., $\mathcal{P}_p$, que se sincronizan entre sí. Dicha sincronización puede expresarse de varias maneras. La metodología SCTL-MUS propone la síntesis de un proceso sincronizador $\mathcal{Y}$ y la utilización de los operadores arquitectónicos: *entrelazamiento, sincronización parcial* y *sincronización total*.

Por tanto, un sistema $\mathcal{S}$ consta de un conjunto de procesos unidos por el operador *entrelazamiento* –$\mathcal{S}_{ent} = \mathcal{P}_1 \,|||\, \mathcal{P}_2 \,|||\, ... \,|||\, \mathcal{P}_p$–, que a su vez se sincronizan con un proceso sincronizador:

$\mathcal{S} = \mathcal{S}_{ent} \,|[\Lambda_{sin}]|\, \mathcal{Y}$. La figura 12.1 muestra la estructura de datos genérica para el mantenimiento de un sistema distribuido.

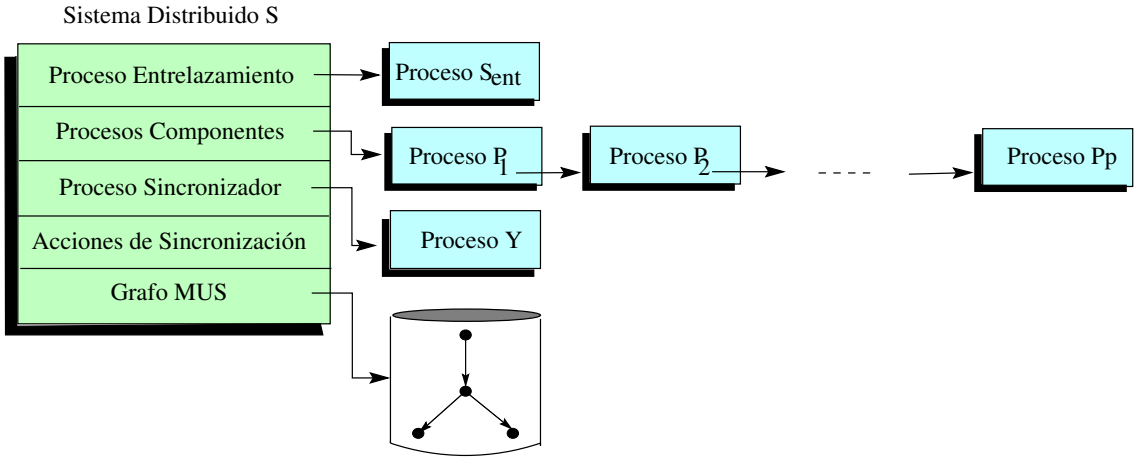


Figura 12.1: Mantenimiento de un Sistema Distribuido.

El proceso sincronizador $\mathcal{Y}$ se sintetiza a partir del proceso entrelazamiento $\mathcal{S}_{ent}$ y de un conjunto de requisitos de sincronización. Para ello, basta con aplicar el algoritmo de sincronización 11.3. Por otra parte, es posible reutilizar sincronizadores parciales, sintetizando el sincronizador global a partir de estos. Esto supone, tal y como se explicó en el capítulo 11, que cada proceso sincronizador mantenga la unión de estados de cada uno de sus grafos, ya que el sincronizador global se obtiene a partir de los grafos de los sincronizadores parciales con reducciones de estados comunes. La figura 12.2 muestra la estructura de datos almacenada para cada proceso sincronizador.

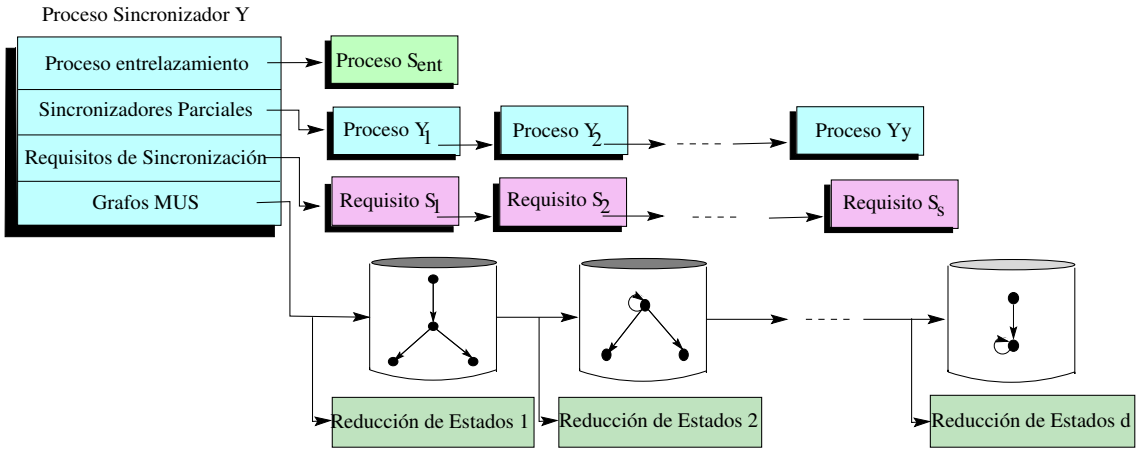


Figura 12.2: Mantenimiento de Procesos Sincronizadores.

Cada uno de los procesos componentes $\mathcal{P}_i$ se sintetiza de manera independiente, especificando requisitos SCTL, que pueden ser a su vez invarianzas o finalidades. La diferencia entre unas y otras reside en los estados de aplicabilidad de las mismas. Según el proceso de síntesis definido en los capítulos 9 y 10, éste se realiza de manera incremental, partiendo de un estado inicial en el que

se especifican qué eventos del proceso son posibles y cuáles no. Se pueden distinguir varios grados de incrementos en el proceso de síntesis. Por ejemplo, se puede considerar como incremento la especificación de un nuevo requisito, de manera que el sistema evoluciona desde un proceso $\mathcal{P}_{i,j}$ a otro $\mathcal{P}_{i,j+1}$, siendo la diferencia entre uno y otro la especificación del nuevo requisito.

Sin embargo, dicho incremento puede descomponerse en incrementos más pequeños, tomando como unidad de los mismos la síntesis de un requisito en un estado determinado. Dependiendo del tipo de requisito especificado –finalidad o invarianza–, serán necesarios más o menos incrementos de este tipo. Para distinguir entre los dos tipos de incrementos, a los primeros se les denominará **incrementos de requisito** y a los segundos **incrementos de grafo**. Sólo los incrementos de grafo que den origen a grafos compatibles –sin inconsistencias–, generarán **versiones** o evoluciones del proceso (ver figura 12.3).

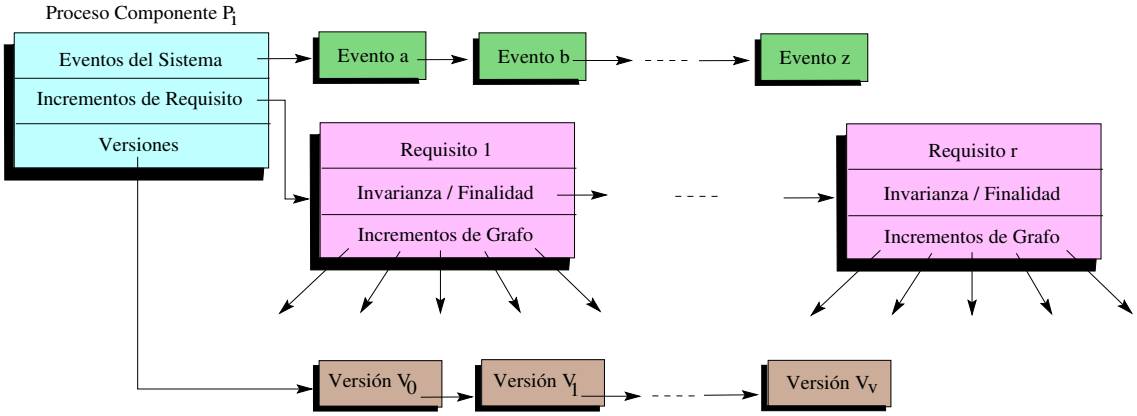


Figura 12.3: Mantenimiento de Procesos Componentes.

Cada incremento de grafo contiene información importante para la fase de mantenimiento. Esto es debido a que dichos incrementos se corresponden con llamadas al algoritmo de síntesis, en el que se toman decisiones. El almacenamiento de estas tomas de decisión es fundamental, ya que es posible que una modificación en el sistema no sea viable debido a la toma de decisión realizada en un incremento de grafo. La toma de una nueva decisión puede permitir evolucionar al sistema satisfaciendo nuevos requisitos.

Tal y como se explicó en el capítulo 10, las decisiones tomadas en cada incremento de grafo constan de dos partes: decisión del grafo del requisito que se sintetiza y decisión de estados en los que se solapan los estados libres del grafo elegido. A las primeras se les denomina **decisiones de grafo** y a las segundas **decisiones de estado**. Sólo las primeras cuyas decisiones de estado son todas compatibles producen un incremento de grafo. Es necesario, por tanto, almacenar en cada decisión –tanto de grafo como de estado– si dicha decisión es válida o no.

Las decisiones de estado forman una estructura de árbol, ya que en general, es necesario decidir el solapamiento de varios estados. Dicha elección se distribuye jerárquicamente, de manera que las decisiones de grafo válidas se corresponden con ramas del árbol de decisiones de estado cuya hoja contiene una decisión de estado válida. Es necesario, por tanto, almacenar en cada decisión de estado si ésta es la hoja –la última decisión– o no. Sólo las decisiones de estado que

son últimas y válidas generan decisiones de grafo válidas, originando **versiones** del proceso correspondiente. Además, es necesario almacenar en cada decisión de grafo, si ya se han agotado todas las decisiones de estado posibles (ver en la figura 12.4 el parámetro **E**nd **O**f **D**ecision). El almacenamiento de estos dos parámetros indicando si una decisión de estado es la última y si se han agotado las decisiones de estado para una decisión de grafo, permite calcularlas de manera interactiva o automática bajo demanda; en vez de tener que disponer de todas ellas cada vez que se realiza un nuevo incremento de grafo.

Cada incremento válido de grafo produce, por tanto, una evolución en el grafo del proceso que se está sintetizando. Dicha evolución se corresponde con una decisión de grafo que a su vez se corresponde con una serie de decisiones de estado. La figura 12.4 muestra la estructura almacenada para los incrementos de grafo y las versiones de procesos. Para relacionar las decisiones con cada versión, basta con que ésta apunte a la hoja de decisión de estado correspondiente. Además, cada versión se relaciona con la versión anterior del proceso, lo que permite recuperar la historia completa –la totalidad de incrementos– de cada versión.

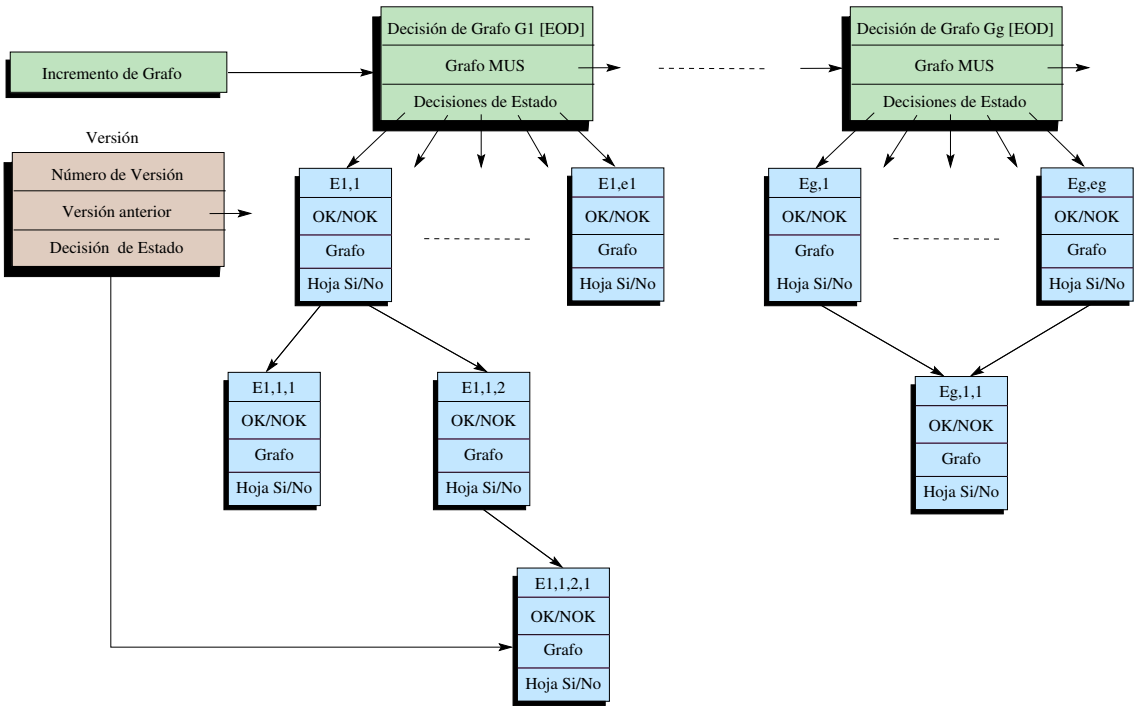


Figura 12.4: Mantenimiento de Incrementos, Versiones y Decisiones.

12.2 Especificaciones SCTL-MUS

En la sección anterior se han identificado las estructuras de almacenamiento necesarias para el mantenimiento de un sistema distribuido genérico. Para ello, se ha tenido en cuenta el proceso de síntesis incremental propuesto, así como las características de los procesos sincronizadores estudiados. A continuación, se aborda el estudio de la estructura de los requisitos SCTL y sus

grafos MUS, proponiendo mecanismos para su almacenamiento que permitan realizar las tareas de mantenimiento de los sistemas descritos en la sección anterior.

12.2.1 Requisitos SCTL

La estructura de las fórmulas de los requisitos SCTL es jerárquica y recursiva. Cada requisito –salvo los requisitos atómicos y los precedidos por el operador *negación*– puede descomponerse en dos subrequisitos unidos por un operador lógico o temporal. Si dicho operador es temporal, al primer subrequisito se le denomina premisa, y al segundo consecuencia.

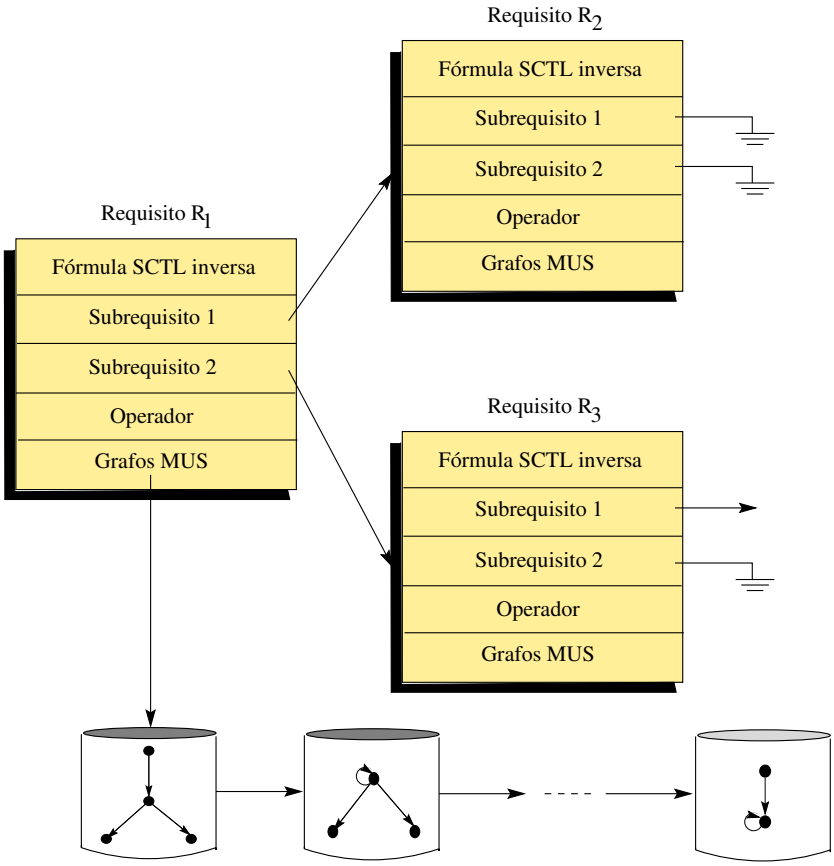


Figura 12.5: Mantenimiento de Requisitos SCTL.

Parece adecuado, por tanto, almacenar los requisitos SCTL siguiendo su estructura de subrequisitos, ya que ésta es la estrategia utilizada por la totalidad de algoritmos de la metodología SCTL-MUS. El proceso de traducción de un requisito SCTL a sus grafos MUS se realiza a partir de los grafos MUS de sus subrequisitos. El proceso de verificación se realiza de manera recursiva, obteniendo, previamente, el grado de satisfacción de los subrequisitos. Finalmente, el proceso de síntesis se realiza de manera incremental, añadiendo gradualmente los requisitos especificados, que pueden ser descompuestos en subrequisitos.

En la figura 12.5 se muestra la estructura de datos almacenada para un requisito SCTL. En ella, se incluye la fórmula SCTL en notación inversa, ya que es la utilizada por la totalidad de

algoritmos. Los requisitos atómicos no están compuestos por subrequisitos, mientras que los que se forman con el operador *negación*, contienen un único subrequisito.

12.2.2 Grafos MUS

El almacenamiento de los grafos MUS se reduce al almacenamiento de un conjunto de matrices. Dichas matrices representan los grafos subespecificados MUS, tal y como se definió en el capítulo 4. Para cada acción $a_i \in \Lambda$, existe una matriz $d[a_i]$ cuadrada de $N+1$ filas, siendo N el número de estados del grafo. La fila adicional se utiliza para el estado de subespecificación. Además, se almacena una matriz para la acción de subespecificación a_{sub} con las mismas características que las anteriores.

Los elementos de las matrices pueden tomar cuatro valores diferentes: $0, 1, \frac{1}{2}$ y $\frac{3}{4}$. Con dos *bits* bastaría para almacenar dichos valores; sin embargo, se recomienda utilizar un número mayor de *bits*, de manera que el sistema sea más flexible ante modificaciones en los grafos MUS.

12.2.3 Reutilización

El almacenamiento de las matrices correspondientes a los grafos MUS supone mantener un conjunto de estados en los que cada acción toma uno de los cuatro valores anteriormente citados. Esto supone el almacenamiento de una gran cantidad de información para cada grafo MUS.

Sin embargo, la información almacenada es siempre la misma: un arco etiquetado que se asocia con un estado origen, un estado destino y una acción. El mecanismo de almacenamiento que se propone se basa en utilizar los estados como entidades ficticias. La única información interesante de estos es una relación de orden entre ellos, e identificar un mismo estado ante distintos arcos. Sin embargo, es posible que la totalidad de grafos MUS del sistema comparta el mismo conjunto de estados ficticio. La información real de cada uno de ellos estará en su conjunto de arcos y acciones correspondientes (ver figura 12.6).

La codificación y compresión de los datos almacenados se ha dejado como línea futura de este trabajo. En este sentido puede estudiarse el almacenamiento reducido de ristras de valores, e incluso aplicar algoritmos de compresión basados en transformaciones de matrices.

Por otra parte, la estructura común de varios requisitos, así como la naturaleza de la información almacenada para los grafos, permite definir estructuras con un grado de abstracción mayor.

Se define un **MetaRequisito** como un requisito SCTL en el que sus acciones se sustituyen por **MetaAcciones** y sus operadores temporales por un **MetaOperador**. Cada MetaAcción representa cualquier evento o acción, y el MetaOperador representa cualquier operador temporal. Los MetaRequisitos mantienen la misma estructura jerárquica que los requisitos, de manera que un conjunto de requisitos puede ser representado por un único MetaRequisito. Por ejemplo, todos los requisitos atómicos comparten el siguiente MetaRequisito: $\_\mathcal{R}_{at} = true\_\bigotimes[\neg]\_a$. El símbolo $\_\bigotimes$ representa el MetaOperador, mientras que las MetaAcciones se preceden por el carácter $\_$

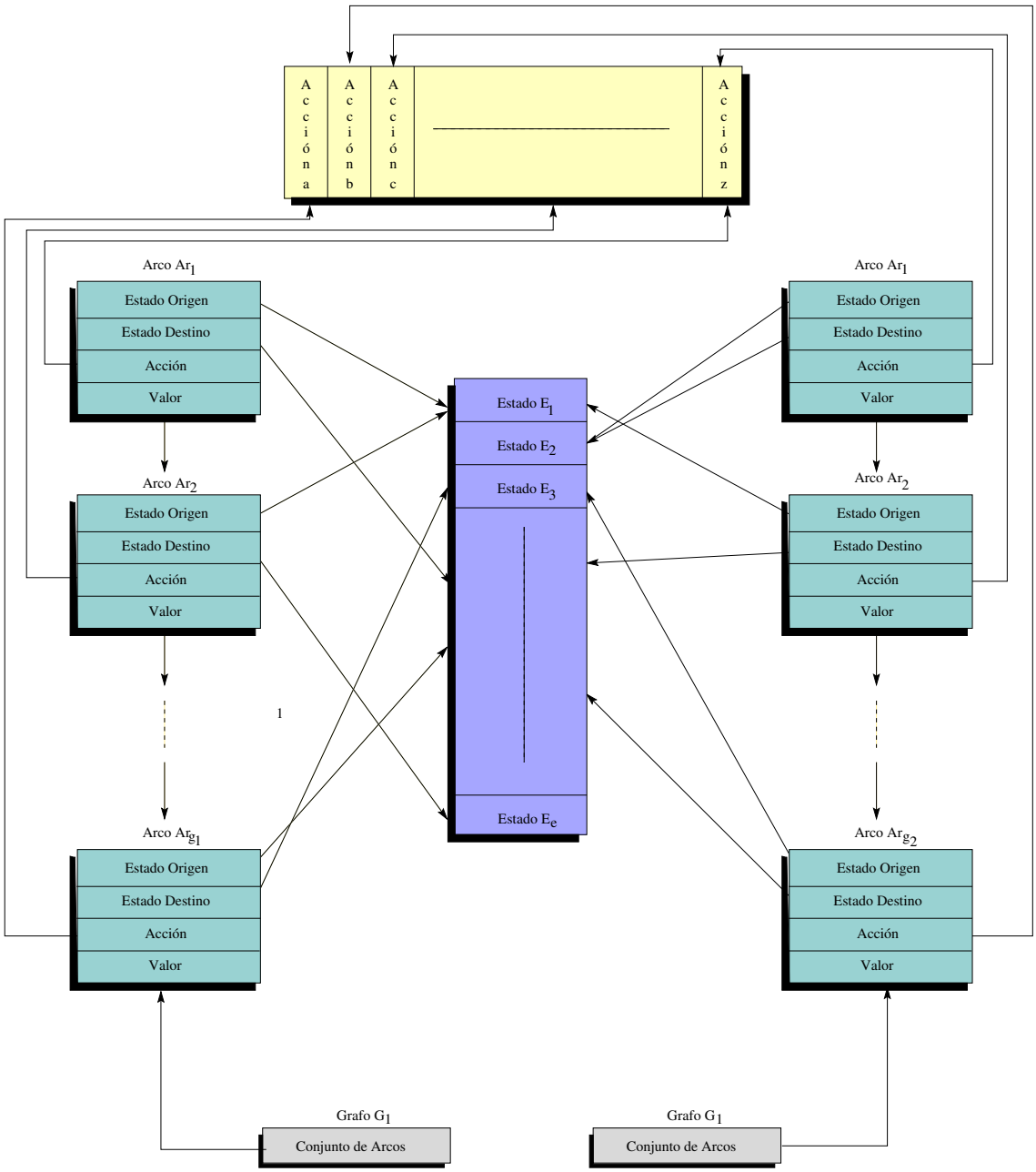


Figura 12.6: Mantenimiento de Grafos MUS.

