UNIVERSIDADE DE VIGO

Dpto. de Enxeñería Telemática
ETSE de Telecomunicación

TESIS DOCTORAL

REUTILIZACIÓN DE REQUISITOS FUNCIONALES DE SISTEMAS DISTRIBUIDOS UTILIZANDO TÉCNICAS DE DESCRIPCIÓN FORMAL

Autor: Rebeca P. Díaz Redondo
Ingeniero de Telecomunicación

Director: José J. Pazos Arias
Doctor Ingeniero de Telecomunicación

Febrero de 2002

Dpto. de Enxeñería Telemática
ETSE de Telecomunicación
Universidade de Vigo
Campus Universitario s/n
E-36200 Vigo

TESIS DOCTORAL

REUTILIZACIÓN DE REQUISITOS FUNCIONALES DE SISTEMAS DISTRIBUIDOS UTILIZANDO TÉCNICAS DE DESCRIPCIÓN FORMAL

Autor: Rebeca P. Díaz Redondo
Ingeniero de Telecomunicación

Director: José J. Pazos Arias
Doctor Ingeniero de Telecomunicación

Febrero de 2002

TESIS DOCTORAL

REUTILIZACIÓN DE REQUISITOS FUNCIONALES DE SISTEMAS DISTRIBUIDOS UTILIZANDO TÉCNICAS DE DESCRIPCIÓN FORMAL

Autor: Dña. Rebeca P. Díaz Redondo
Director: Dr. D. José J. Pazos Arias

TRIBUNAL CALIFICADOR

Presidente: Dr. D.

Vocales:

Dr. D.

Dr. D.

Dr. D.

Secretario: Dr. D.

CALIFICACIÓN:

Vigo, a de de 2002.

A mis padres,
a Adriana, y a Pedro.

Agradecimientos

Me gustaría, desde estas líneas, expresar mi gratitud hacia todos los que habéis caminado junto a mí estos años, allanándome la vía y permitiendo que hoy haya alcanzado una nueva encrucijada.

La ayuda de José J. Pazos, director de este trabajo, ha resultado inestimable, tanto en todos los aspectos relacionados con esta tesis, como en el día a día como compañero, muchas gracias.

A mi compañera de despacho, docencia, y fatigas, Ana F. Vilas, agradecerle especialmente que haya compartido conmigo todas las etapas que suponen un trabajo de tesis. El haberlo sobrellevado juntas me ha dado fuerzas y, desde luego, me deja muchos momentos divertidos para recordar.

Si a algún lector le parece que el formato de este documento es agradable y apropiado, debería saber que no habría sido posible sin los acertados consejos de Manuel F. Veiga, que siempre conseguía disponer de un momento para colaborar conmigo y, amablemente, resolver mis dudas.

Me gustaría agradecerle a Belén Barragáns no sólo su colaboración durante las primeras fases de implementación de este trabajo, sino el haber sido una estupenda compañera de docencia estos dos últimos años (ya vamos por el tercero...).

La dedicación de Carmen Page en la implementación de una interfaz gráfica para el entorno de reutilización ha resultado imprescindible, desde aquí, gracias.

La financiación proporcionada por la Xunta y materializada en dos proyectos, XUGA 32206A97 (1997-1999) y PGIDT01PXI32203PR (2000-2004), supuso el soporte técnico necesario en la elaboración de este trabajo.

Desde luego este camino me habría resultado mucho más difícil de recorrer sin el estupendo ambiente de trabajo del que he disfrutado junto a los que, además de compañeros, considero buenos amigos: Ana F. Vilas, Alberto Gil, Andrés Suárez, Belén Barragáns, Cándido López, Estela Sousa, Jorge G. Duque, José C. L. Ardao, José J. Pazos, Manolo Ramos, Manuel F. Veiga, y Raúl Rodríguez. Muchas gracias a todos.

Por último, aunque no menos importante, a mi familia, a Pedro, y a mis amigos, especialmente a Juan, María, Victoria, Rosa (y familia) y Celeste, porque vuestra confianza siempre supuso un acicate para mí y puede que no lo sepáis, pero habéis sido y sois imprescindibles.