Para obtener el grafo MUS de un MetaRequisito es necesario instanciar sus MetaOperadores por operadores temporales, ya que el grafo –el comportamiento del requisito– depende de estos. Por tanto, cada MetaRequisito podrá tener un conjunto de instanciaciones de operadores, y para cada una de ellas, se podrá obtener un conjunto de grafos MUS. Dichos grafos contienen MetaAcciones, por lo que se denominan **MetaGrafos**.

La definición del MetaGrafo permite que los requisitos que se diferencien únicamente en el nombre de sus acciones, compartan –se traduzca y se almacene una sola vez– el mismo grafo MUS, sin más que instanciar el MetaGrafo correspondiente. Además, la estructura jerárquica utilizada para el almacenamiento de los MetaRequisitos permite la reutilización de requisitos que

compartan subrequisitos. Esto proporciona un alto grado de reutilización, ya que la totalidad de requisitos se definen a partir de sus componentes, por lo que las "piezas" del sistema se sintetizarán una única vez, reutilizándolas en la síntesis de las "piezas" más grandes.

La instanciación de un MetaRequisito consiste, por tanto, en sustituir sus MetaAcciones por las acciones correspondientes, así como sus MetaOperadores por los operadores temporales de dicho requisito. Para ello, basta con recorrer las fórmulas SCTL de ambos y realizar dichas sustituciones. Sin embargo, se propone almacenar la relación de orden de cada acción (MetaAcción) y cada operador temporal (MetaOperador) en la fórmula SCTL. Esto permite realizar la instanciación sin analizar la estructura de las fórmulas SCTL (ver la figura 12.7).

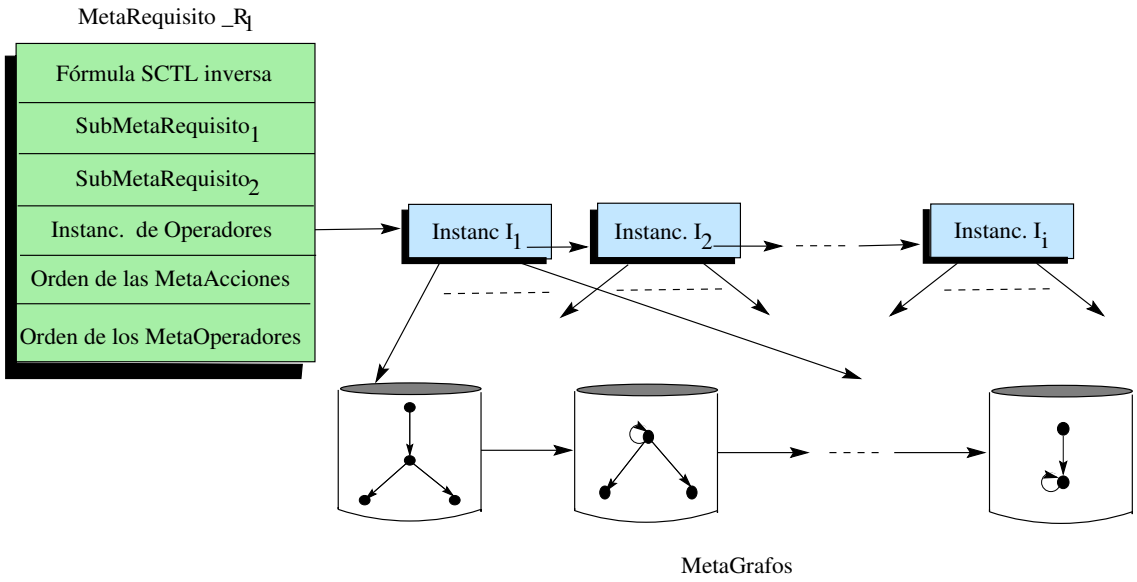


Figura 12.7: Estructura de los MetaRequisitos y MetaGrafos.

Parte VI

Implementación y Ejemplo de Aplicación

Capítulo 13

Implementación

Durante el desarrollo de este trabajo se han ido implementando cada uno de los algoritmos descritos. Estas implementaciones han permitido validar dichos algoritmos e identificar mejoras en los mismos. La implementación se ha realizado en el lenguaje de programación C [KR92], utilizando las herramientas Lex y $Yacc$ [MB91] para la generación de analizadores léxicos y gramaticales. Sin embargo, la implementación de dichos algoritmos no es suficiente para proporcionar al usuario una interfaz de acceso adecuada a la metodología SCTL-MUS desarrollada. Por ello, en este capítulo se describe el diseño y la funcionalidad de una herramienta software que da soporte al desarrollo de especificaciones con la metodología SCTL-MUS.

13.1 Diseño Genérico

En esta sección se describe la arquitectura de la herramienta diseñada, identificando sus componentes y la funcionalidad de cada uno de ellos. En la siguiente sección se particulariza la arquitectura general diseñada, tomando decisiones sobre las alternativas propuestas, razonando sobre las ventajas e inconvenientes de cada una de ellas. A continuación, se identifican los diferentes módulos de la herramienta software desarrollada, definiendo la funcionalidad de cada uno de ellos. Cada módulo se divide en dos, ya que se opta por una arquitectura cliente-servidor:

- Módulo de algoritmos SCTL-MUS: Este módulo se corresponde con la parte implementada durante el desarrollo de este trabajo. Su objetivo es proporcionar la posibilidad de ejecutar cada uno de los algoritmos desarrollados. Para ello, se divide en dos:
  - Cliente de algoritmos SCTL-MUS: Debe poder interaccionar con cada uno de los algoritmos desarrollados de una manera sencilla e intuitiva para el usuario.
  - Servidor de algoritmos SCTL-MUS: Se debe implementar en un lenguaje compilado (no interpretado), ya que debe poder atender de una manera eficiente las peticiones de varios clientes. Es necesario, por tanto, definir un protocolo mediante el cual el cliente y el servidor se intercambien los parámetros de entrada-salida necesarios, definiendo representaciones adecuadas para el transporte de los mismos.

- Módulo de datos: Este módulo será el encargado de mantener la información relativa a las especificaciones SCTL-MUS, tal y como se explicó en el capítulo 12. A este módulo también se le dota de arquitectura cliente-servidor:

  - El servidor consta de una base de datos relacional y un motor gestor de dicha base de datos o SGDB (Servidor Gestor de la Base de Datos). La base de datos mantiene, de una manera ordenada y eficiente, las estructuras de datos identificadas en el capítulo 12. El sistema gestor de la base de datos permite acceder a los clientes a dicha información.

    Se identifican dos tipos de clientes:

  - Clientes con acceso directo a la base de datos. Estos clientes utilizan el lenguaje SQL [Aba97] para acceder a la base de datos, lo que permite realizar de una manera eficiente y rápida el acceso a la información almacenada. El acceso se realiza a través de ODBC (*Open DataBase Connectivity*), lo que permite acceder a la base de datos mediante sentencias SQL estándar.

  - Clientes que no acceden directamente a la base de datos. Estos clientes no necesitan incorporar sentencias SQL en su código, ya que acceden a la base de datos a través de un servidor intermedio que actúa como un cliente directo. Es necesario, por tanto, definir un protocolo de comunicaciones entre el cliente y el servidor intermedio.

    La principal desventaja de este tipo de clientes es la ralentización de los accesos a la base de datos y su poca flexibilidad (las operaciones permitidas son las soportadas por el protocolo definido). Sin embargo, sus principales ventajas residen en la simplicidad de los clientes (la carga del acceso a los datos se sitúa en el servidor intermedio) y en el acceso seguro a los datos. La seguridad se basa en que sólo los servidores intermedios acceden directamente a los datos, y ningún cliente externo puede acceder a los mismos.

- Módulo de representación gráfica del sistema. El objetivo de este módulo es proporcionar una vista adecuada del marco de trabajo. Este módulo consta, por tanto, únicamente de clientes. La implementación de los mismos deberá ser portable e incluir interfaces gráficas fáciles de utilizar y atractivas. Dentro de este módulo se pueden identificar tantos clientes como vistas diferentes del entorno de trabajo:

  - Cliente genérico: Dicho cliente será una interfaz gráfica que muestre mediante componentes gráficos (menús, botones, listas...) los datos almacenados en la base de datos, interactuando con ellos durante todas las fases del ciclo de vida propuesto por la metodología SCTL-MUS.

  - Clientes generadores de especificaciones: Estos clientes pretenden dar una visión diferente del sistema implementado, considerando a éste como un generador de especificaciones. Por ello, su objetivo es presentar, en formatos de representación adecuados, el producto final de la metodología SCTL-MUS; es decir, la arquitectura inicial del sistema, resumiendo cada uno de sus componentes: acciones, requisitos, tomas de decisión, requisitos de sincronización y especificación E-LOTOS.

– Cliente de administración: Cliente gráfico específico que interactúe con los elementos de bajo nivel utilizados en la metodología SCTL-MUS; por ejemplo, con las matrices de representación de los grafos MUS. Está destinado a facilitar las tareas de depuración y modificación de los algoritmos desarrollados, y su acceso debe limitarse a usuarios administradores del sistema.

13.2 Diseño SCTL-MUS

En esta sección se particulariza la arquitectura genérica explicada en la sección anterior. En primer lugar, se opta por integrar todos los clientes en uno sólo, con el fin de dar una imagen homogénea del marco de trabajo. El cliente debe, por tanto, incluir una interfaz gráfica atractiva, representaciones gráficas de los elementos que intervienen en el diseño, y un mecanismo de acceso a la base de datos. Además, tiene que poder interactuar con el servidor de algoritmos SCTL-MUS.

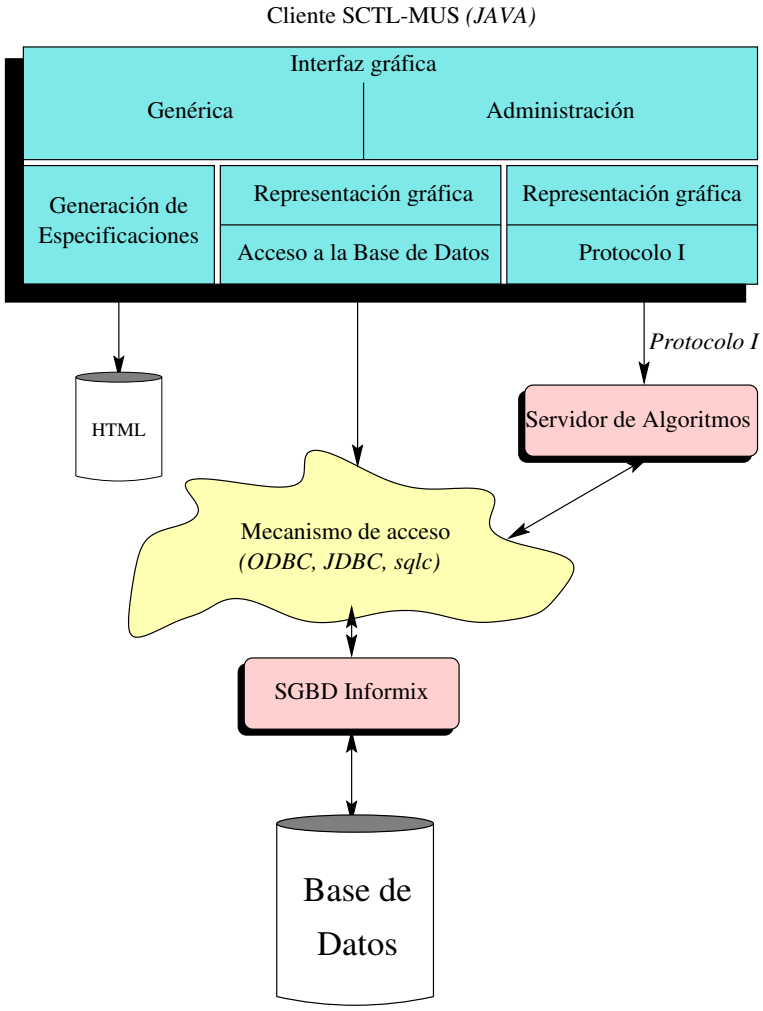


Figura 13.1: Implementación SCTL-MUS I.

La figura 13.1 muestra la arquitectura básica de la implementación SCTL-MUS. El lenguaje de implementación elegido para el cliente ha sido $Java$ [Mor99], ya que proporciona las características mencionadas en la sección anterior, además de permitir su distribución de una manera sencilla y portable mediante *applets*. Para la base de datos se ha elegido *Informix* [Inf96b], que proporciona una interfaz *sqlc* [Inf96a] para acceder directamente a la base de datos a través de una librería en C. Finalmente, se ha optado por un generador de especificaciones HTML, lo que permite su visualización a través del WWW y su exportación a otros formatos de representación.

Tanto el cliente como el servidor de algoritmos deben acceder a la base de datos, convirtiéndose así, en clientes del módulo de datos explicado en la sección anterior. Es posible, por tanto, optar por dos tipos de clientes. Para el cliente Java, la mejor solución sería el acceso directo a los datos, ya que permitiría desarrollar clientes muy flexibles y eficientes en el acceso a los datos. Para ello, se puede utilizar JDBC [Sip99] (*Java DataBase Connectivity*) de manera similar al ODBC. Sin embargo, el mayor problema de este tipo de clientes reside en la falta de seguridad en el acceso a los datos, y que pueden generar clientes complejos.

Por estas razones, se ha optado por una solución híbrida. Se utiliza un servidor intermedio que es el que proporciona un acceso controlado a la base de datos. Este acceso se realiza de manera indirecta, ya que el cliente sólo puede conectarse al servidor intermedio, y es éste el que le proporciona una conexión JDBC controlada mediante un sello temporal. Esto permite, además, restringir el acceso a través de JDBC, obligando a que el acceso a algunos datos sea obligatoriamente a través del servidor intermedio.

El acceso del servidor de algoritmos a los datos se implementa mediante un acceso directo a través de ODBC, ya que el acceso seguro está garantizado por tratarse el cliente de un servidor del sistema. La figura 13.2 muestra esta arquitectura. El servidor intermedio actúa como cliente directo de la base de datos a través de librerías SQL en C.

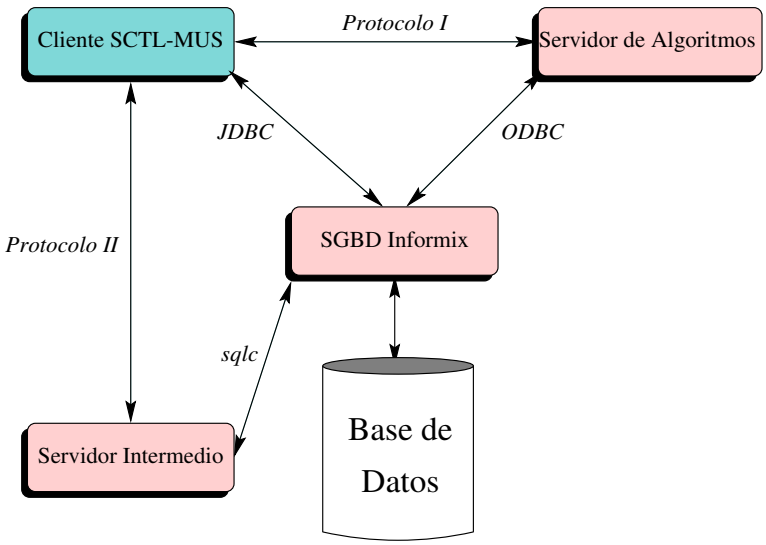


Figura 13.2: Implementación SCTL-MUS II.

Una vez obtenida una primera arquitectura, se explora la posibilidad de distribuir cada uno de los servidores de la misma. La base de datos se mantiene centralizada, dejando el diseño de una base de datos distribuida como línea futura de este trabajo. Sin embargo, es posible distribuir el servidor de algoritmos, creando varios servidores en diferentes máquinas. La arquitectura diseñada –debido a que el servidor de algoritmos no tiene por qué residir en la misma máquina que la base de datos– no sufre ninguna alteración, salvo la decisión de a qué servidor de algoritmos se conecta cada cliente. Esta cuestión se resolverá más adelante.

Otro punto de centralización de la arquitectura diseñada reside en la existencia de un único servidor intermedio. La multiprogramación de dicho servidor no resuelve el problema, ya que todos los clientes se conectan directamente a él. Además, dicho servidor puede atender peticiones de clientes que puede resolver sin acceder a la base de datos, bien sea para tareas de gestión de cada cliente, o mediante el mantenimiento de una caché. En este caso, parece lógico distribuir dicho servidor intermedio, descomponiéndolo en dos: un único servidor de acceso a la base de datos; y un conjunto de servidores que actúan como caché y descentralizan el acceso a la base de datos.

El servidor de acceso a la base de datos puede actuar como punto inicial de la conexión, proporcionando al usuario un servidor intermedio y un servidor de algoritmos. Para decidir qué servidores elegir se pueden utilizar parámetros de carga –almacenados en la base de datos– y de proximidad al cliente –se le puede proporcionar una lista al cliente y que él elija el más próximo (el que le responda antes)–. La figura 13.3 muestra la arquitectura de la nueva implementación.

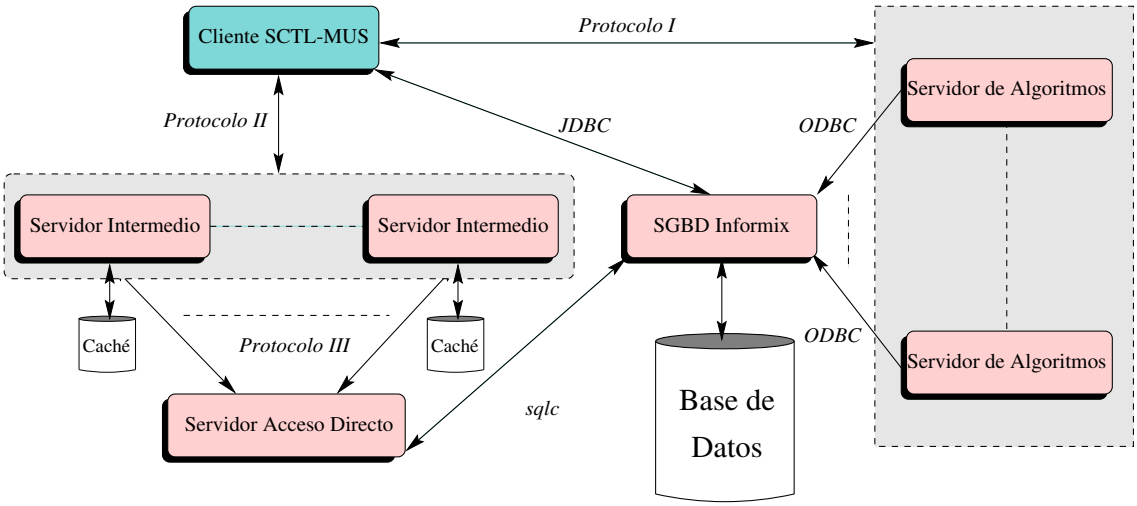


Figura 13.3: Implementación SCTL-MUS III.

Finalmente, se modifica dicha arquitectura para reducir el número de conexiones a la base de datos, ya que cada servidor de algoritmos abre una conexión ODBC con ella. Por ello, se utiliza un nuevo servidor local de acceso directo a la base de datos a través de la librería *sqlc* –obsérvese que esta decisión ya se ha tomado implícitamente en los servidores intermedios–. De esta manera, en la base de datos existe una conexión para el sistema de servidores de algoritmos, otra para los

servidores intermedios y una JDBC para cada cliente. La figura 13.4 muestra la arquitectura final de la implementación SCTL-MUS diseñada, en la que se utiliza el mismo servidor de acceso directo para los servidores intermedios y para los servidores de algoritmos.

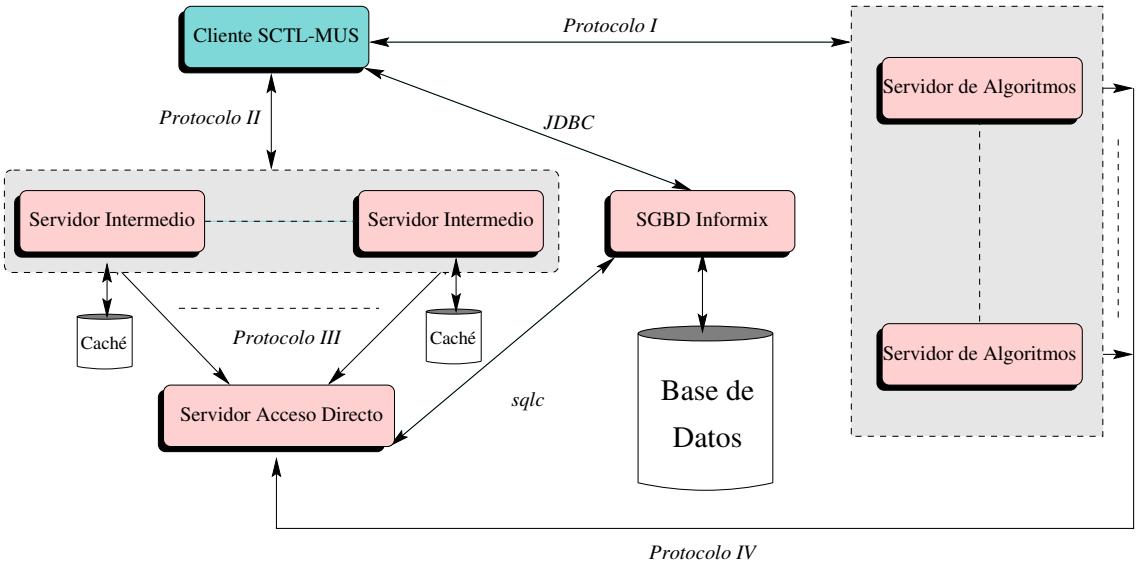


Figura 13.4: Implementación SCTL-MUS IV.

La figura 13.5 muestra la ventana principal del cliente SCTL-MUS desarrollado. Consiste en un *applet* Java que proporciona una visión jerárquica de cada uno de los elementos que participan en la metodología SCTL-MUS. Cada usuario dispone de un conjunto de sesiones de trabajo. En cada sesión se pueden diseñar varios sistemas, compartiendo todos ellos el mismo conjunto de eventos. Además, cada sistema se compone de un conjunto de incrementos de requisitos, en los que se puede realizar o automatizar las decisiones de síntesis. Los grafos MUS se pueden mostrar gráficamente o a bajo nivel, mediante su representación matricial. La representación gráfica de los mismos facilita la interacción con el sistema, seleccionando estados donde aplicar la síntesis o verificación de requisitos, o descomponiendo cada grafo en sus grafos componentes.

La estructura de MetaRequisitos y MetaGrafos se oculta totalmente a los usuarios, y se permite a estos que puedan compartir cada una de las entidades o elementos jerárquicos del sistema. Para ello, se proporciona un mecanismo de permisos a nivel de entidad, de manera que cada usuario puede dar permiso de lectura y/o escritura a cada sesión, sistema, evento, requisito, etc.

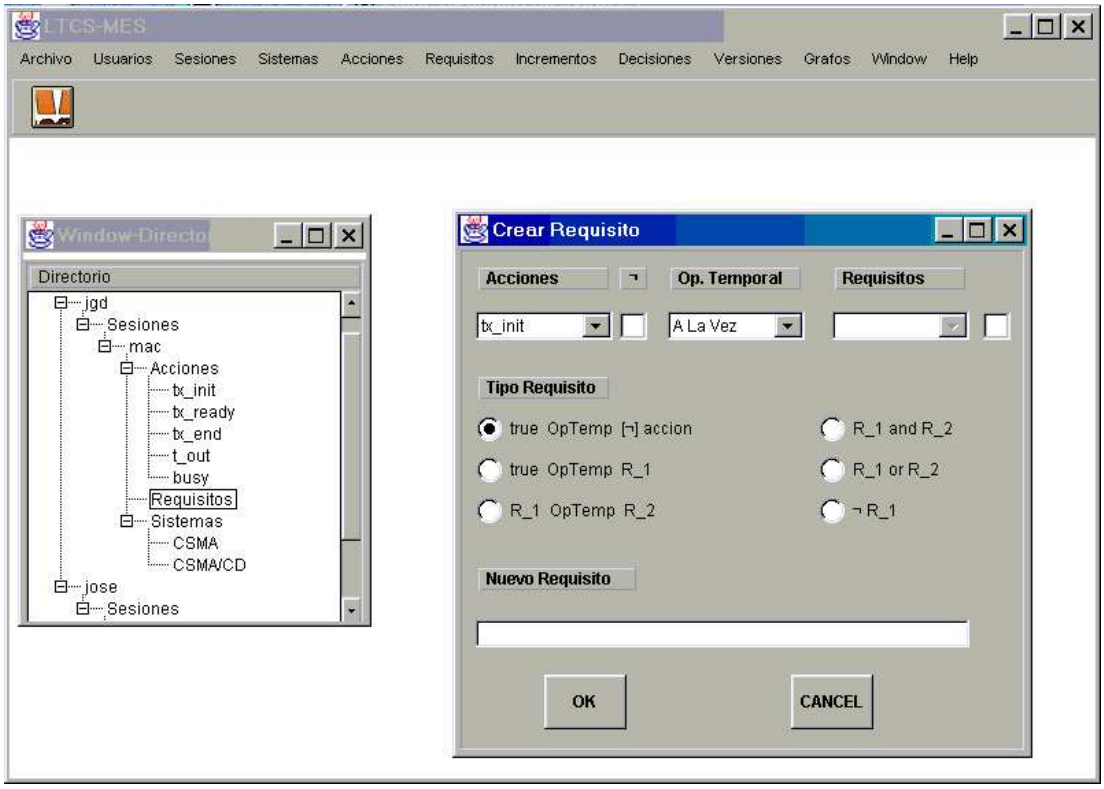


Figura 13.5: Implementación SCTL-MUS V.

Capítulo 14

Protocolo CSMA/CD

La técnica de acceso al medio CSMA/CD [Sta97, Tan96] (*Carrier Sense Multiple Access/Collision Detection*) y sus precursoras pueden ser denominadas de acceso aleatorio o de contienda. Son de acceso aleatorio en el sentido de que no existe un tiempo preestablecido o predecible para la transmisión de las estaciones: ésta se realiza aleatoriamente. Son de contienda en el sentido de que las estaciones compiten para conseguir tiempo del medio.

En este capítulo se realiza la síntesis completa de un proceso emisor según el protocolo CSMA/CD, así como la síntesis de un proceso sincronizador entre las estaciones emisoras. Para ello, y con el fin de mostrar la reutilización en el proceso de síntesis, se aborda primero la síntesis con el protocolo CSMA, refinando dicho protocolo para obtener CSMA/CD.

14.1 CSMA

14.1.1 Descripción

Con la técnica CSMA (*Carrier Sense Multiple Access*), una estación que desea transmitir escucha primero el medio para determinar si existe alguna otra transmisión en curso (sensible a la portadora). Si el medio se está usando, la estación debe esperar. En cambio, si se encuentra libre, la estación puede transmitir. Puede suceder que dos o más estaciones intenten transmitir aproximadamente al mismo tiempo, en cuyo caso se producirá colisión: los datos de ambas transmisiones interfieren y no se reciben con éxito. Para solucionar esto, cada estación debe implementar el mecanismo de retención y retransmisión ante la ausencia de un asentimiento.

Esta estrategia resulta efectiva para redes en las que el tiempo de transmisión de trama es mucho mayor que el retardo máximo de propagación σ. Las colisiones sólo se producen cuando más de un usuario comienza a transmitir con diferencias pequeñas de tiempo –menores que σ–. Si una estación comienza a transmitir una trama y no existen colisiones durante el retardo máximo de propagación, no se producirá colisión para esta trama dado que en ese instante todas las estaciones están enteradas de la transmisión.

Con CSMA es necesario un algoritmo para especificar lo que debe hacer una estación si encuentra el medio ocupado. La técnica 1–persistente es la aproximación más usual, y es la utilizada en IEEE 802.3. Una estación que desea transmitir escucha el medio y sigue las siguientes reglas:

1. Transmite si el medio se encuentra libre, si no se aplica la regla 2.
2. Si el medio está ocupado, continúa escuchando hasta que el canal se detecta libre, entonces transmite inmediatamente.

Se producirá colisión si dos o más estaciones están en espera de transmitir.

14.1.2 Especificación de Acciones

A continuación, se describen las acciones o eventos observables en una estación emisora $\mathcal{E}_i$:

- Estar preparada para el envío o transmisión de una nueva trama: $\mathbf{tx\_ready_i}$.
- Desde que una estación está preparada para transmitir hasta que efectivamente comienza con una transmisión, puede transcurrir un tiempo debido a que el medio esté ocupado. Por ello, se identifican dos nuevos eventos que representan la detección del medio ocupado y el momento en el que la estación comienza la transmisión de una trama: $\mathbf{busy_i}$ y $\mathbf{tx\_init_i}$.
- La finalización de la transmisión de una trama: $\mathbf{tx\_end_i}$.
- Suponiendo que las tramas no "quepan" en el medio, es decir, que el tiempo de transmisión de una trama es mayor que σ –el tiempo máximo de propagación por el medio–, se identifica un nuevo evento. Dicho evento consiste en el vencimiento de un temporizador de valor σ, ya que una vez que pasa dicho tiempo de transmisión, todas las demás estaciones detectan el medio ocupado: $\mathbf{t\_out\_\sigma_i}$.

14.1.3 Especificación de los Requisitos de una Estación Emisora

En primer lugar, se debe especificar el estado inicial. En este caso, sólo un evento puede ser posible inicialmente: estar preparado para transmitir. $E_0[tx\_ready_i] = 1$, $E_0[tx\_init_i] = 0$, $E_0[t\_out\_\sigma_i] = 0$, $E_0[tx\_end_i] = 0$, $E_0[busy_i] = 0$.

Además, se identifican cuatro invarianzas que debe satisfacer una estación emisora $\mathcal{E}_i$:

1. Una vez que una estación está lista para transmitir, pueden suceder dos cosas: que se detecte el medio ocupado; o que se empiece la transmisión.

req $\mathcal{I}_{1,i}$ **is**
$$(true \Rightarrow tx\_ready_i) \Rightarrow \bigcirc\ ((true \Rightarrow busy_i) \wedge (true \Rightarrow tx\_init_i))$$
endreq

2. Si el medio está ocupado, el inicio de la transmisión se demora hasta que el medio queda libre.

req $\mathcal{I}_{2,i}$ **is**
$(true \Rightarrow busy_i) \Rightarrow \bigcirc\ ((true \Rightarrow busy_i) \wedge (true \Rightarrow tx\_init_i))$
endreq

3. El protocolo CSMA no detecta las colisiones. Por tanto, una vez que se inicia la transmisión de una trama, ésta finaliza y ningún otro evento puede producirse durante la transmisión. Dado que una trama no cabe en el medio, en primer lugar vencerá el temporizador σ y posteriormente finalizará la transmisión.

req $\mathcal{I}_{3,i}$ **is**
$(true \Rightarrow tx\_init_i) \Rightarrow \bigcirc\ ((true \Rightarrow t\_out\_\sigma_i) \wedge (true \Rightarrow \neg tx\_ready_i) \wedge$
$\wedge\ (true \Rightarrow \neg tx\_init_i) \wedge (true \Rightarrow \neg tx\_end_i) \wedge$
$\wedge\ (true \Rightarrow \neg busy_i))$
endreq

req $\mathcal{I}_{4,i}$ **is**
$(true \Rightarrow tx\_out\_\sigma_i) \Rightarrow \bigcirc\ ((true \Rightarrow tx\_end_i) \wedge (true \Rightarrow \neg tx\_ready_i) \wedge$
$\wedge\ (true \Rightarrow \neg tx\_init_i) \wedge (true \Rightarrow \neg t\_out\_\sigma_i) \wedge$
$\wedge\ (true \Rightarrow \neg busy_i))$
endreq

4. Una vez concluida la transmisión la estación vuelve al estado inicial.

req $\mathcal{I}_{5,i}$ **is**
$(true \Rightarrow tx\_end_i) \Rightarrow \bigcirc\ ((true \Rightarrow tx\_ready_i) \wedge (true \Rightarrow \neg tx\_init_i) \wedge$
$\wedge\ (true \Rightarrow \neg tx\_end_i) \wedge (true \Rightarrow \neg t\_out\_\sigma_i) \wedge$
$\wedge\ (true \Rightarrow \neg busy_i))$
endreq

14.1.4 Traducción SCTL-MUS

Antes de pasar a la fase de síntesis o verificación, es necesario obtener los grafos MUS correspondientes a los requisitos SCTL especificados. Además, para el almacenamiento y mantenimiento del sistema especificado, se crea una base de datos con la información necesaria a cada uno de dichos requisitos, tal y como se describe en el capítulo 12.

En primer lugar, se transforman los requisitos especificados en MetaRequisitos, obteniéndose los MetaRequisitos $\_\mathcal{R}_3$, $\_\mathcal{R}_4$ y $\_\mathcal{R}_7$. Además, de manera transparente al usuario, se dan de alta los MetaRequisitos componentes $\_\mathcal{R}_1$, $\_\mathcal{R}_2$, $\_\mathcal{R}_5$ y $\_\mathcal{R}_6$.

req $\_\mathcal{R}_3$ **is**
$(t \bigoplus \_a) \bigoplus ((t \bigoplus \_b) \wedge (t \bigoplus \_c))$
endreq

req $\_\mathcal{R}_4$ **is**
$(t \bigoplus \_a) \bigoplus ((t \bigoplus \_a) \wedge (t \bigoplus \_b))$
endreq

req $\_\mathcal{R}_7$ **is**
$(t \bigoplus \_a) \bigoplus ((t \bigoplus \_b) \wedge (t \bigoplus \neg\_c) \wedge (t \bigoplus \neg\_d) \wedge (t \bigoplus \neg\_a) \wedge (t \bigoplus \neg\_e))$
endreq

req $\_\mathcal{R}_1$ **is**
$t \bigoplus \_a$
endreq

req $\_\mathcal{R}_6$ **is**
$(t \bigoplus \_a) \wedge (t \bigoplus \neg\_b) \wedge (t \bigoplus \neg\_c) \wedge (t \bigoplus \neg\_d) \wedge (t \bigoplus \neg\_e)$
endreq

req $\_\mathcal{R}_2$ **is**
$(t \bigoplus \_a) \wedge (t \bigoplus \_b)$
endreq

req $\_\mathcal{R}_5$ **is**
$(t \bigoplus \_a) \wedge (t \bigoplus \neg\_b) \wedge (t \bigoplus \neg\_c) \wedge (t \bigoplus \neg\_d)$
endreq

Mediante el algoritmo de traducción 10.3 se obtienen los MetaGrafos de cada uno de los MetaRequisitos. Primero se aplica dicho algoritmo al MetaRequisito atómico $\_\mathcal{R}_1$, obteniéndose dos MetaGrafos, tal y como se muestra en la figura 14.1. Los grafos del MetaRequisito $\_\mathcal{R}_2$ se obtienen a partir de los MetaGrafos de $\_\mathcal{R}_1$.

De manera similar, los MetaGrafos de $\_\mathcal{R}_3$ se obtienen a partir de sus componentes o subrequisitos $\_\mathcal{R}_1$ y $\_\mathcal{R}_2$. Los MetaGrafos de $\_\mathcal{R}_4$ coinciden con los de $\_\mathcal{R}_3$, salvo en que la MetaAcción $\_b$ se debe sustituir por la misma MetaAcción $\_a$. Esto supone que los MetaGrafos cuarto, séptimo y decimotercero desaparezcan, ya que en ellos los estados siguientes a dichas acciones son diferentes, y al unirlos se obtienen MetaGrafos ya existentes.

Las invarianzas $\mathcal{I}_1$ e $\mathcal{I}_2$ se corresponden con instanciaciones de los MetaRequisitos $\_\mathcal{R}_3$ y $\_\mathcal{R}_4$ respectivamente. La figura 14.2 muestra la estructura de MetaRequisitos y MetaGrafos correspondientes a las invarianzas $\mathcal{I}_3, \mathcal{I}_4$ e $\mathcal{I}_5$, que se corresponden con instanciaciones del MetaRequisito $\_\mathcal{R}_7$.

14.1.5 Síntesis Incremental

Para la síntesis automática del grafo MUS de un proceso que satisfaga los requisitos especificados, se aplica el proceso de síntesis incremental descrito en los capítulos 9 y 10. Por tanto, se parte de un grafo MUS inicial que consta únicamente del estado E_0 (ver la figura 14.3), y se le van añadiendo, gradualmente, los grafos de los requisitos especificados.

En el proceso de síntesis incremental primero se sintetiza la invarianza $\mathcal{I}_1$, cuya premisa se satisface en el estado E_0. Para ello, se elige su grafo MUS más simple que puede ser sintetizado en el grafo del sistema. La síntesis de $\mathcal{I}_1$ implica la aparición de un estado en el que se satisface

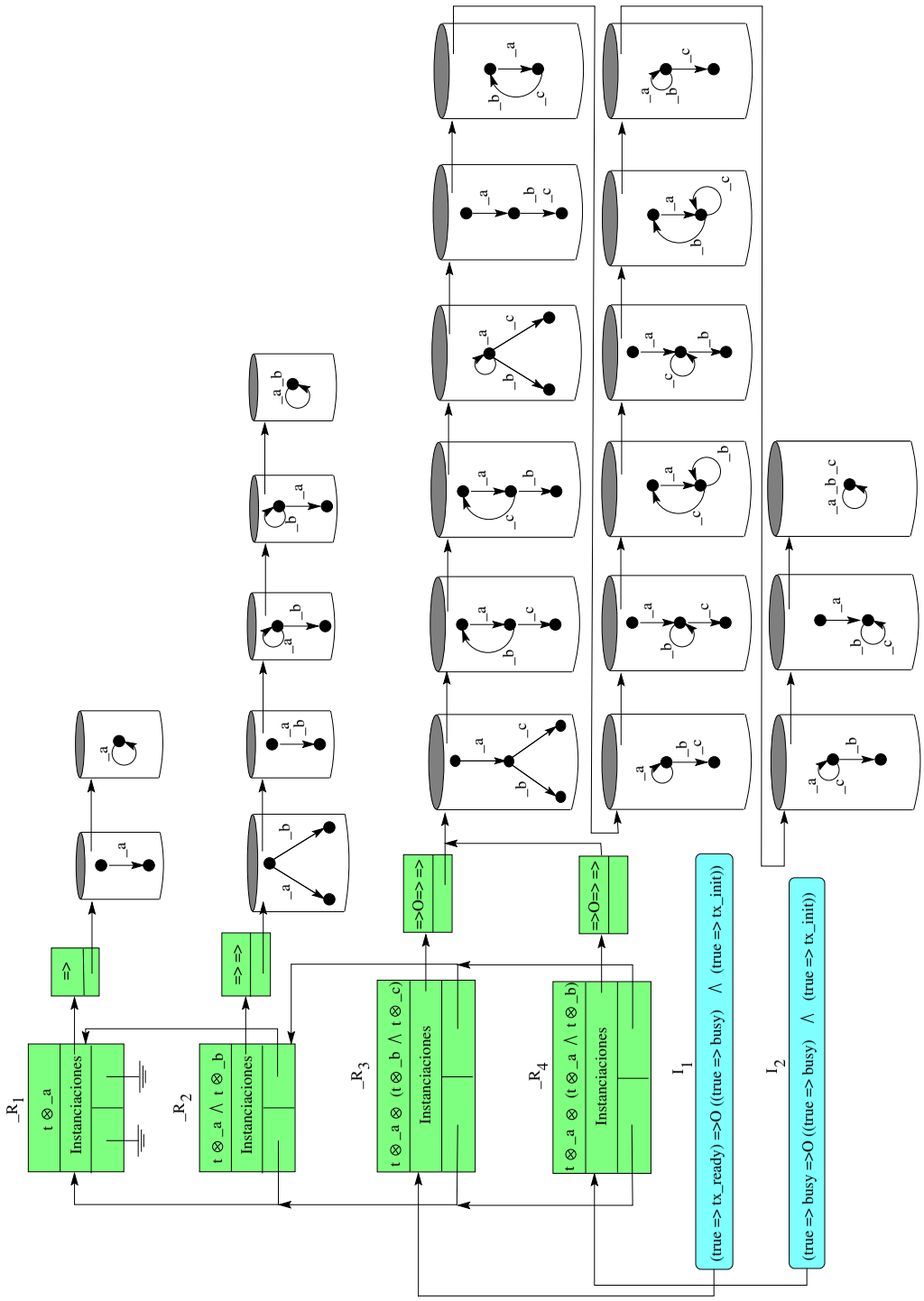


Figura 14.1: Estructura de MetaRequisitos I.

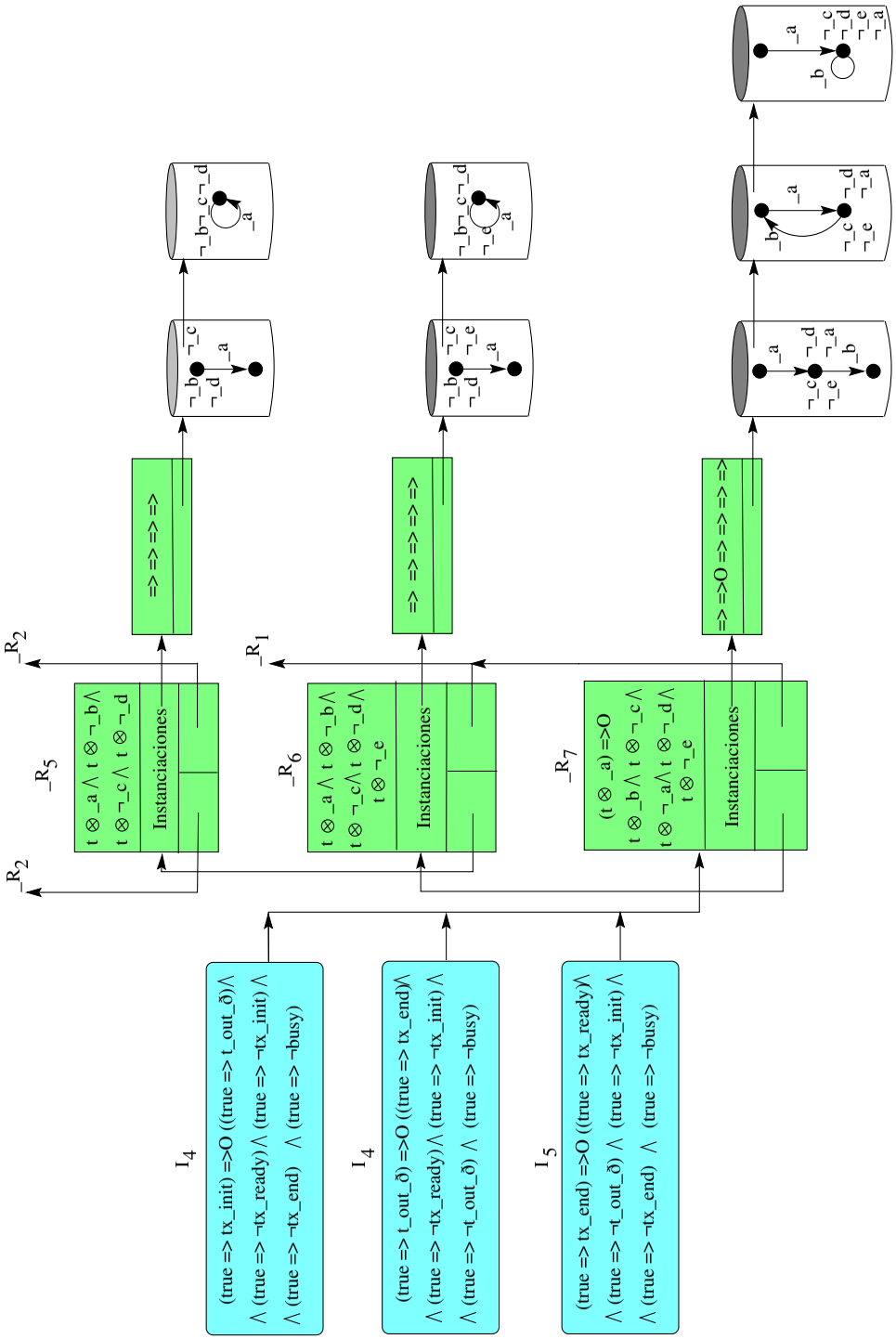


Figura 14.2: Estructura de MetaRequisitos II.

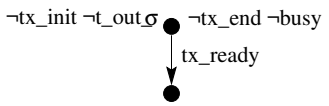
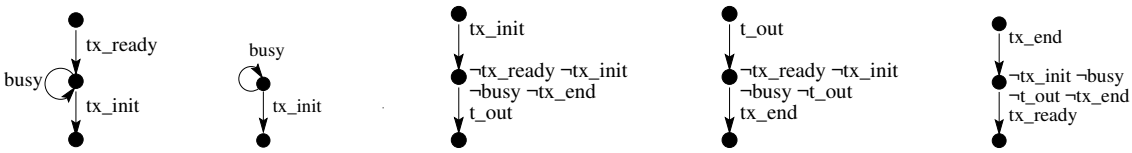


Figura 14.3: Grafo MUS inicial del sistema $\mathcal{M}_{\mathcal{S}}$.

la premisa de la invarianza $\mathcal{I}_2$, por lo que ésta se añade al sistema eligiendo de nuevo su grafo compatible más simple. Este proceso se repite hasta que las cinco invarianzas especificadas se sintetizan en los estados del sistema en los que son aplicables. Esto supone, tal y como se explicó en el capítulo 10, la toma de decisiones, tanto del grafo a sintetizar –el mínimo– como de los estados donde éste se solapa.

La figura 14.4 muestra los grafos MUS de los requisitos especificados que han sido utilizados como primera elección por el algoritmo de síntesis. Dichos grafos son los menores de cada requisito que permiten un solapamiento consistente.



(a) $\mathcal{M}_{I_1}$ (b) $\mathcal{M}_{I_2}$ (c) $\mathcal{M}_{I_3}$ (d) $\mathcal{M}_{I_4}$ (e) $\mathcal{M}_{I_5}$

Figura 14.4: CSMA: Grafos utilizados por el Algoritmo de Síntesis.

La figura 14.5 muestra la síntesis del grafo MUS que satisface los requisitos especificados. Dicho grafo representa el comportamiento de una estación emisora según el protocolo CSMA.

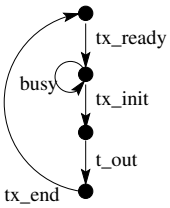


Figura 14.5: CSMA: Grafo final de una estación emisora $\mathcal{E}$.

El proceso de síntesis incremental se ha resumido, mostrando los grafos que han sido utilizados por el algoritmo de síntesis. Sin embargo, conviene destacar algunas características de este proceso, que se enumeran a continuación:

- La especificación del estado inicial se traduce a un requisito SCTL especificado como finalidad, de manera que éste es el primer requisito sintetizado por el proceso de síntesis. Dicha finalidad se construye a partir de la constante $true$, el operador *A la vez* y la especificación de las acciones del estado inicial.

- La especificación de una invarianza supone que ésta debe sintetizarse en todos los estados del

sistema en los que se satisfaga su premisa. Por otra parte, dicha premisa podría satisfacerse también en alguno de los grafos de otro requisito especificado, que a su vez se sintetizará en los estados del grafo del sistema correspondientes. Esto sugiere realizar síntesis parciales de cada invarianza en cada uno de los requisitos –finalidades e invarianzas– especificados, con dos objetivos fundamentales:

1. Reutilización: Sea $\mathcal{I}_i$ una invarianza tal que su premisa se satisface en un grafo $\mathcal{G}_{\mathcal{R}_j}$ de un requisito $\mathcal{R}_j$. Cada vez que sea necesario sintetizar el grafo $\mathcal{G}_{\mathcal{R}_j}$ en el sistema, será necesario sintetizar, además, la invarianza $\mathcal{I}_i$. Es posible reutilizar dicha síntesis si previamente se realiza una síntesis parcial de la invarianza $\mathcal{I}_i$ en dicho grafo. De esta manera, la invarianza $\mathcal{I}_i$ sólo se sintetizará en aquellos estados del grafo en los que no se ha sintetizado $\mathcal{G}_{\mathcal{R}_j}$, ya que en estos su síntesis se realiza de manera implícita. Cabe destacar que esta estrategia para reutilizar síntesis parciales es la que se utiliza en el algoritmo de síntesis mediante la aplicación de reducción de estados, tal y como se explicó en el capítulo 10.

2. Aprendizaje: Si dicho proceso de síntesis parcial se realiza a nivel de MetaRequisitos y MetaGrafos, además de la reutilización en el proceso propio de síntesis, se obtiene reutilización en la síntesis de requisitos que comparten los mismos MetaGrafos. Para ello, basta con realizar la síntesis de los MetaRequisitos correspondientes, haciendo una instanciación de MetaAcciones que permita identificar las MetaAcciones comunes de ambos MetaRequisitos. Esto permite dotar al sistema de una capacidad de aprendizaje, de manera que nunca repita síntesis parciales. Esto supone un nivel alto de reutilización ya que la estructura jerárquica de los requisitos SCTL siempre permite descomponer a estos en subrequisitos más pequeños, cuyo conocimiento (traducción, síntesis y verificación) puede ser reutilizado.

La especificación del estado inicial supone, por tanto, la especificación de una finalidad $\mathcal{F}_1$, cuyo MetaRequisito es $\_\mathcal{R}_6$, y por tanto se reutiliza el proceso de obtención de sus MetaGrafos.

req $\mathcal{F}_1$ **is**
$(true \Rightarrow tx\_ready) \wedge (true \Rightarrow \neg tx\_init) \wedge (true \Rightarrow \neg t\_out\_\sigma) \wedge (true \Rightarrow \neg tx\_end)$
$\wedge (true \Rightarrow \neg busy)$
endreq

14.1.5.1 Síntesis Parcial

La finalidad que representa el estado inicial satisface la premisa de la invarianza $\mathcal{I}_1$, por lo que es posible realizar una síntesis parcial de ambas. Además, dicha síntesis se realiza a nivel de MetaAcciones, para poder reutilizarla según lo expuesto anteriormente. Para ello, basta con realizar una instanciación que permita identificar las MetaAcciones comunes e indicar los estados

donde se produce la síntesis –el solapamiento o unión de ambos MetaGrafos–. En este caso, la síntesis parcial se puede representar por:

$$\mathcal{F}_1 \bigcup \mathcal{I}_1 \;=\; \_\mathcal{R}_6(\_a, \_b, \_c, \_d, \_e) \bigcup_{E_0,E_0} \_\mathcal{R}_3(\_a, \_e, \_b)$$

Esta unión obtiene como resultado un conjunto de MetaGrafos similares a los de la invarianza $\_\mathcal{R}_3$, en los que se añade al estado E_0 la especificación de las acciones negadas en $\_\mathcal{R}_6$. Dicha unión supone la eliminación de grafos incompatibles, en este caso los grafos cuarto, séptimo, duodécimo, decimotercero y decimoquinto –ver figura 14.1–.

De manera similar, se realizan síntesis parciales del resto de invarianzas. La premisa de $\mathcal{I}_1$ se satisface también en los MetaGrafos de $\mathcal{I}_5$:

$$\mathcal{I}_5 \bigcup \mathcal{I}_1 \;=\; \_\mathcal{R}_7(\_a, \_b, \_c, \_a, \_d, \_e) \bigcup_{E_1,E_0} \_\mathcal{R}_3(\_c, \_e, \_b)$$

La premisa de $\mathcal{I}_2$ se satisface en los grafos de $\mathcal{I}_1$ y en sus propios grafos. Ésta es una síntesis recursiva, por lo que en los grafos resultantes habrá instanciaciones a dicha invarianza que serán desplegados por los algoritmos a medida que esto sea necesario.

$$\begin{aligned} \mathcal{I}_1 \bigcup \mathcal{I}_2 \;&=\; \_\mathcal{R}_3(\_a, \_b, \_c) \bigcup_{E_1,E_0} \_\mathcal{R}_4(\_b, \_b, \_c) \\ \mathcal{I}_2 \bigcup \mathcal{I}_2 \;&=\; \_\mathcal{R}_4(\_a, \_a, \_b) \bigcup_{E_1,E_0} \_\mathcal{R}_4(\_a, \_a, \_b) \end{aligned}$$

La premisa de $\mathcal{I}_3$ se satisface en los grafos de $\mathcal{I}_1$ e $\mathcal{I}_2$:

$$\begin{aligned} \mathcal{I}_1 \bigcup \mathcal{I}_3 \;&=\; \_\mathcal{R}_3(\_a, \_b, \_c) \bigcup_{E_1,E_0} \_\mathcal{R}_7(\_c, \_d, \_a, \_c, \_e, \_b) \\ \mathcal{I}_2 \bigcup \mathcal{I}_3 \;&=\; \_\mathcal{R}_4(\_a, \_a, \_b) \bigcup_{E_1,E_0} \_\mathcal{R}_7(\_b, \_c, \_d, \_b, \_e, \_a) \end{aligned}$$

La premisa de $\mathcal{I}_4$ se satisface en $\mathcal{I}_3$:

$$\mathcal{I}_3 \bigcup \mathcal{I}_4 \;=\; \_\mathcal{R}_7(\_a, \_b, \_c, \_a, \_d, \_e) \bigcup_{E_1,E_0} \_\mathcal{R}_7(\_b, \_d, \_c, \_a, \_b, \_e)$$

La premisa de $\mathcal{I}_5$ se satisface en $\mathcal{I}_4$. En este caso se reutiliza la síntesis parcial anterior, ya que se realiza la misma unión de MetaGrafos:

$$\mathcal{I}_4 \bigcup \mathcal{I}_5 \;=\; \_\mathcal{R}_7(\_a, \_b, \_c, \_a, \_d, \_e) \bigcup_{E_1,E_0} \_\mathcal{R}_7(\_b, \_d, \_c, \_a, \_b, \_e)$$

Este proceso de síntesis parcial se puede repetir de manera recursiva sobre las síntesis parciales obtenidas, ya que éstas pueden satisfacer en alguno de sus nuevos estados alguna nueva invarianza. Las síntesis parciales permiten obtener un alto grado de reutilización sin pérdida de eficiencia. Obsérvese que para sintetizar el grafo del sistema de la figura 14.5 sólo se realizan las síntesis parciales necesarias, el resto se pueden realizar sin detener el actual proceso de síntesis, demorarlas hasta que el sistema quede libre, o incluso realizarlas bajo demanda –sólo cuando sea estrictamente necesario–.

14.1.6 Diseño de la Arquitectura

El diseño de la arquitectura de este tipo de protocolos tiene una complejidad añadida, ya que la sincronización se realiza entre un número indefinido de procesos, uno por cada estación emisora de la red local.

La sincronización que tiene lugar entre las distintas estaciones emisoras se relaciona con la detección del medio ocupado cuando una está transmitiendo. Una vez que una estación comienza una transmisión, la finaliza, y después del tiempo máximo de propagación σ, cualquier estación preparada verá el medio ocupado. Por tanto, se especifica la siguiente invarianza como requisito de sincronización:

- Cualquier estación $\mathcal{E}_j$ es capaz de detectar el medio ocupado después de que otra estación $\mathcal{E}_i$ lleve transmitiendo σ unidades de tiempo. Es decir, si una estación lleva σ unidades de tiempo transmitiendo –puede terminar la transmisión–, las demás no pueden iniciar la transmisión de una trama; ya que detectarán el medio ocupado. $\forall j \neq i$:

req $\mathcal{I}_{sin_i}$ **is**
$(true \Rightarrow tx\_end_i) \;\Rightarrow\; (true \Rightarrow \neg tx\_init_j)$
endreq

En las siguientes secciones se realiza el diseño de la arquitectura, sintetizando un proceso sincronizador en el caso de dos estaciones emisoras. La extensión a un número n de estaciones es inmediata. Para ello, se va a sintetizar el proceso sincronizador de manera incremental, especificando por separado los requisitos de sincronización. En el caso de dos estaciones emisoras, existirán dos requisitos de sincronización:

req $\mathcal{I}_{sin_1}$ **is**
$(true \Rightarrow tx\_end_1) \;\Rightarrow\; (true \Rightarrow \neg tx\_init_2)$
endreq

req $\mathcal{I}_{sin_2}$ **is**
$(true \Rightarrow tx\_end_2) \;\Rightarrow\; (true \Rightarrow \neg tx\_init_1)$
endreq

A continuación, se sintetizan dos procesos sincronizadores, uno para cada requisito de sincronización. Finalmente, se obtiene el proceso sincronizador global, a partir de los primeros. Dicho proceso, podría haberse obtenido aplicando directamente el algoritmo de sincronizacion 11.3. Sin embargo, de esta manera se muestra la reutilización de procesos sincronizadores previamente sintetizados.

14.1.6.1 Proceso Sincronizador $\mathcal{Y}_1$

En la figura 14.6 se muestra el grafo del proceso entrelazamiento de dos estaciones emisoras (ver el grafo de la figura 14.5), $\mathcal{S}_{ent} = \mathcal{E}_1 \,|||\, \mathcal{E}_2$. Para simplificar la notación, se le asigna a cada una de las acciones un identificador compuesto por una única letra: $tx\_ready = a$, $busy = b$, $tx\_init = c$, $t\_out\_\sigma = d$, $tx\_end = e$.

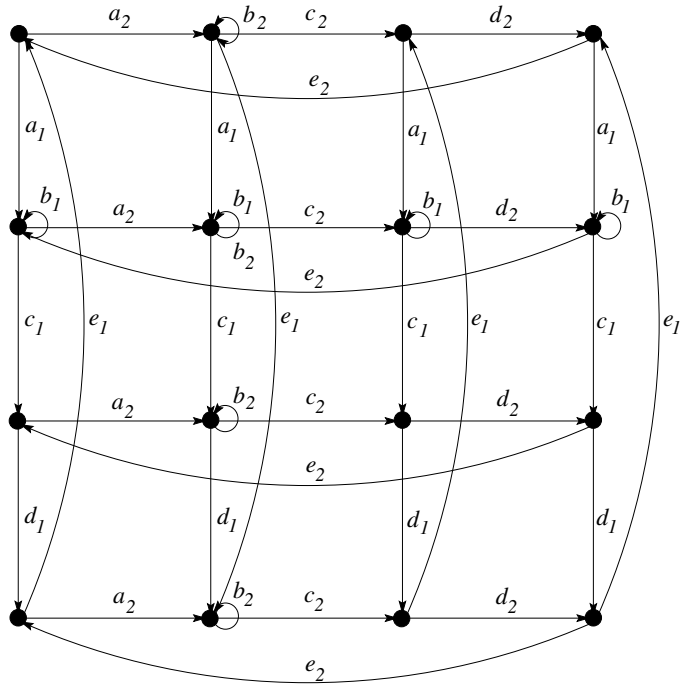


Figura 14.6: Grafo del proceso entrelazamiento $\mathcal{S}_{ent}$.

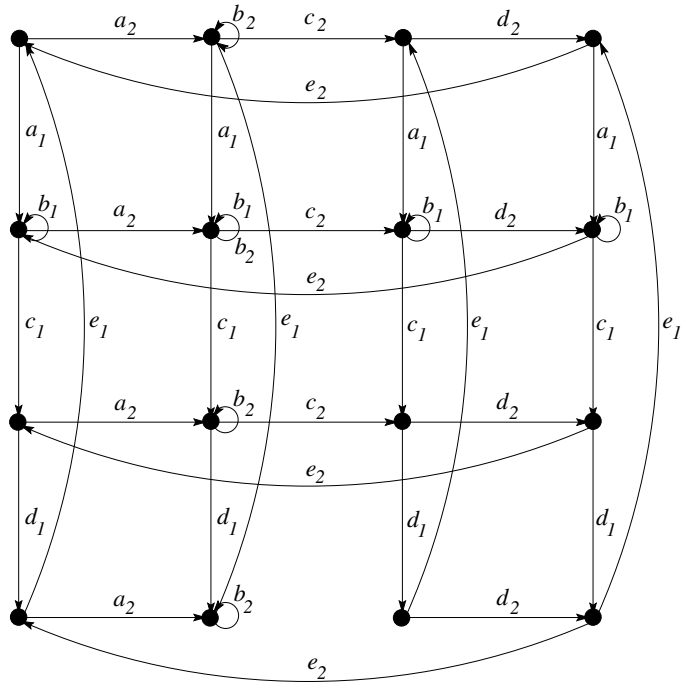


Figura 14.7: Grafo resultante $\mathcal{S}_1$.

La aplicación del requisito de sincronización $\mathcal{I}_{sin_1} = (true \Rightarrow e_1) \;\Rightarrow\; (true \Rightarrow \neg c_2)$ hace que se elimine una rama del proceso entrelazamiento $\mathcal{S}_{ent}$, obteniendo el proceso resultante $\mathcal{S}_1$, tal y como se muestra en la figura 14.7. Mediante el algoritmo de sincronización 11.3 se obtiene el proceso sincronizador $\mathcal{Y}_1$: $\mathcal{S} = \mathcal{S}_{ent} \,|[c_2, d_1, e_1, e_2]|\, \mathcal{Y}_1$, cuyo grafo se muestra en la figura 14.8.

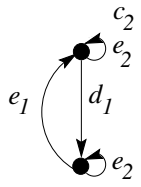


Figura 14.8: Proceso sincronizador $\mathcal{Y}_1$.

14.1.6.2 Proceso Sincronizador $\mathcal{Y}_2$

En la figura 14.9 se muestra el proceso resultante $\mathcal{S}_2$ después de aplicar el requisito de sincronización $\mathcal{I}_{sin_2}$ al proceso entrelazamiento $\mathcal{S}_{ent}$. El algoritmo de sincronización 11.3 obtiene el proceso sincronizador $\mathcal{Y}_2$: $\mathcal{S} = \mathcal{S}_{ent} \,|[c_1, d_2, e_1, e_2]|\, \mathcal{Y}_2$, cuyo grafo se muestra en la figura 14.10.

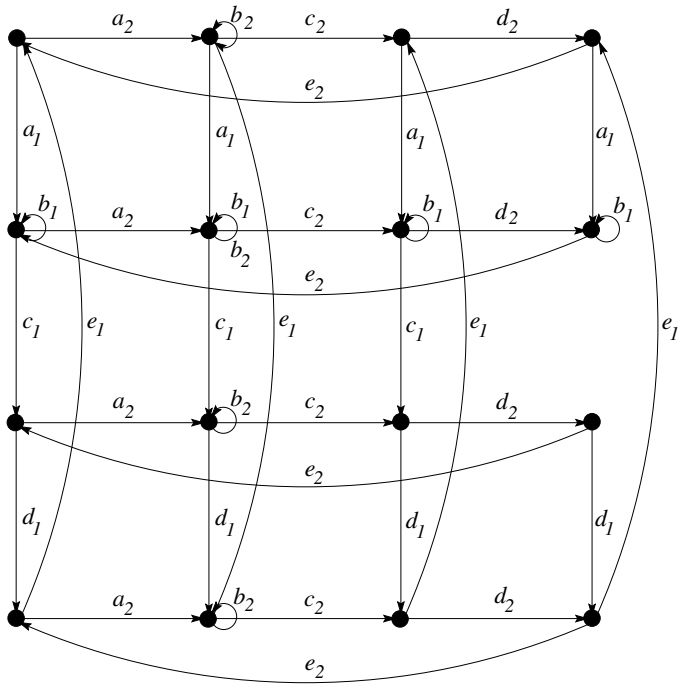


Figura 14.9: Grafo resultante $\mathcal{S}_2$.

14.1.6.3 Proceso Sincronizador Global $\mathcal{Y}_{CSMA}$

Para sintetizar un proceso sincronizador global existen dos opciones: aplicar el algoritmo de sincronización 11.3 al proceso resultante $\mathcal{S}_{CSMA}$ (ver figura 14.11); o reutilizar los procesos

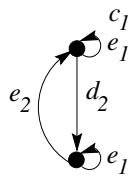


Figura 14.10: Proceso sincronizador $\mathcal{Y}_2$.

sincronizadores $\mathcal{Y}_1$ y $\mathcal{Y}_2$. Aplicando el algoritmo de sincronización 11.3 se obtiene el proceso $\mathcal{Y}_{CSMA}$, cuyo grafo se muestra en la figura 14.12. $\mathcal{S}_{CSMA} = \mathcal{S}_{ent} \; |[c_1, c_2, d_1, d_2, e_1, e_2]| \; \mathcal{Y}_{CSMA}$.

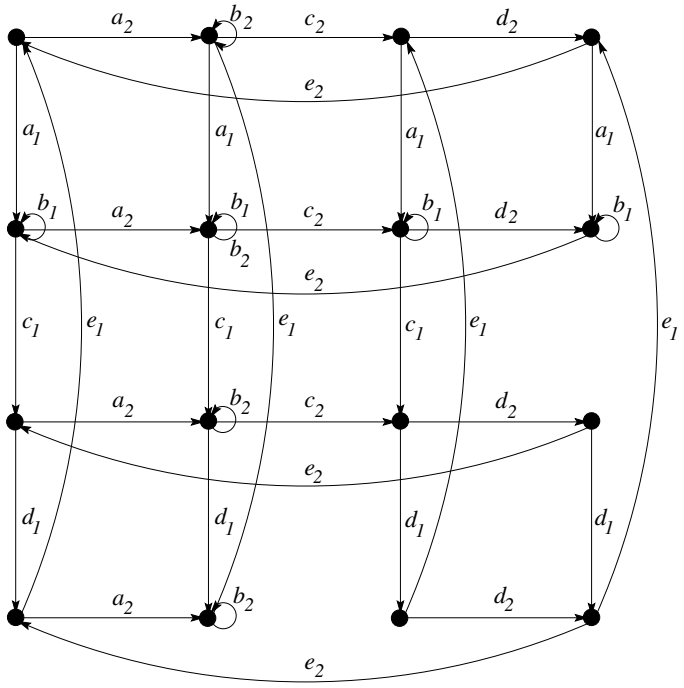


Figura 14.11: Grafo resultante $\mathcal{S}_{CSMA}$.

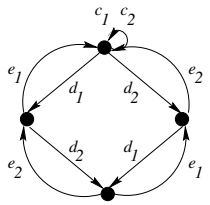
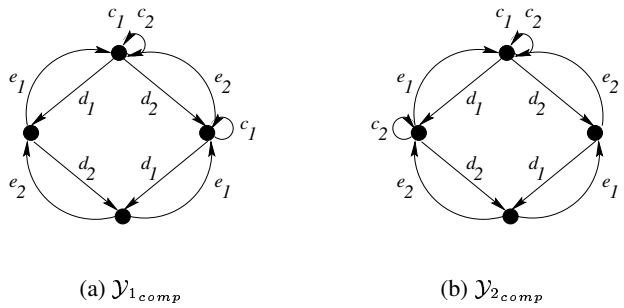


Figura 14.12: Proceso sincronizador $\mathcal{Y}_{CSMA}$.

Para obtener dicho proceso a partir de los sincronizadores parciales $\mathcal{Y}_1$ y $\mathcal{Y}_2$, basta con obtener un grafo de su familia de reducción de estados que sea compatible. Es decir, que contenga la misma unión de estados. La figura 14.13 muestra dichos grafos compatibles.

El sincronizador global $\mathcal{Y}_{CSMA}$ se obtiene sincronizando totalmente ambos procesos, eliminando así, las ramas prohibidas por cada uno de ellos. El sistema final, se puede expresar, por tanto, como sigue: $\mathcal{S}_{CSMA} = (\mathcal{E}_1 \; ||| \; \mathcal{E}_2) \; |[c_1, c_2, d_1, d_2, e_1, e_2]| \; \mathcal{Y}_{CSMA}$.



(a) $\mathcal{Y}_{1_{comp}}$ (b) $\mathcal{Y}_{2_{comp}}$

Figura 14.13: Grafos compatibles de los procesos sincronizadores parciales.

14.1.7 Verificación

La obtención de un sistema con estructura, permite realizar pruebas parciales sobre cada uno de los procesos, y pruebas de integración sobre el sistema resultante. A partir del grafo del sistema final $\mathcal{S}_{CSMA}$, es posible comprobar las propiedades más importantes del protocolo CSMA:

- No se garantiza el medio. El comienzo de transmisión de una estación –el acceso al medio–, no impide que otra estación pueda acceder también a él. $\forall j \neq i$:

req $\mathcal{I}_{ver_{1,i}}$ **is**
$(true \Rightarrow t\_out\_\sigma_i) \;\Rightarrow\; ((true \Rightarrow \neg t\_out\_\sigma_j) \wedge (true \Rightarrow \neg tx\_end_j)$
endreq

  Estas invarianzas no se satisfacen, ya que, las acciones $t\_out\_\sigma_1$ y $t\_out\_\sigma_2$ son posibles en el estado E_{10}[1] y las acciones $t\_out\_\sigma_1$ y tx_end_2 lo son en el estado E_{11}. De manera similar, tx_end_1 y tx_end_2 son posibles en el estado E_{15}, mientras que tx_end_1 y $t\_out\_\sigma_2$ lo son en el estado E_{14}.

- Una vez que una estación comienza la transmisión, ésta la finaliza, independientemente de que la transmisión de otra estación colisione con ella. Para ello, basta con verificar las invarianzas $\mathcal{I}_3$ e $\mathcal{I}_4$ especificadas en el proceso de síntesis.

  Dichas invarianzas se satisfacen en todos los estados de aplicabilidad del grafo $\mathcal{S}_{CSMA}$. La premisa de la invarianza $\mathcal{I}_{3,1}$ se satisface en los estados E_4, E_5 y E_6, mientras que su consecuencia se satisface en sus estados de aplicabilidad: E_8, E_9 y E_{10}. La premisa de la invarianza $\mathcal{I}_{4,1}$ se satisface en los estados E_8, E_9, E_{10} y E_{11}, mientras que su consecuencia se satisface en sus estados de aplicabilidad: E_{12}, E_{13}, E_{14} y E_{15}. Las invarianzas $\mathcal{I}_{3,2}$ y $\mathcal{I}_{4,2}$ se satisfacen de manera similar a las anteriores.

[1] Los estados se han numerado de izquierda a derecha y de arriba hacia abajo, empezando en el estado E_0.

14.2 CSMA/CD

14.2.1 Descripción

La técnica CSMA es claramente ineficiente. Cuando colisionan dos tramas, el medio permanece libre –no se ocupa con transmisiones sin colisión– durante la transmisión de ambas. La capacidad desaprovechada, en comparación con el tiempo de propagación, puede ser considerable para tramas largas. Este desaprovechamiento puede reducirse si una estación continúa escuchando el medio mientras dura la transmisión, lo que conduce a las siguientes reglas para la técnica CSMA/CD:

1. La estación transmite si el medio está libre, si no se aplica la regla 2.
2. Si el medio se encuentra ocupado, la estación continúa escuchando hasta que encuentra libre el canal, en cuyo caso transmite inmediatamente.
3. Si se detecta una colisión durante la transmisión, las estaciones transmiten una señal corta de interferencia para asegurarse de que todas las estaciones constatan la producción de colisión y cesan de transmitir.
4. Después de transmitir la señal de interferencia se espera una cantidad de tiempo aleatorio, tras lo que intenta transmitir de nuevo (iteración desde el paso 1).

Según dichas reglas, se deduce que el tiempo que conlleva la detección de la colisión no es mayor que dos veces el tiempo máximo de propagación σ.

14.2.2 Especificación de Acciones

Se añaden dos nuevos eventos a los identificados para CSMA:

- La detección de una colisión por parte de la estación $\mathcal{E}_i'$: $\mathbf{collision_i}$.
- Según lo expuesto en la sección anterior, para poder detectar la colisión es necesario que el tiempo de transmisión de trama sea superior a dos veces el tiempo de propagación σ. Por tanto, se añade un nuevo evento correspondiente a un temporizador de valor 2σ, que se disparará después de t_out_σ y antes de finalizar la transmisión de la trama: $\boldsymbol{t\_out\_2\sigma}$.

14.2.3 Especificación de Requisitos

El estado inicial del sistema es el mismo, y las dos nuevas acciones no son posibles en dicho estado: $E_0[collision_i] = 0$ y $E_0[t\_out\_2\sigma] = 0$. Las dos primeras invarianzas especificadas para CSMA se mantienen, ya que se refieren a la escucha del medio y al comportamiento de una estación si detecta el medio ocupado. La invarianza $\mathcal{I}_4$ se modifica únicamente en la acción de la premisa, ya que en este caso, para que se pueda finalizar la transmisión es necesario que venza el

temporizador correspondiente a 2σ:

req $\mathcal{I}'_{4,i}$ **is**

$$(true \Rightarrow tx\_out\_2\sigma_i) \Rightarrow \bigcirc\ ((true \Rightarrow tx\_end_i) \wedge (true \Rightarrow \neg tx\_ready_i) \wedge \wedge (true \Rightarrow \neg tx\_init_i) \wedge (true \Rightarrow \neg t\_out\_\sigma_i) \wedge \wedge (true \Rightarrow \neg busy_i))$$

endreq

La invarianza $\mathcal{I}_3$ se modifica en su consecuencia, en la que se le añade como posible el evento $collision$, ya que éste se puede detectar una vez iniciada la transmisión. Además, es necesario añadir una nueva invarianza $\mathcal{I}_6$ similar a la anterior, que permita disparar el temporizador $t\_out\_2\sigma$ una vez disparado el primero. Sólo después de 2σ unidades de tiempo será posible finalizar la transmisión. En ese caso, ya no es posible que se produzca colisión y ésta finaliza, por lo que se mantiene la misma invarianza $\mathcal{I}_5$ especificada para CSMA.

req $\mathcal{I}'_{3,i}$ **is**

$$(true \Rightarrow tx\_init_i) \Rightarrow \bigcirc\ ((true \Rightarrow t\_out\_\sigma_i) \wedge (true \Rightarrow \neg tx\_ready_i) \wedge \wedge (true \Rightarrow \neg tx\_init_i) \wedge (true \Rightarrow \neg tx\_end_i) \wedge \wedge (true \Rightarrow \neg busy_i) \wedge (true \Rightarrow collision_i))$$

endreq

req $\mathcal{I}_{6,i}$ **is**

$$(true \Rightarrow t\_out\_\sigma_i) \Rightarrow \bigcirc\ ((true \Rightarrow t\_out\_2\sigma_i) \wedge (true \Rightarrow \neg tx\_ready_i) \wedge \wedge (true \Rightarrow \neg tx\_init_i) \wedge (true \Rightarrow \neg tx\_end_i) \wedge \wedge (true \Rightarrow \neg busy_i) \wedge (true \Rightarrow collision_i))$$

endreq

14.2.4 Traducción SCTL-MUS

Las nuevas invarianzas comparten los mismos MetaGrafos que las invarianzas especificadas para CSMA, por lo que se reutiliza el proceso de traducción realizado en el caso anterior.

14.2.5 Síntesis Incremental

Del proceso de síntesis realizado para CSMA, se reutiliza la síntesis parcial de las invarianzas $\mathcal{I}_1, \mathcal{I}_2$ e $\mathcal{I}_5$, ya que éstas se mantienen. La figura 14.14 muestra el grafo correspondiente a la síntesis de estas tres invarianzas.

La síntesis de las invarianzas $\mathcal{I}'_3, \mathcal{I}'_4$ e $\mathcal{I}_6$ mantienen el mismo MetaGrafo que $\mathcal{I}_3, \mathcal{I}_4$ e $\mathcal{I}_5$ salvo la nueva acción $collision$. Esto supone que los MetaGrafos compatibles coincidan con los de

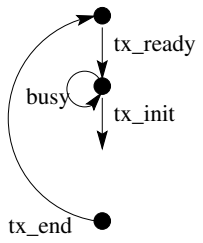
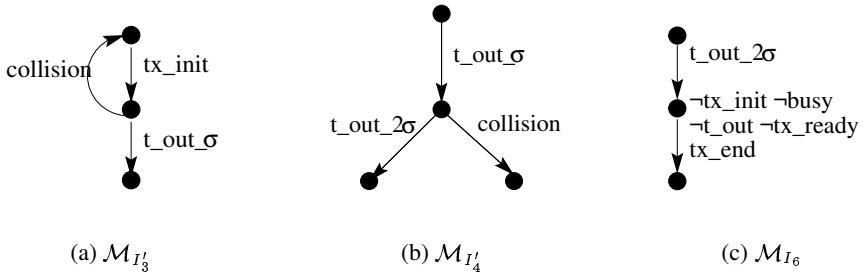


Figura 14.14: Reutilización del proceso de síntesis.

antes salvo en dicha acción. La figura 14.15 muestra los grafos MUS de las nuevas invarianzas especificadas que han sido utilizados como primera elección por el algoritmo de síntesis.



(a) $\mathcal{M}_{I'_3}$ (b) $\mathcal{M}_{I'_4}$ (c) $\mathcal{M}_{I_6}$

Figura 14.15: CSMA/CD: Grafos utilizados por el algoritmo de síntesis.

Finalmente, la figura 14.16 muestra el grafo MUS de una estación emisora según el protocolo CSMA/CD.

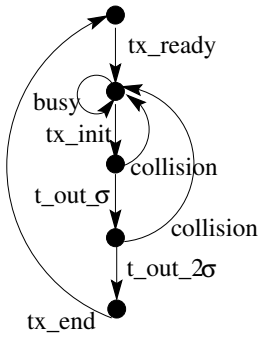


Figura 14.16: CSMA/CD: Grafo final de una estación emisora $\mathcal{E}'$.

14.2.6 Diseño de la Arquitectura

Los requisitos de sincronización para CSMA/CD contienen los vistos anteriormente para CSMA, ya que estos se referían a la escucha y detección del medio ocupado, que se mantiene para CSMA/CD.

Sin embargo, la invarianza $\mathcal{I}_{sin_i}$ se modifica, ya que, el evento que ahora indica que ya se lleva transmitiendo σ unidades de tiempo es $t\_out\_2\sigma$ y no el fin de la transmisión tx_end como sucedía

en CSMA. Además, es necesario añadir un nuevo requisito de sincronización que permita expresar la detección de la colisión:

- Una vez que una estación lleva transmitiendo 2σ unidades de tiempo, el resto de estaciones detectan dicha transmisión. Es decir, abandonan la transmisión o no la inician, $\forall j \neq i$:

req $\mathcal{I}_{sin_{1,i}}$ **is**
$(true \Rightarrow t\_out\_2\sigma_i) \;\Rightarrow\; (true \Rightarrow \neg tx\_init_j)$
endreq

req $\mathcal{I}_{sin_{2,i}}$ **is**
$(true \Rightarrow tx\_end_i) \;\Rightarrow\; ((true \Rightarrow \neg tx\_init_j) \wedge (true \Rightarrow \neg t\_out\_\sigma_j) \wedge$
$\wedge\, (true \Rightarrow \neg t\_out\_2\sigma_j))$
endreq

En la figura 14.17 se muestra el grafo del proceso entrelazamiento de dos estaciones emisoras (ver el grafo de la figura 14.16), $\mathcal{S}'_{ent} = \mathcal{E}'_1 \,|||\, \mathcal{E}'_2$. Los nuevos eventos –$t\_out\_2\sigma$ y $collision$– se identifican mediante las letras f y g respectivamente.

La aplicación de los requisitos de sincronización elimina 8 ramas y un estado del proceso entrelazamiento $\mathcal{S}'_{ent}$, obteniendo el proceso resultante $\mathcal{S}_{CSMA/CD}$, tal y como se muestra en la figura 14.18.

Mediante el algoritmo de sincronización 11.3 se obtiene el proceso sincronizador $\mathcal{Y}_{CSMA/CD}$: $\mathcal{S}_{CSMA/CD} = \mathcal{S}'_{ent} \,|[c_1, c_2, d_1, d_2, e_1, e_2, f_1, f_2, g_1, g_2]|\, \mathcal{Y}_{CSMA/CD}$, cuyo grafo se muestra en la figura 14.19.

14.2.7 Verificación

La detección de la colisión permite mejorar notablemente la eficiencia del protocolo, ya que las transmisiones que finalizan lo hacen con seguridad de no haber colisionado con ninguna otra. Es decir, una vez que una estación lleva transmitiendo 2σ unidades de tiempo, tiene garantizado el medio.

Esta característica puede ser comprobada sobre el sistema obtenido $\mathcal{S}_{CSMA/CD}$ mediante la verificación del siguiente requisito. $\forall j \neq i$:

req $\mathcal{I}'_{ver_{1,i}}$ **is**
$(true \Rightarrow tx\_end_i) \;\Rightarrow\; (true \Rightarrow \neg tx\_end_j)$
endreq

Dichas invarianzas se satisfacen en el sistema obtenido, ya que no existe ningún estado en el que las acciones tx_end_1 (e_1) y $tx\_end_2)$ (e_2) sean posibles.

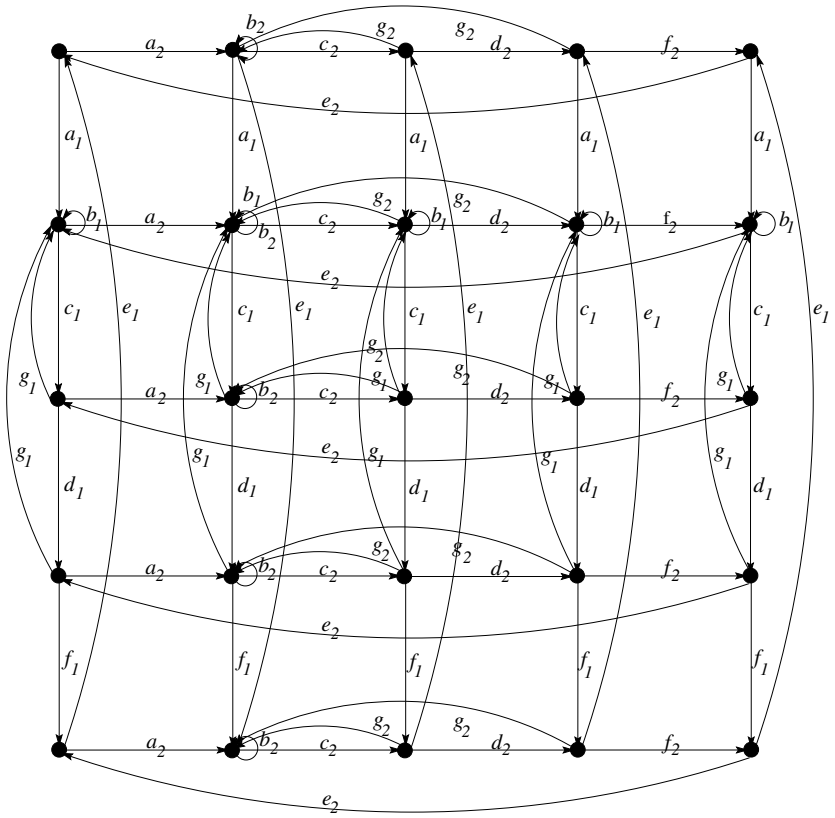


Figura 14.17: Grafo del proceso entrelazamiento $\mathcal{S}'_{ent}$.

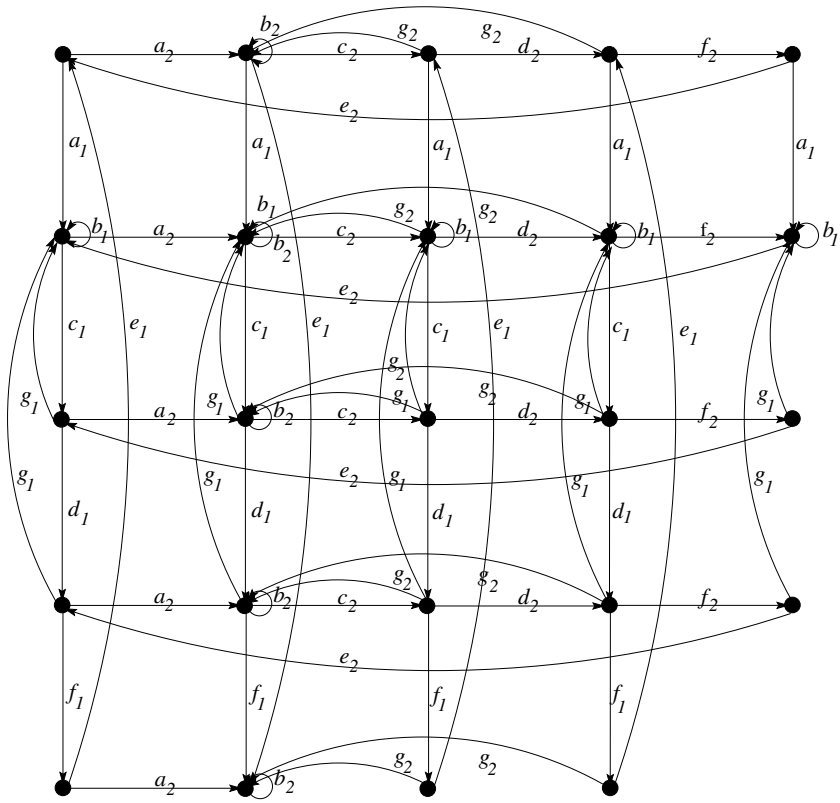


Figura 14.18: Grafo resultante $\mathcal{S}_{CSMA/CD}$.

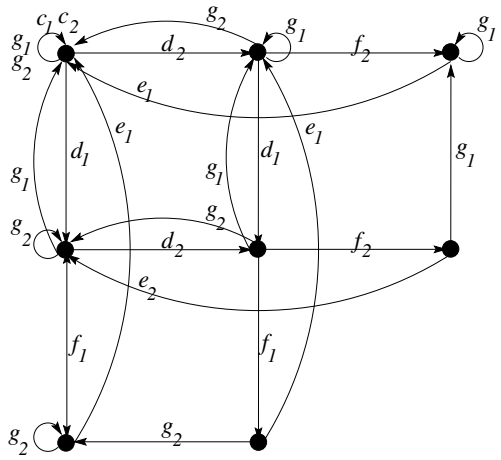


Figura 14.19: Proceso sincronizador $\mathcal{Y}_{CSMA/CD}$.

Sin embargo, el sistema sintetizado no satisface los objetivos de CSMA/CD, ya que las transmisiones en las que se produce colisión no abortan, sino que continúan y finalizan. Esto se comprueba mediante la verificación del siguiente requisito:

- Si dos estaciones están transmitiendo, ambas deben detectar la colisión y abortar dicha transmisión. Es decir, a partir de que dos estaciones inician una transmisión, ambas deben ser abortadas. Esto no se cumple en el sistema sintetizado $\mathcal{S}_{CSMA/CD}$, ya que, no se satisface la siguiente invarianza:

req $\mathcal{I}_{ver_2}$ **is**

$$((true \Rightarrow t\_out\_2\sigma_1) \wedge (true \Rightarrow t\_out\_2\sigma_2)) \Rightarrow (((true \Rightarrow t\_out\_2\sigma_1) \Rightarrow \bigcirc (true \Rightarrow \neg tx\_end_1)) \wedge ((true \Rightarrow t\_out\_2\sigma_2) \Rightarrow \bigcirc (true \Rightarrow \neg tx\_end_2))$$

endreq

En el estado E_{18} se satisface su premisa, pero los estados de aplicabilidad E_{19} y E_{23} incumplen su consecuencia. Esto quiere decir que sería posible concluir ambas transmisiones.

Para evitar esto, se deben introducir nuevos requisitos de sincronización, según el razonamiento que se expone a continuación. Si una estación puede llegar a transmitir durante 2σ unidades de tiempo, tiene garantizado el medio, ya que el resto deben haber detectado previamente el medio ocupado. Dicha colisión debe ser detectada por las estaciones una vez iniciada la transmisión –la inician sólo si el medio no está ocupado–, de manera que si dos o más estaciones inician la transmisión con una diferencia de tiempo menor que σ –tiempo necesario para que se escuchara el medio como ocupado–, ambas deben abortar la transmisión. Esto supone que ninguna puede llegar a transmitir 2σ unidades de tiempo.

- Si dos estaciones emisoras han iniciado la transmisión –puede suceder el evento t_out_σ–, ninguna podrá finalizar la transmisión –en ninguna se podrá producir el evento $t\_out\_2\sigma$–. $\forall j \neq i$:

req $\mathcal{I}_{sin_{3,i}}$ **is**
$((true \Rightarrow t\_out\_\sigma_i) \wedge (true \Rightarrow t\_out\_\sigma_j)) \Rightarrow \bigcirc ((true \Rightarrow \neg t\_out\_2\sigma_i)$
$\wedge (true \Rightarrow \neg t\_out\_2\sigma_j))$
endreq

- Si una estación inició la transmisión, las demás no podrán finalizar la misma, ya que antes detectarán la colisión: $\forall j \neq i$:

req $\mathcal{I}_{sin_{4,i}}$ **is**
$(true \Rightarrow t\_out\_\sigma_i) \Rightarrow (true \Rightarrow \neg t\_out\_2\sigma_j)$
endreq

La aplicación de estos requisitos sobre el proceso entrelazamiento $\mathcal{S}'_{ent}$ mostrado en la figura 14.17 hace que se eliminen 12 nuevas ramas y cuatro estados. La figura 14.20 muestra el nuevo proceso resultante $\mathcal{S}'_{CSMA/CD}$.

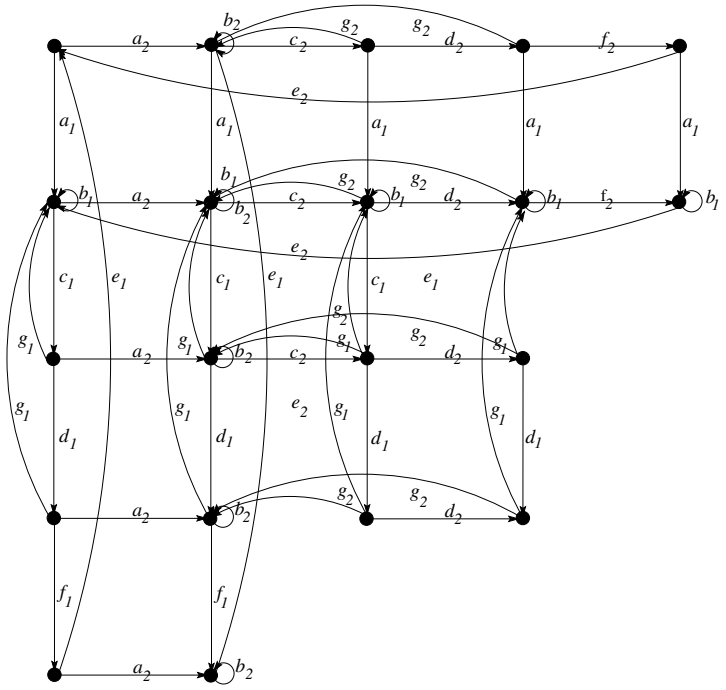


Figura 14.20: Grafo resultante $\mathcal{S}'_{CSMA/CD}$.

Para la obtención del nuevo proceso sincronizador $\mathcal{Y}'_{CSMA/CD}$ (ver figura 14.21), basta con eliminar dichas ramas y estados del proceso sincronizador anterior, ya que no supone ningún cambio en la reducción de estados –al contener dicho proceso todas las ramas y estados eliminados–.

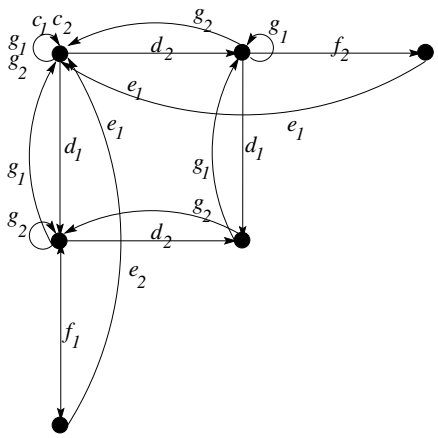


Figura 14.21: Proceso sincronizador $\mathcal{Y}'_{CSMA/CD}$.

Parte VII

Conclusiones

Capítulo 15

Conclusiones y Líneas de Trabajo Futuras

15.1 Conclusiones

El trabajo desarrollado en la presente tesis se basa en la definición e integración de una metodología formal en un proceso incremental de desarrollo software. Para la consecución de dicho objetivo se han realizado esfuerzos en tres direcciones: definición de los formalismos adecuados; integración en una metodología estructurada; y diseño e implementación de algoritmos que faciliten las tareas de cada una de las fases de desarrollo software abordadas. A continuación, se resumen las principales aportaciones de esta tesis, agrupándolas según la clasificación anterior.

15.1.1 Definición de SCTL y MUS

Dentro de los objetivos de esta tesis, se planteó la definición de nuevos formalismos, creados a medida para la metodología desarrollada. La razón fundamental para dicha elección, en vez de optar por otros formalismos ya existentes, se basa en la formalización del concepto de subespecificación. La subespecificación juega un papel muy importante en el trabajo realizado. Surge como un enlace entre el usuario y la especificación formal, ya que las especificaciones informales son inherentemente incompletas. La utilización de métodos formales en las primeras fases de desarrollo software muestra claras ventajas. Sin embargo, estas ventajas no son aprovechadas debido, fundamentalmente, al carácter inherentemente informal de dichas fases iniciales. La introducción de la subespecificación en la expresividad de los formalismos definidos pretende reducir esa distancia, formalizando la parte informal del proceso de especificación.

La subespecificación permite ver el sistema especificado como un conjunto de piezas que pueden evolucionar (partes subespecificadas). Dicha evolución puede realizarse en dos sentidos, hacia partes del sistema desarrollado, o hacia piezas que no forman parte del mismo. Inicialmente, el sistema se puede concebir como un sistema totalmente subespecificado, al que se le pueden

añadir piezas o al que se le pueden restringir las mismas.

Una vez planteada la necesidad de definir un nuevo formalismo que introduzca en su expresividad el concepto de subespecificación, se definen dos formalismos con diferentes objetivos, ya que, según se concluye en el capítulo 1, las características deseables de las técnicas formales dependen de la fase del ciclo de vida en la que éstas se apliquen; siendo más adecuadas para las primeras fases las técnicas orientadas a propiedades, mientras que las técnicas constructivas se adecuan mejor a las fases de diseño. Los formalismos definidos son:

- Una lógica temporal denominada SCTL para la fase inicial de captura y especificación de requisitos. Su principal objetivo es proporcionar una interfaz sencilla entre el usuario y la metodología formal desarrollada. Por ello, se define una semántica para las proposiciones SCTL basada en el lenguaje causal, donde una premisa causa una consecuencia en un momento determinado por un operador temporal: "*si es posible una premisa, entonces A La Vez (Antes o Después) debe ser posible una consecuencia*".

  Una de las principales características de SCTL, en la que se basan la mayoría de los algoritmos desarrollados en este trabajo, reside en la definición jerárquica y recursiva de los requisitos SCTL. Para ello, se definen requisitos atómicos, a partir de los cuales, se construyen los requisitos más complejos. Esto permite, además, definir estructuras abstractas como los MetaRequisitos, almacenando de una manera eficiente la información relativa a los requisitos SCTL para la fase de mantenimiento.

  La lógica definida pretende sentar las bases de un formalismo orientado a propiedades para las primeras fases del ciclo de vida. Por ello, su definición es muy flexible, permitiendo enriquecer su expresividad mediante nuevos operadores temporales, o nuevos tipos de requisitos –finalidades, invarianzas–.

- Un modelo de estados MUS que representa el comportamiento de los requisitos SCTL y de los sistemas especificados. La definición de los grafos MUS permite sintetizar de manera automática prototipos que satisfacen un conjunto de requisitos SCTL especificados.

  Se define un proceso incremental en el que el sistema especificado se enriquece mediante la especificación de nuevos requisitos. Dicho proceso se realiza de manera automática, reutilizando la síntesis común de cada requisito y decidiendo los grados de libertad de cada síntesis. Este proceso se formaliza definiendo incrementos de requisito, incrementos de grafo, decisiones de grafo, decisiones de estado y versiones de procesos; lo que permite mantener, a nivel de especificación SCTL-MUS, los sistemas desarrollados.

  Los grafos MUS permiten expresar el concepto de subespecificación, extendiendo el soportado por la lógica SCTL. Para ello, se define un estado de subespecificación y una acción de subespecificación. Ambos permiten especificar parcialmente los elementos del grafo, así como obtener, de manera directa, información relevante del sistema modelado.

  El estado de subespecificación permite: especificar un evento posible sin tener que especificar el estado siguiente a dicho evento; especificar arcos potenciales –estados asociados

a cada evento sin haber especificado que dicho evento es posible–; y obtener el resumen del grafo subespecificado correspondiente a cada una de las acciones o eventos del sistema, mostrando de una manera directa si se trata de eventos posibles o prohibidos. Esta información es relevante para realizar de manera eficiente la fase de verificación.

La acción de subespecificación permite especificar un arco posible entre dos estados sin tener que especificar la acción a través de la que se produce dicha evolución. Además, la matriz de adyacencia correspondiente a dicha acción muestra el comportamiento genérico del sistema, resumiendo las evoluciones existentes entre los estados del mismo (los arcos), sin especificar a través de qué acción o evento se produce cada una de las evoluciones.

15.1.2 Integración en una Metodología Formal de Desarrollo Software

Un aspecto importante en la metodología desarrollada está muy ligado a la manera de concebir los sistemas diseñados –añadiendo o restringiendo piezas–, y se basa en mantener esta flexibilidad durante todo el proceso de desarrollo software. Por ello, se opta por un proceso de desarrollo iterativo con prototipado, en el que se define un proceso de síntesis incremental.

En la primera fase de especificación de requisitos, MUS se utiliza como otra perspectiva de los requisitos SCTL especificados, observando el comportamiento de estos mediante un grafo o modelo de estados. Para ello, se define un algoritmo de traducción entre requisitos SCTL y grafos MUS, lo que permite, además, realizar tareas de verificación parciales sobre los requisitos especificados. La fase de especificación, por tanto, interactúa con el usuario a dos niveles: a través de la lógica SCTL, especificando requisitos; y a través de los grafos MUS, observando y validando el comportamiento de los requisitos especificados.

En una segunda etapa, MUS permite sintetizar prototipos que satisfacen un conjunto de requisitos SCTL. Dichos prototipos pueden animarse, refinarse mediante la especificación de nuevos requisitos, así como pasar una serie de pruebas en la que se verifican requisitos SCTL y se valida el prototipo sintetizado. Para ello, es necesario definir un algoritmo de síntesis y un algoritmo de verificación. Tanto la fase de síntesis como la fase de verificación se abordan combinando la lógica SCTL y los grafos MUS. Sin embargo, se estudia y se presenta una alternativa independiente de la lógica utilizada, basada únicamente en los grafos MUS. Esto proporciona una gran flexibilidad a la metodología propuesta ante la definición de nuevos formalismos o modificaciones de SCTL.

Finalmente, la especificación SCTL-MUS del sistema diseñado se traduce a la técnica formal constructiva E-LOTOS, Para ello, se desarrolla un algoritmo de traducción entre MUS y dicha técnica.

Este ciclo se repite para cada uno de los procesos componentes (partes funcionales del sistema), incluyendo la parte de sincronización en una última etapa, realizando un diseño de la arquitectura en el que se da estructura al sistema especificado. En este sentido, la metodología SCTL-MUS, con el objetivo de uniformar el proceso de diseño, propone la síntesis de un proceso sincronizador a partir de un conjunto de requisitos de sincronización especificados en SCTL. Esto supone la definición de un nuevo algoritmo de sincronización, así como la elección de los

operadores arquitectónicos para expresar la estructura del sistema. Con el objetivo de integrar la arquitectura inicial obtenida en el entorno transformacional LIRA se utilizan los operadores arquitectónicos de E-LOTOS.

La metodología presentada aborda la fase de mantenimiento como una nueva iteración en el proceso de desarrollo software. Por tanto, puede suponer la alteración de los requisitos especificados, la síntesis de nuevos prototipos y/o la verificación de nuevas propiedades. Por ello, basta con identificar y definir las estructuras de datos para soportar dicho proceso basado en una síntesis incremental. En este aspecto, los mecanismos de almacenamiento definidos permiten la reutilización del proceso de síntesis entre sistemas con requisitos comunes. Dicha reutilización puede verse como una capacidad de aprendizaje del sistema.

15.1.3 Algoritmos Desarrollados e Implementación

El primer algoritmo desarrollado es el de traducción SCTL-MUS, que inicialmente se basa en la traducción de los requisitos atómicos. El diseño de un algoritmo de síntesis incremental se aborda en varias etapas. En primer lugar, se adapta el algoritmo de traducción obtenido, transformándolo en un algoritmo de síntesis. Dicho algoritmo de refina en busca de reutilización y eficiencia, obteniendo finalmente un algoritmo independiente de SCTL, haciendo así más flexible la metodología propuesta.

La reutilización en el proceso de síntesis se basa en reutilizar el proceso común de la síntesis de un requisito en distintos sistemas. Para ello, se define un algoritmo de reducción de estados, que permite obtener familias de grafos MUS que representan un mismo requisito SCTL. La eficiencia se consigue almacenando en los grafos MUS la información necesaria para no realizar síntesis de requisitos previamente sintetizados –debido a la aparición de nuevos estados de aplicabilidad de dicho requisito–. Finalmente, la independencia de SCTL se obtiene definiendo un algoritmo de síntesis basado en el solapamiento de dos grafos MUS: el correspondiente al sistema, y el correspondiente al requisito especificado. La obtención de este último algoritmo sugiere la revisión del algoritmo de traducción SCTL-MUS, en el que no se reutiliza la traducción de los requisitos más simples. El algoritmo de solapamiento definido permite obtener un algoritmo de traducción en el que se reutiliza totalmente la traducción de los requisitos SCTL realizada, ya que ésta se obtiene a partir del grafo MUS correspondiente a los subrequisitos de uno dado.

Por otra parte, la combinación de SCTL y MUS permite abordar la fase de verificación o prueba desde dos perspectivas: mediante la demostración de teoremas, a partir del álgebra de IPM como base matemática; y el *model checking*, mediante la definición de una relación de satisfacción de requisitos SCTL en los grafos MUS. La principal aportación en este aspecto reside en la introducción del término de subespecificación, enriqueciendo los grados de satisfacción clásicos y aportando información detallada al usuario, con el objetivo de establecer y asentar los requisitos iniciales del sistema especificado. La formalización del conjunto de grados de satisfacción, dotándole de una estructura de álgebra de De Morgan, así como la definición de una relación de equivalencia, sienta las bases necesarias para abordar la verificación mediante demostradores de

teoremas. Sin embargo, en este trabajo se opta por una técnica de *model checking*, lo que permite combinar las características de los dos formalismos definidos SCTL y MUS, reduciendo las verificaciones recursivamente hasta evaluar requisitos atómicos.

El último algoritmo desarrollado es un algoritmo de traducción entre especificaciones SCTL-MUS y E-LOTOS. E-LOTOS se utiliza en la fase de refinamiento, que no se aborda en este trabajo. Sin embargo, se utilizan los grafos MUS como paso intermedio entre una especificación orientada a propiedades y una especificación constructiva. Para ello, se define un algoritmo de traducción de MUS a E-LOTOS, así como un algoritmo que permite expresar el sistema especificado como un conjunto de procesos sincronizados por un proceso sincronizador. Dicho algoritmo permite, además, la reutilización de sincronizadores parciales, y está basado en el mismo algoritmo de reducción de estados utilizado para obtener las familias de grafos MUS de un requisito SCTL.

La fase de mantenimiento también se basa en la combinación de los formalismos SCTL y MUS. En este sentido, la principal aportación reside en la formalización de las decisiones de diseño, lo que permite, de manera automática, optar por una u otra decisión hasta obtener (si es posible) un grafo MUS consistente con los requisitos SCTL especificados. Dichas decisiones se basan en la elección de uno u otro grafo MUS representante del requisito SCTL especificado. Además, se busca la reutilización de la información almacenada, definiendo estructuras abstractas como los MetaRequisitos y los MetaGrafos.

La totalidad de algoritmos desarrollados, así como las estructuras necesarias para la fase de mantenimiento se implementan en una arquitectura distribuida en la que se distinguen tres tipos de servidores: servidores de algoritmos, servidores intermedios y servidores de acceso directo a la base de datos. Los clientes se estructuran en tres partes fundamentales: una interfaz gráfica genérica y de administración, una vista de la base de datos mediante diferentes tipos de representación de los datos almacenados, y una interfaz de acceso a los algoritmos implementados.

15.2 Trabajo Futuro

Las principales líneas de trabajo futuro, identificadas durante el desarrollo de esta tesis, se pueden clasificar en dos grandes grupos:

1. Por una parte, se propone profundizar en aspectos de interés que han surgido en el desarrollo del presente trabajo:

    - La representación matricial de los grafos MUS, así como el diseño de algoritmos que dependen únicamente de MUS, sugiere la vectorización de dichos algoritmos, lo que permitiría su ejecución en paralelo. En este sentido, se apunta trabajo en dos direcciones: en primer lugar, hacia el estudio de la complejidad de los algoritmos propuestos, así como las mejoras posibles si dichos algoritmos se paralelizan; en segundo lugar, hacia el estudio de los grafos MUS, evaluando la necesidad de incluir mayor información en su representación matricial para facilitar la paralelización de los algoritmos

desarrollados.

En este último aspecto, ya se ha incluido información redundante mediante la acción y el estado de subespecificación. La primera permite obtener rápidamente los estados de aplicabilidad de los requisitos SCTL, mientras que el estado de subespecificación permite obtener de manera directa el valor de cada acción en todos los estados del grafo.

- Durante todo el desarrollo de este trabajo se ha puesto especial hincapié en la reutilización de especificaciones. Dicha reutilización se sustenta sobre la definición jerárquica de los requisitos SCTL y se extiende al proceso de síntesis, creando familias de grafos en los que se sintetiza la parte común a todos los sistemas en los que se especifica el requisito que representan. Sin embargo, no se aborda la reutilización como un objetivo general, sino que se trata de manera particular en cada una de las fases definidas.

  En este sentido, se propone plantear la reutilización desde un punto de vista de comportamiento, que se representa mediante los grafos MUS. Es decir, reutilizar partes de la especificación que tengan un comportamiento común. Para ello, es necesario definir una métrica y un método que permita decidir si dos comportamientos –dos grafos– son semejantes, cuantificando dicha semejanza. La experiencia del trabajo desarrollado sugiere identificar filtros o cajas negras que ante la entrada de dos comportamientos semejantes propaguen hacia su salida una información común. Un ejemplo interesante es utilizar el algoritmo de reducción de estados como filtro, ya que todos los grafos que obtiene representan el mismo requisito SCTL, y además, forman una familia de procesos sincronizadores. Otro filtro propuesto es el algoritmo de verificación, para lo cual es necesario identificar un conjunto de patrones –grafos MUS de sistemas genéricos– sobre los que evaluar los comportamientos que se quieren reutilizar.

- En cuanto al proceso de verificación, en la metodología propuesta se opta por una técnica de *model checking*, combinando los dos formalismos definidos. Una línea de trabajo futuro clara es combinar dicho método con demostradores de teoremas, partiendo de la estructura de álgebra de De Morgan del conjunto de grados de satisfacción. Parece apropiado combinar ambos métodos en las verificaciones parciales. Dado que en los grafos MUS se almacena el resultado de verificación de los requisitos a nivel de estado, es posible aplicar demostradores de teoremas a dicho nivel. En este caso, es necesario definir relaciones de equivalencia a nivel de estado, de manera que dos requisitos puedan ser equivalentes en un estado y en otros no, ya que el comportamiento de los mismos se particulariza (sintetizando un grafo MUS concreto) en cada estado del sistema.

- Finalmente, se propone avanzar en la implementación diseñada. En la actualidad, se dispone de una versión con un único servidor de algoritmos y un servidor intermedio. El trabajo futuro constará, por tanto, en distribuir dichos servidores tal y como se expuso en el capítulo 13. Por otra parte, parece interesante el diseño distribuido de la base de datos, en la que cada usuario pudiera mantener, en servidores diferentes, la

información propia de cada uno de ellos. En ese caso, sería interesante que la base de datos implementara replicación de la estructura de MetaRequisitos y MetaGrafos, ya que ésta se comparte para la totalidad de usuarios.

2. Por otra parte, se propone ampliar la metodología definida, aumentando los aspectos modelados por los formalismos definidos, definiendo nuevos niveles de abstracción y profundizando en su integración con el entorno LIRA:

   - Los formalismos definidos permiten especificar el ordenamiento temporal de los eventos del sistema. Sin embargo, no permiten modelar el intercambio de información (datos), siendo ésta una línea clara de avance. Para ello, se propone recuperar el trabajo desarrollado por A. Gil [GS99], estudiando el impacto de la subespecificación en la integración de ambos trabajos.

     Además, el presente trabajo se limita a los requisitos funcionales. El tratamiento de requisitos no funcionales enriquecería notablemente la metodología desarrollada. Un primer paso sería estudiar la incorporación de requisitos de tiempo, para lo que se propone utilizar la semántica definida en E-LOTOS.

   - El punto de partida en la metodología propuesta es una especificación de requisitos SCTL. Sin embargo, el proceso de adquisición de requisitos es mucho más complejo. Se propone avanzar en el modelo de desarrollo propuesto, aumentado el grado de abstracción inicial. Un único esquema de representación –SCTL-MUS– puede ser complementado con el fin de mostrar y documentar aspectos no soportados por los formalismos SCTL y MUS. La combinación de varios esquemas de representación, en los que se mezclan notaciones formales con informales, requisitos funcionales y no funcionales, así como varios puntos de vista del sistema (*viewpoints*) permite obtener documentos completos de especificación de requisitos.

     La inclusión de nuevas notaciones informales –en lenguaje natural–, así como la inclusión de puntos de vista sugiere la definición de nuevos esquemas de representación que actúen como paso intermedio entre los usuarios y la metodología desarrollada. En este sentido se propone una alternativa para incorporar tanto el intercambio de datos como los requisitos de tiempo. Dicha alternativa se basa en modelar cada uno de estos aspectos con un nuevo modelo de representación del sistema (punto de vista) aislando el comportamiento funcional, el intercambio de datos y las restricciones temporales. Sin embargo, es necesario realizar un estudio exhaustivo sobre esta posibilidad para determinar si la complejidad de esta alternativa compensa sus ventajas.

   - En la metodología propuesta se distinguen tres grandes etapas de desarrollo: requisitos iniciales, refinamiento y mantenimiento. La segunda fase no se aborda en este trabajo, ya que forma parte del desarrollo de un trabajo de investigación previo que concluyó con el desarrollo del entorno transformacional LIRA. La metodología SCTL-MUS proporciona una arquitectura inicial del sistema como entrada de LIRA, que la refina mediante transformaciones E-LOTOS.

Las transformaciones E-LOTOS realizadas en el entorno LIRA se realizan a partir de una arquitectura inicial y permiten añadir un mayor grado de detalle a la especificación hasta que ésta pueda ser traducida a un lenguaje de programación de alto nivel. Estas transformaciones, con menor nivel de abstracción, no deben incorporar cambios funcionales en la especificación SCTL-MUS, pero si en su arquitectura. La traducción de dichos cambios de estructura a una especificación SCTL-MUS permitiría completar el modelo de desarrollo propuesto, limitando las transformaciones a las soportadas por la expresividad de SCTL-MUS, o incorporando nuevos operadores arquitectónicos.

Parte VIII

Apéndices

Apéndice A

Ejemplo del Algoritmo de Reducción de Estados

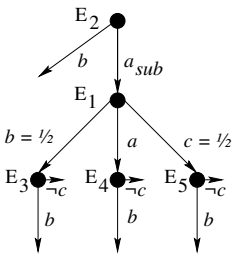
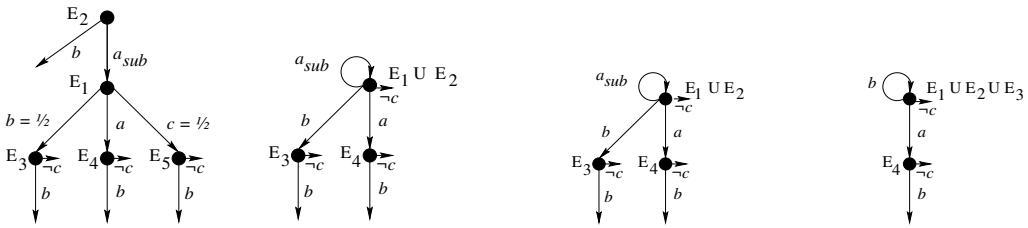


Figura A.1: Grafo MUS $\mathcal{M}_1^{\mathcal{R}_{red}}$

req $\mathcal{R}_{red}$ is

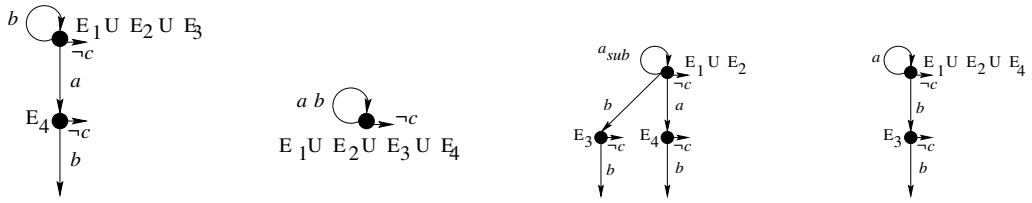
$$(true \Rightarrow \odot\, b) \Rightarrow \bigcirc\, (true \Rightarrow b) \Rightarrow ((true \Rightarrow \neg c) \wedge (true \Rightarrow \odot\, a))$$

endreq



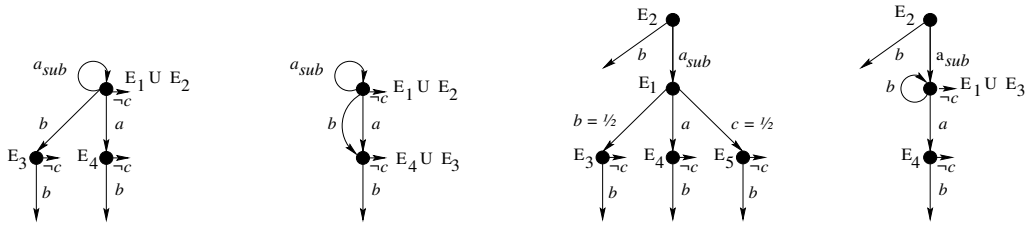
(a) $\mathcal{M}_2^{\mathcal{R}_{red}}$ (b) $\mathcal{M}_3^{\mathcal{R}_{red}}$

Figura A.2: Grafos $\mathcal{M}_2^{\mathcal{R}_{red}}$ y $\mathcal{M}_3^{\mathcal{R}_{red}}$



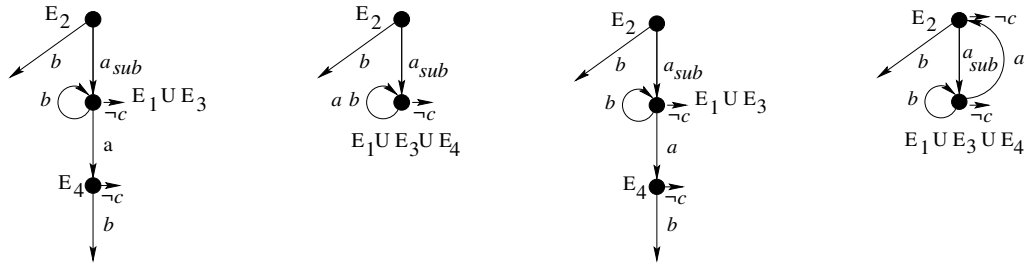
(a) $\mathcal{M}_4^{\mathcal{R}_{red}}$ (b) $\mathcal{M}_5^{\mathcal{R}_{red}}$

Figura A.3: Grafos MUS $\mathcal{M}_4^{\mathcal{R}_{red}}$ y $\mathcal{M}_5^{\mathcal{R}_{red}}$



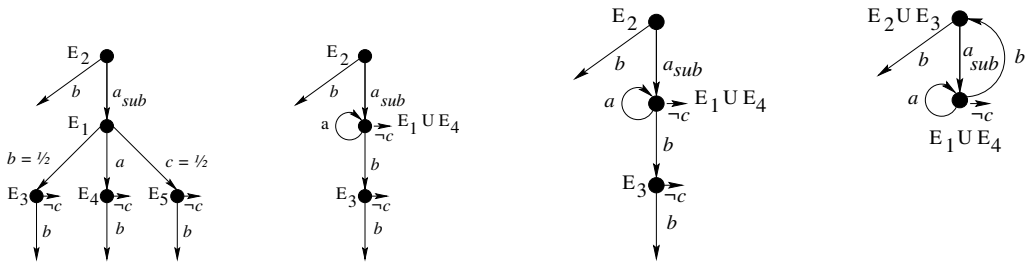
(a) $\mathcal{M}_6^{\mathcal{R}_{red}}$ (b) $\mathcal{M}_7^{\mathcal{R}_{red}}$

Figura A.4: Grafos MUS $\mathcal{M}_6^{\mathcal{R}_{red}}$ y $\mathcal{M}_7^{\mathcal{R}_{red}}$



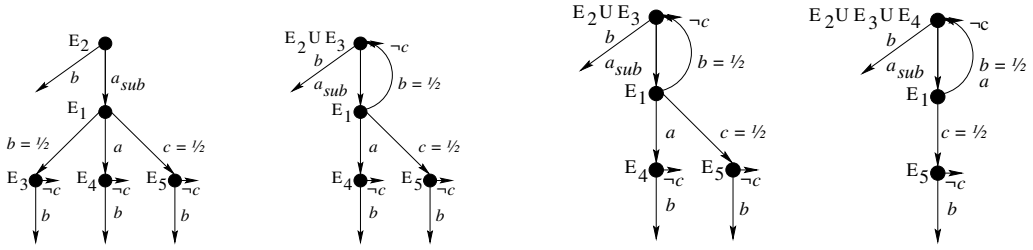
(a) $\mathcal{M}_8^{\mathcal{R}_{red}}$ (b) $\mathcal{M}_9^{\mathcal{R}_{red}}$

Figura A.5: Grafos MUS $\mathcal{M}_8^{\mathcal{R}_{red}}$ y $\mathcal{M}_9^{\mathcal{R}_{red}}$



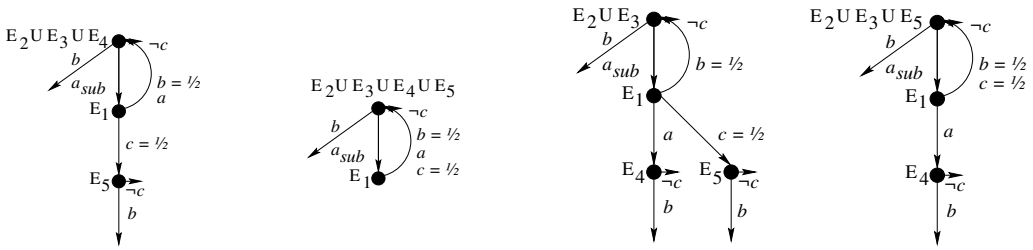
(a) $\mathcal{M}_{10}^{\mathcal{R}_{red}}$ (b) $\mathcal{M}_{11}^{\mathcal{R}_{red}}$

Figura A.6: Grafos MUS $\mathcal{M}_{10}^{\mathcal{R}_{red}}$ y $\mathcal{M}_{11}^{\mathcal{R}_{red}}$



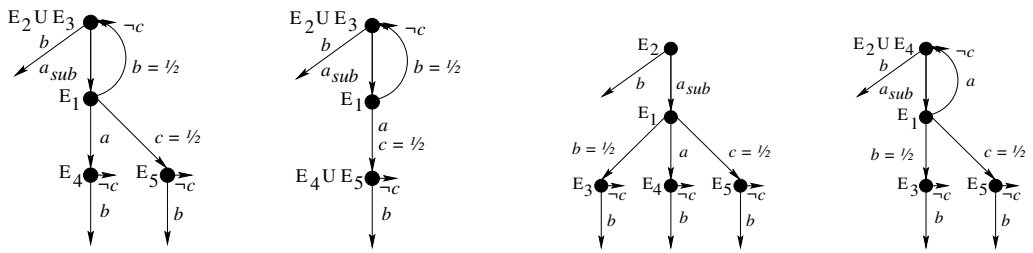
(a) $\mathcal{M}_{12}^{\mathcal{R}_{red}}$ (b) $\mathcal{M}_{13}^{\mathcal{R}_{red}}$

Figura A.7: Grafos MUS $\mathcal{M}_{12}^{\mathcal{R}_{red}}$ y $\mathcal{M}_{13}^{\mathcal{R}_{red}}$



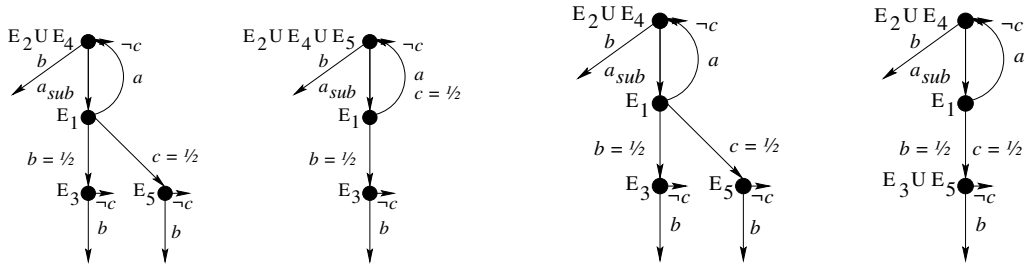
(a) $\mathcal{M}_{14}^{\mathcal{R}_{red}}$ (b) $\mathcal{M}_{15}^{\mathcal{R}_{red}}$

Figura A.8: Grafos MUS $\mathcal{M}_{14}^{\mathcal{R}_{red}}$ y $\mathcal{M}_{15}^{\mathcal{R}_{red}}$



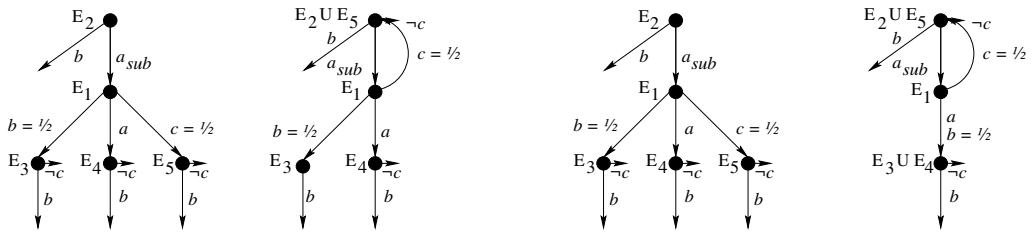
(a) $\mathcal{M}_{16}^{\mathcal{R}_{red}}$ (b) $\mathcal{M}_{17}^{\mathcal{R}_{red}}$

Figura A.9: Grafos MUS $\mathcal{M}_{16}^{\mathcal{R}_{red}}$ y $\mathcal{M}_{17}^{\mathcal{R}_{red}}$



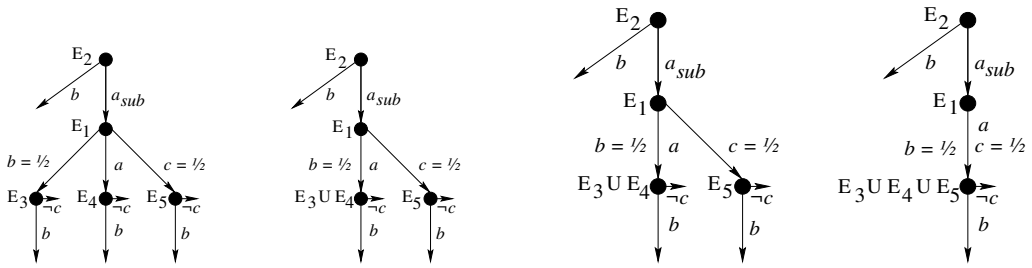
(a) $\mathcal{M}_{18}^{\mathcal{R}_{red}}$ (b) $\mathcal{M}_{19}^{\mathcal{R}_{red}}$

Figura A.10: Grafos MUS $\mathcal{M}_{18}^{\mathcal{R}_{red}}$ y $\mathcal{M}_{19}^{\mathcal{R}_{red}}$



(a) $\mathcal{M}_{20}^{\mathcal{R}_{red}}$ (b) $\mathcal{M}_{21}^{\mathcal{R}_{red}}$

Figura A.11: Grafos MUS $\mathcal{M}_{20}^{\mathcal{R}_{red}}$ y $\mathcal{M}_{21}^{\mathcal{R}_{red}}$



(a) $\mathcal{M}_{22}^{\mathcal{R}_{red}}$ (b) $\mathcal{M}_{23}^{\mathcal{R}_{red}}$

Figura A.12: Grafos MUS $\mathcal{M}_{22}^{\mathcal{R}_{red}}$ y $\mathcal{M}_{23}^{\mathcal{R}_{red}}$

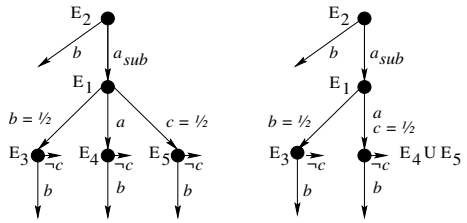


Figura A.13: Grafo MUS $\mathcal{M}_{24}^{\mathcal{R}_{red}}$

Apéndice B

Algoritmos auxiliares

B.1 Algoritmo de Notación Inversa

El algoritmo de notación inversa recibe como parámetro un requisito SCTL $\mathcal{R}$, y devuelve un requisito $\overline{\mathcal{R}}$ en notación inversa o prefija –primero el operador y luego los operandos–. La implementación de dicho algoritmo se realizó mediante un analizador sintáctico y gramatical construido con las herramientas *lex* y *yacc*. En el ejemplo B.1 se muestran tres requisitos SCTL y su representación en notación inversa.

Ejemplo B.1 Ejemplo del Algoritmo de Notación Inversa.

Requisito SCTL
$\boldsymbol{\mathcal{R}}_1 :\ (true \Rightarrow a) \Rightarrow \odot(true \Rightarrow \bigcirc b)$
$\boldsymbol{\mathcal{R}}_2 :\ (true \Rightarrow a) \wedge (true \Rightarrow \odot\, b) \Rightarrow \bigcirc(true \Rightarrow \odot\, c)$
$\boldsymbol{\mathcal{R}}_3 :\ (true \Rightarrow a) \wedge ((true \Rightarrow \odot\, b) \Rightarrow \bigcirc(true \Rightarrow \odot\, c))$

Requisito SCTL en Notación Inversa
$\overline{\boldsymbol{\mathcal{R}}_1} : \Rightarrow \odot \Rightarrow true\ a \Rightarrow \bigcirc true\ b$
$\overline{\boldsymbol{\mathcal{R}}_2} : \Rightarrow \bigcirc \wedge \Rightarrow true\ a \Rightarrow \odot\, true\ b \Rightarrow \odot\, true\ c$
$\overline{\boldsymbol{\mathcal{R}}_3} :\ \wedge \Rightarrow true\ a \Rightarrow \bigcirc \Rightarrow \odot\, true\ b \Rightarrow \odot\, true\ c$

B.2 Algoritmo de Cardinalidad

Dado un requisito SCTL $\mathcal{R}$, éste se puede considerar como un conjunto de n **items**. Si $\mathcal{R}[i]$ es el *item* que ocupa la posición i-ésima, $1 \leq i \leq n$, en el requisito $\mathcal{R}$, $\mathcal{R}[i]$ podrá ser cualquier

elemento constructor de requisitos SCTL: la especificación de una acción del conjunto Λ; un operador temporal del conjunto Θ; un operador lógico del conjunto Γ; una constante del conjunto Ψ; o un identificador de requisito precedido del símbolo $\uparrow$.

El ejemplo B.2 muestra el conjunto de *items* que forman el requisito $\mathcal{R}_{items}$ mostrado a continuación. Se considera como un *item* la especificación de la acción en el requisito, bien sea especificada como no posible o como posible. Es decir, el operador $\neg$ no se considera como un *item* del requisito si va ligado a una acción.

req $\mathcal{R}_{items}$ **is**
$(true \Rightarrow a_1) \ \Rightarrow \bigcirc \ ((true \Rightarrow \odot \neg a_2) \ \Rightarrow \uparrow R_{items})$
endreq

Ejemplo B.2 *Items* de un requisito SCTL.

Item[1]	$true$
Item[2]	$\Rightarrow$
Item[3]	a_1
Item[4]	$\Rightarrow \bigcirc$
Item[5]	$true$
Item[6]	$\Rightarrow \odot$
Item[7]	$\neg a_2$
Item[8]	$\Rightarrow$
Item[9]	$\uparrow R_{items}$

Definición B.1. Se define el **Cardinal de un Requisito SCTL** , $\mathcal{R}$, denotado por $\sharp R$, como el número de *items* que lo componen: $R = \{R[1], ..., R[n]\}, \sharp R \triangleq n$.

Algoritmo B.1 Algoritmo de Cardinalidad ($I = \{I[1], ..., I[n]\}$)

[1] Si $I[1] = \uparrow$ devolver $cardinal = 1$;
[2] Si $\overline{\mathcal{R}} = \{I[1], I[2], I[3]\}$ es un Requisito atómico, devolver $cardinal = 3$;
[3] $cardinal = 1 + \mathbf{Algoritmo\ de\ Cardinalidad}\ (\{\mathbf{I[2]}, ..., \mathbf{I[n]}\})$;
[4] $cardinal = cardinal + \mathbf{Algoritmo\ de\ Cardinalidad}\ (\{\mathbf{I[cardinal+1]}, ..., \mathbf{I[n]}\})$;
[5] Devolver $cardinal$;

El algoritmo de cardinalidad recibe como parámetro un conjunto de *items* SCTL, $\{I\}$, y devuelve el cardinal del primer requisito SCTL en notación inversa encontrado. Es decir, devuelve el menor m tal que los primeros m items de $\{I\}$ forman un requisito SCTL en notación inversa.

Su pseudocódigo se muestra en el algoritmo B.1. Obviamente, si el conjunto de *items* recibido forman un requisito SCTL en notación inversa, el algoritmo devuelve su cardinal, $\sharp I$.

Ejemplo B.3 Ejemplo del Algoritmo de Cardinalidad.

I	**Cardinalidad**
$\Rightarrow \odot true\ a$	3
$\wedge \Rightarrow true\ a \Rightarrow \odot true\ b \Rightarrow \odot true\ c$	7
$\Rightarrow \odot true\ b \Rightarrow \odot true\ c$	3

B.3 Algoritmo de Partición

Dado un requisito SCTL no atómico expresado en notación inversa $\overline{\mathcal{R}}$, siempre se puede dividir en dos requisitos SCTL en notación inversa, $\overline{\mathcal{R}}_{sub_1}$ y $\overline{\mathcal{R}}_{sub_2}$, unidos por un operador $\bigoplus \in \{\Theta \cup \Gamma\}$. La obtención de $\bigoplus$ es inmediata, ya que, se corresponde con el primer *item* del requisito $\overline{\mathcal{R}}$, $\bigoplus = \overline{\mathcal{R}}[1]$. El algoritmo de partición de requisitos recibe un requisito SCTL no atómico en notación inversa $\overline{\mathcal{R}}$, y devuelve $\overline{\mathcal{R}}_{sub_1}$ y $\overline{\mathcal{R}}_{sub_2}$. Su pseudocódigo se muestra en el algoritmo B.2. En el ejemplo B.4 se ilustra el funcionamiento del algoritmo de partición.

Algoritmo B.2 Algoritmo de Partición ($\overline{\mathcal{R}} = \{\overline{\mathcal{R}}[1], ..., \overline{\mathcal{R}}[n]\}$), $n > 3$

[1] $n\_items\_1 =$ **Algoritmo de Cardinalidad** $(\{\overline{\mathcal{R}}[\mathbf{2}], ..., \overline{\mathcal{R}}[\mathbf{n}]\})$;
[2] $\overline{\mathcal{R}}_{sub_1} = \{\overline{\mathcal{R}}[2], ..., \overline{\mathcal{R}}[n\_items\_1 + 1]\}$
[3] $\overline{\mathcal{R}}_{sub_2} = \{\overline{\mathcal{R}}[n\_items\_1 + 2], ..., \overline{\mathcal{R}}[n]\}$
[4] Devolver $\overline{\mathcal{R}}_{sub_1}$ y $\overline{\mathcal{R}}_{sub_2}$

Ejemplo B.4 Ejemplo del Algoritmo de Partición.

$\overline{\mathcal{R}}$	$\overline{\mathcal{R}_1}$	$\overline{\mathcal{R}_2}$
$\Rightarrow \odot \Rightarrow true\ a \Rightarrow \bigcirc true\ b$	$\Rightarrow true\ a$	$\Rightarrow \bigcirc true\ b$
$\Rightarrow \bigcirc \wedge \Rightarrow true\ a \Rightarrow \odot true\ b \Rightarrow \odot true\ c$	$\wedge \Rightarrow true\ a \Rightarrow \odot true\ b$	$\Rightarrow \odot true\ c$
$\wedge \Rightarrow true\ a \Rightarrow \bigcirc \Rightarrow \odot true\ b \Rightarrow \odot true\ c$	$\Rightarrow true\ a$	$\Rightarrow \bigcirc \Rightarrow \odot true\ b \Rightarrow \odot true\ c$

B.4 Algoritmo de Extracción de SubRequisitos

Según la sintaxis SCTL definida, los requisitos SCTL están compuestos por una premisa, un operador temporal y una consecuencia. A su vez, la premisa y la consecuencia son de nuevo requisitos SCTL que estarán formados cada uno de ellos por una premisa, un operador temporal y una consecuencia. Esta estructura se repite hasta que, finalmente, la premisa es una constante y la consecuencia es la especificación de una acción; es decir, hasta que en la recursión se obtiene un requisito atómico.

Definición B.2. Dado un requisito SCTL $\mathcal{R}$, se definen los **Subrequisitos** de $\mathcal{R}$ como el conjunto de requisitos SCTL que lo forman, $\mathcal{R}_{sub} = \{\mathcal{R}_{sub_1}, ..., \mathcal{R}_{sub_m}\}$,

El objetivo del **algoritmo de extracción de subrequisitos** es extraer el conjunto de subrequisitos que forman un requisito SCTL $\mathcal{R}$. A continuación se muestran dos versiones de dicho algoritmo, una recursiva y otra iterativa. En ambos casos, el algoritmo de extracción recibe como parámetro un requisito SCTL en notación inversa $\overline{\mathcal{R}}$, y devuelve la lista de subrequisitos que lo componen $\overline{\mathcal{R}}_{sub} = \{\overline{\mathcal{R}}_{sub_1}, ..., \overline{\mathcal{R}}_{sub_m}\}$. Además, el algoritmo de extracción recursivo recibe como parámetro un número de orden, i, que se incrementa cada vez que se invoca una nueva recursión de dicho algoritmo. Por tanto, la primera vez que se invoca al algoritmo, el valor de i debe ser 1. Su pseudocódigo se muestra en el algoritmo B.3.

Algoritmo B.3 Algoritmo de Extracción Recursivo ($\overline{\mathcal{R}} = \{\overline{\mathcal{R}}[1], ..., \overline{\mathcal{R}}[n]\},\ i = 1$)

[1] Si $\overline{\mathcal{R}}$ es un Requisito atómico, Fin;
[2] $j = i,\ i = i + 1$;
[3] $\{\overline{\mathcal{R}}_{sub_j}, \overline{\mathcal{R}}_{aux}\} = \mathbf{Algoritmo\ de\ Particion}\ (\overline{\mathcal{R}})$
[4] $\mathbf{Algoritmo\ de\ Extraccion\ Recursivo}\ (\overline{\mathcal{R}}_{\mathbf{sub_j}}, \mathbf{i})$;
[5] $j = i,\ i = i + 1$;
[6] $\overline{\mathcal{R}}_{sub_j} = \overline{\mathcal{R}}_{aux}$;
[7] $\mathbf{Algoritmo\ de\ Extraccion\ Recursivo}\ (\overline{\mathcal{R}}_{\mathbf{sub_j}}, \mathbf{i})$;
[8] Fin;

En el ejemplo B.5 se muestra un ejemplo del algoritmo de extracción recursivo, detallándose los subrequisitos extraídos en cada recursión del algoritmo. El algoritmo B.4 muestra el pseudocódigo del algoritmo de extracción iterativo. Obsérvese como se va haciendo hueco para dos nuevos subrequisitos cada vez que aparece un nuevo subrequisito no atómico (pasos **[5]** y **[6]**). En el ejemplo B.6 se muestra un ejemplo del algoritmo de extracción iterativo, detallándose los subrequisitos extraídos en cada iteración del algoritmo. Solamente se indican las iteraciones en las que se extraen nuevos subrequisitos (las iteraciones en las que se invoca el algoritmo de partición). La complejidad de ambos algoritmos de extracción es la misma, ya que, los dos realizan el mismo número de llamadas al algoritmo de partición.

Ejemplo B.5 Ejemplo del Algoritmo de Extracción Recursivo.

Rec	$\overline{\mathcal{R}}$	**Subrequisitos**
[1]	$\Rightarrow \odot \Rightarrow true\ a \Rightarrow \bigcirc true\ b$	$\overline{\mathcal{R}}_{sub_1} = \Rightarrow true\ a$ $\overline{\mathcal{R}}_{sub_2} = \Rightarrow \bigcirc true\ b$

Rec	$\overline{\mathcal{R}}$	**Subrequisitos**
[1]	$\Rightarrow \bigcirc \wedge \Rightarrow true\ a \Rightarrow \odot true\ b \Rightarrow \odot true\ c$	$\overline{\mathcal{R}}_{sub_1} = \wedge \Rightarrow true\ a \Rightarrow \odot true\ b$ $\overline{\mathcal{R}}_{sub_4} = \Rightarrow \odot true\ c$
[2]	$\wedge \Rightarrow true\ a \Rightarrow \odot true\ b$	$\overline{\mathcal{R}}_{sub_2} = \Rightarrow true\ a$ $\overline{\mathcal{R}}_{sub_3} = \Rightarrow \odot true\ b$

Rec	$\overline{\mathcal{R}}$	**Subrequisitos**
[1]	$\wedge \Rightarrow true\ a \Rightarrow \bigcirc \Rightarrow \odot true\ b \Rightarrow \odot true\ c$	$\overline{\mathcal{R}}_{sub_1} = \Rightarrow true\ a$ $\overline{\mathcal{R}}_{sub_2} = \Rightarrow \bigcirc \Rightarrow \odot true\ b \Rightarrow \odot true\ c$
[2]	$\Rightarrow \bigcirc \Rightarrow \odot true\ b \Rightarrow \odot true\ c$	$\overline{\mathcal{R}}_{sub_3} = \Rightarrow \odot true\ b$ $\overline{\mathcal{R}}_{sub_4} = \Rightarrow \odot true\ c$

Algoritmo B.4 Algoritmo de Extracción Iterativo ($\overline{\mathcal{R}} = \{\overline{\mathcal{R}}[1], ..., \overline{\mathcal{R}}[n]\}$)

[1] Si $\overline{\mathcal{R}}$ es un Requisito atómico, Fin;
[2] $\{\overline{\mathcal{R}}_{sub_1}, \overline{\mathcal{R}}_{sub_2}\} = \mathbf{Algoritmo\ de\ Particion}\ (\overline{\mathcal{R}})$
[3] $i = 1,\ ult = 2$;
[4] Si $\overline{\mathcal{R}}_{sub_i}$ es un Requisito atómico, ir al paso 8;
[5] *FOR* $j = ult$ *DOWNTO* $j = i + 1$ *DO* $\overline{\mathcal{R}}_{sub_{j+2}} = \overline{\mathcal{R}}_{sub_j}$;
[6] $ult = ult + 2$;
[7] $\{\overline{\mathcal{R}}_{sub_{i+1}}, \overline{\mathcal{R}}_{sub_{i+2}}\} = \mathbf{Algoritmo\ de\ Particion}\ (\overline{\mathcal{R}}_{sub_i})$
[8] $i = i + 1$;
[9] Si $i \leq ult$, ir al paso 4;
[10] Fin;

Ejemplo B.6 Ejemplo del Algoritmo de Extracción Iterativo.

$\overline{\mathcal{R}} \equiv \Rightarrow \odot \Rightarrow \boldsymbol{true\ a} \Rightarrow \bigcirc \boldsymbol{true\ b}$	
Iteración [1]	
$\overline{\mathcal{R}}_{sub_1} = \Rightarrow true\ a$	
$\overline{\mathcal{R}}_{sub_4} = \Rightarrow \odot true\ c$	
$\overline{\mathcal{R}} \equiv \Rightarrow \bigcirc \wedge \Rightarrow \boldsymbol{true\ a} \Rightarrow \odot \boldsymbol{true\ b} \Rightarrow \odot \boldsymbol{true\ c}$	
Iteración [1]	**Iteración [2]**
$\overline{\mathcal{R}}_{sub_1} = \wedge \Rightarrow true\ a \Rightarrow \odot true\ b$	$\overline{R}_{sub_1} = \wedge \Rightarrow true\ a \Rightarrow \odot true\ b$
$\overline{\mathcal{R}}_{sub_2} = \Rightarrow \odot true\ c$	$\overline{\mathcal{R}}_{sub_2} = \Rightarrow true\ a$
	$\overline{\mathcal{R}}_{sub_3} = \Rightarrow \odot true\ b$
	$\overline{\mathcal{R}}_{sub_4} = \Rightarrow \odot true\ c$
$\overline{\mathcal{R}} \equiv \wedge \Rightarrow \boldsymbol{true\ a} \Rightarrow \bigcirc \Rightarrow \odot \boldsymbol{true\ b} \Rightarrow \odot \boldsymbol{true\ c}$	
Iteración [1]	**Iteración [2]**
$\overline{\mathcal{R}}_{sub_1} = \Rightarrow true\ a$	$\overline{\mathcal{R}}_{sub_1} = \Rightarrow true\ a$
$\overline{\mathcal{R}}_{sub_2} = \Rightarrow \bigcirc \Rightarrow \odot true\ b \Rightarrow \odot true\ c$	$\overline{\mathcal{R}}_{sub_2} = \Rightarrow \bigcirc \Rightarrow \odot true\ b \Rightarrow \odot true\ c$
	$\overline{\mathcal{R}}_{sub_3} = \Rightarrow \odot true\ b$
	$\overline{\mathcal{R}}_{sub_4} = \Rightarrow \odot true\ c$

Bibliografía

[Aba97] J. Benavides Abajo. *SQL para usuarios y programadores*. Paraninfo, ISBN: 84-283-1821-2, 1997.

[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.

[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.

[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, April 1994.

[BD87] S. Budkowski and P. Dembinski. An introduction to ESTELLE: a specification language for distributed systems. *Computer Networks and ISDN Systems*, 14(1):3–23, 1987. North-Holland, Amsterdam.

[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, July 1996.

[Ber94] D.M. Berry. Wither formal methods? In *Proceedings of the 1994 Monterey Workshop*, Monterey, California, 1994.

[BF96] E. Burke and E. Foxley. *Logic and its applications*. Prentice Hall, 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*, October 1994.

[BH95a] J.P. Bowen and M.G. Hinchey. Seven more myths of formal methods. *IEEE Software*, 12(4):34–41, July 1995.

[BH95b] J.P. Bowen and M.G. Hinchey. Ten commandments of formal methods. *Computer*, 28(4):56–63, July 1995.

[BKM+77] L. Banachowski, A. Kreczmar, G. Mirkowska, H. Rasiowa, and A. Salwicki. An introduction to algorithmic logic. *Mathematical Foundation of Computer Science*, 1977. Bannach Center Publications.

[BM79] R.S. Boyer and J.S. Moore. *A computational logic*. Academic Press, New York, 1979.

[Boe76] B.W. Boehm. Software engineering. *IEEE Transactions on Computers*, 25(12):1226–1241, December 1976.

[Boe81] B. W. Boehm. *Software Engineering Economics*. Prentice Hall, 1981.

[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, October 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, April 1996.

[Bry86] R. E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, 35(8), 1986.

[Bud92] S. 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.

[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, April 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.Ñeilson, 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, April 1988.

[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, January 1993.

[CW96a] E.M. Clarke and Jeannette Wing. Strategic directions in computing research. tools and partial analysis. *ACM Computing Surveys*, 28A(4), December 1996.

[CW96b] E.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.

[Dav93] A.M. Davis. *Software Requirements. Objects, functions and states*. Prentice Hall, 1993.

[DB89] P. Dembinski and S. Budkowski. The specification language ESTELLE. *Diaz et al.*, pages 33–75, 1989.

[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.L. Dill. The Mur∅ verification system. In R. Alur and T.A. Henzinger, editors, *Computer Aided Verification*, volume 1102 of *Lecture Notes in Computer Science*. Springer-Verlag, 1996.

[DR96] D.L. Dill and J. Rushby. Acceptance of formal methods: Lessons from hardware design. *Computer*, 29(4):23–24, 1996.

[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*, December 1994.

[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.

[Eme90] E. A. Emerson. Temporal and modal logic. In Jan Van Leeuwen, editor, *Formal Models and Semantics*, volume B of *Handbook of Theoretical Computer Science*, chapter 16, pages 995–1072. Elsevier Science Publishers B.V., 1990.

[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.

[FG94] J. C. Ferrando and V. Gregori. *Matemática Discreta*. Ed. Reverté, 1994.

[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, July 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, May 1994.

[FL79] M.J. Fischer and R.E. Ladner. Propositional dynamic logic of regular programs. *Journal Computer Systems Science*, 18(2):194–211, 1979.

[Flo95] J. Floch. Supporting evolution and maintenance by using a flexible code generator. In *17th Int'l Conf. on Software Engineering*, Seattle, April 1995.

[GG88] S. Garland and J. 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, December 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. 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*. Springe-Verlag, 1979.

[Got92] R. Gotzhein. Temporal logic and its applications. *Computer Networks and ISDN Systems*, 24:203–218, 1992.

[GS99] A. Gil-Solla. *Diseño y verificación de sistemas distribuidos mediante la aplicación combinada de métodos formales*. PhD thesis, Departamento de Tecnología de las Comunicaciones - University of Vigo, 1999.

[GT96] W.K. Grassmann and Jean-Paul Tremblay. *Logic and discrete mathematics. A computer science perspective*. Prentice-Hall, 1996.

[Hal90] J.A. Hall. Seven myths of formal methods. *IEEE Software*, 7(5):11–19, September 1990.

[Hal96] J.A. Hall. Using formal methods to develop an ATC information system. *IEEE Software*, 12, March 1996.

[Har79a] D. Harel. *First-Order Dynamic Logic*. Number 68 in Lecture Notes in Computer Science. Springer-Verlag, 1979. Berlin.

[Har79b] D. Harel. Two results on process logic. *Information Processing Letters*, 8(4):195–198, 1979.

[HB95] M.G. Hinchey and J.P. Bowen. *Applications of Formal Methods*. Prentice Hall, 1995.

[HB96a] M.G. Hinchey and J.P. Bowen. To formalize or not to formalize. *Computer*, 29(4):18–19, 1996.

[HB96b] M.C. Holloway and R.W. Butler. Impediments to industrial use of formal methods. *Computer*, 29(4):25–26, 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, October 1996.

[HK90] Z. Harel and R.P. Kurshan. Software for analytica development of communications protocols. *AT&T Bell Laboratories Journal*, 69(1):45–59, January-February 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.

[Hoa85] C.A.R. Hoare. *Communicating Sequential Processes*. Prentice-Hall International, 1985.

[Hol91] G.J. 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.

[Inf96a] Informix Software, Inc. 4100 Bohannon Drive Menlo Park, CA 94025. *INFORMIX-ESQL/C Programmer's Manual Version 7.2*, 1996.

[Inf96b] Informix Software, Inc. 4100 Bohannon Drive Menlo Park, CA 94025. *Informix Guide to SQL. Syntax version 7.2*, 1996.

[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.

[ISO91] ISO. *Information Processing Systems - Open Systems Interconnection - Guidelines for the Application of ESTELLE, LOTOS and SDL*. ISO/IEC TR10167, Geneva, 1991.

[ISO93] ISO. *Information Technology Programming Languages -VDM-SL*. ISO/IEC/JTC1/SC22/WG19, n-20 edition, November 1993.

[ISO95] ISO. *Z Notation*. ISO Panel JTC1/SC22/WG19, version 1.2 edition, September 1995.

[ISO98] ISO. *Final Commite Draft on Enhancements to LOTOS*. ISO/IEC JTC1/SC21/WG7, May 1998.

[JG88] G. Jones and M. Goldsmith. *Programming in occam2*. Prentice-Hall, 1988.

[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.

[JMM95] A. Coombes J. Moffett, J.A. Hall and J. McDermid. A model for a causal logic for requirements engineering. *Journal of Requirements Engineering*, 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.

[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.

[KR92] B.W. Kernighan and D.M. Ritchie. *El lenguaje de programación C*. Prentice Hall, ISBN: 968-880-205-0, 1992.

[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.

[Lam84] L. Lamport. The temporal logic of actions. *ACM TOPLAS*, pages 872–923, 1984.

[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, January 1983.

[LG97] Luqi and J. A. Goguen. Formal methods: Promises and problems. *IEEE Software*, pages 73–85, January 1997.

[Lim88] INMOS Limited. *occam2 Reference Manual*. Prentice-Hall, 1988.

[LK95] P. Loucopoulos and V. 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, May 1992.

[MB91] T. Mason and D. Brown. *Lex & Yacc*. O'Reilly & Associates, Inc., 1991.

[McM93] K.L. McMillan. *Symbolic model checking: an approach to the state explosion problem*. Kluwer Academic Publishers, 1993.

[McM98] K.L. McMillan. *Symbolic model checking*. Kluwer Academic Publishers, 5 edition, 1998.

[MdM88] J. Mañas and T. de Miguel. From LOTOS to C. In *Formal Descriptions Techniques. FORTE 88*, Stirling, Escocia, September 1988.

[MFV94] K. Kumax M.D. Fraser and V.K. Vaishnaki. Strategies for incorporating formal specifications in software development. *Communications ACM*, 37(10):74–86, 1994.

[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, April 1985.

[Mor99] M. Morgan. *Descubre Java 1.2*. Prentice Hall, 1999.

[MP92] Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems*. Springer-Verlag, New York, 1992.

[MS93] P. Mukherjee and V. Stavridou. The formal specification of safety requirements for storing explosives. *Formal Aspects of Computing*, 5:299–336, 1993.

[NPT92] J. van Katwijk N. Plat and H. Toetenel. Application and benefits of formal methods in software development. *Software Engineering Journal*, 7(5):335–347, September 1992.

[NR69] P. Naur and B. Randell. Software engineering: A report on a conference sponsored by the nato science committee. Technical report, 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, June 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+97] J.J. Pazos-Arias, J. García-Duque, A. Gil-Solla, R.P. Díaz-Redondo, and A. Fdez.-Vilas. Una lógica temporal causal para la especificación de requisitos funcionales de un sistema distribuido. In J.J. Pazos M. Llamas and M. Fernández, editors, *V Jornadas de Concurrencia*, pages 321–332, 1997.

[PAGDGS+98] J.J. Pazos-Arias, J. García-Duque, A. Gil-Solla, R.P. Díaz-Redondo, and A. Fdez.-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.

[Par78] R. Parikh. A completeness result for PDL. In *Symposium on Mathematical Foundations of Computer Science*, Zakopane, Warsaw, May 1978. Springer-Verlag.

[Pau94] L. C. Paulson. *Isabelle: A Generic Theorem Prover*, volume 828 of *Lecture Notes in Computer Science*. Springer-Verlag, 1994.

[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.

[PG93] N. Peermingeat and D. Glaude. *Álgebra de Boole. Teoría, métodos de cálculo, aplicaciones.* Ed. Vicens Vives, 1993.

[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.

[PLB96] J. Fitzgerald P.G. Larsen and T. Brookes. Applying formal specification in industry. *IEEE Software*, 13(3):48–56, 1996.

[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, October 1977.

[Pnu79] A. Pnueli. The temporal semantics of concurrent programs. In *Proc. International Symposium on Semantics of Concurrent Programs*, Evian, France, July 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.

[Pra78] V.R. Pratt. Semantical considerations on Floyd-Hoare logic. In *Proc. 10th ACM Symposium on Theory of Computing*, pages 326–337, May 1978.

[Pra79] V.R. Pratt. Process logic. In *Proc. 6th ACM Symposium on Principles of Programming Languages*, pages 93–100, January 1979.

[PvED89] C.A. Vissers P.H.J. van Eijk and M. Diaz, editors. *The Formal Description Technique LOTOS*. Elsevier Science Publishers B.V, 1989.

[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, June 1989.

[QS82] J. Queille and J. Sifakis. Specification y verification of concurrent systems with CAESAR. In *Proc. of fith ISP*, 1982.

[ROM89] D. Richardson, T. O'Malley, and C.T. Moore. Approaches to specification-based testing. In *ACM Sigsoft 89: Third Symposium on Software Testing, Analysis and Verification*, December 1989.

[Ros94] A.W. Roscoe. *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, June 1990. American Mathematical Society.

[RW92] K.A. Ross and C.R.B. Wright. *Dicrete mathematics*. Prentice-Hall, 1992.

[Sip99] M. Siple. *The Complete Guide Java Database Programming*. McGraw-Hill, 1999.

[Som95] I. Sommerville. *Software Engineering*. Addison-Wesley, 1995.

[Spi88] J.M Spivey. *Introducing Z: a Specification Language and its Formal Semantics*. Cambridge University Press, 1988.

[Spi92] J.M Spivey. *The Z notation: A Reference Manual*. Prentice Hall, London, 1992.

[Sta97] W. Stallings. *Data and Computer Communications*. Prentice Hall, 1997.

[Std84] IEEE Std.830-1984. *IEEE Guide to Software Requirements Specification*. The Institute of Electrical and Electronics Engineers: New York, 1984.

[Std90] IEEE Std.610.12-1990. *IEEE Standard Glossary of Software Engineering Terminology*. The Institute of Electrical and Electronics Engineers: New York, 1990.

[Sti96] C. Stirling. *Modal and Temporal Logics for Processes*. Number 1043 in Lecture Notes in Computer Science. Springer-Verlag, 1996.

[Tan96] A. S. Tanenbaum. *Computer Networks*. Prentice Hall, 1996.

[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.

[Tri80] E. Trillas. *Conjuntos Borrosos*. Vicens-Vives, 1980.

[Tur93] K. J. Turner. *Using Formal Description Techniques. An introduction to ESTELLE, LOTOS and SDL*. John Wiley & Sons, 1993.

[VW86] M.Y. Vardi and P. Wolper. An automata-theoretic approach to automatic program verification. In *Proc. of Logic in Computer Science*, 1986.

[Xia95] J. Xiaoping. An approach to animating Z specifications. In *Proc. Int'l Computer Software and Applications Conf*, Dallas, Texas, 1995.

[Z.188] CCITT Recommendation Z.100. Specification and description language sdl, annexes a-f to z.100. Technical report, Blue Book, Volume VI.20 - VI.24, ITU, General Secretariat, Sales Section, Places des Nations, CH-1211, Genova 20, 1988.