Vigo, Febrero de 2002

Resumen

La reutilización de software fue planteada en su día como una vía complementaria para la mejora de los procesos de desarrollo de sistemas, con los objetivos de aligerar todas las tareas propias de estos procesos e incrementar la calidad de los sistemas obtenidos. Los investigadores en este ámbito coinciden en asegurar que un programa de reutilización sistemático, automatizado y formal conseguiría todos estos propósitos, aunque hasta el momento los intentos de incorporar este tipo de planes de reutilización se han visto ralentizados por diversos factores, de los cuales quizá la carencia de entornos tecnológicos apropiados sea uno de los más destacables.

Es en este campo donde, en los últimos años, han surgido diferentes estudios con el objetivo de dotar de soporte técnico a las tareas propias de un entorno de reutilización, y permitir que ésta pueda ser realizada sistemáticamente. Sin embargo y hasta el momento, no ha sido posible la obtención de una metodología amplia y común a todos los procesos de desarrollo de sistemas, sino que los entornos de reutilización, para que sean efectivos, han de ser diseñados específicamente para cada proceso de diseño y desarrollo de software. Es precisamente en esta línea en la que se enmarca el trabajo de esta tesis, proporcionando las bases teóricas y formales, y describiendo una metodología de aplicación de las mismas con el objetivo de lograr un entorno de reutilización especialmente adaptado a un proceso de obtención de sistemas ya existente.

El entorno de trabajo de partida es el resultado de los estudios realizados por el grupo de Redes e Ingeniería del Software del Departamento de Enxeñería Telemática de la Universidade de Vigo. La principal aportación de estas investigaciones ha sido la obtención de una metodología de diseño y desarrollo de procesos software, llamada SCTL-MUS, que combina la formalización del proceso, conjugando diferentes FDTs, y un enfoque iterativo e incremental del mismo. La utilización de la lógica multivalorada SCTL para la especificación de requisitos funcionales permite que la captura de éstos se realice formalmente, pero sin alejarse demasiado de la semántica del lenguaje natural; y el modelo de estados subespecificados MUS facilita las labores de prototipado y realimentación con el usuario, necesarias en un modelo iterativo e incremental.

Debido a las peculiaridades de este proceso, se ha constatado que uno de los puntos débiles del mismo es la elevada frecuencia de ejecución del algoritmo de verificación —basado en técnicas de *model checking*—, y, debido a ello, se propone la reutilización de información de verificación asociada a modelos de sistemas, completos o incompletos, con el objetivo de aliviar la carga computacional requerida para las tareas de verificación. Pero no es ésta la única circunstancia propicia para la reutilización en el entorno de desarrollo de partida, sino que a la hora de comenzar con el proceso de creación de un nuevo sistema también sería conveniente ser capaces de evitar trabajo redundante y partir de algún modelo apropiado sobre el que se haya trabajado con anterioridad, de esta forma podría incremen-

tarse la eficiencia del proceso al reducirse las labores de síntesis iniciales. De esta manera nuestra propuesta establece la reutilización de elementos software de un elevado nivel de abstracción, como es el caso de requisitos funcionales y de información de verificación, frente a la reutilización más habitual de elementos de bajo nivel de abstracción, normalmente código que, aunque *a priori* más sencilla, resulta menos rentable y atractiva.

La gestión adecuada de un elevado volumen de información es el principal escollo que es necesario solventar para que un proceso de desarrollo con reutilización pueda encarnar una alternativa viable a los procesos de desarrollo tradicionales. Es vital que el acceso a esta información pueda realizarse eficaz y eficientemente ya que, en caso contrario, todo el trabajo redundante que se intenta minimizar se transforma en un trabajo igual o todavía mayor a la hora de recuperar la información requerida. Precisamente por ello, la primera parte del trabajo se dedica a la obtención de criterios de clasificación que permitan la gestión adecuada de la biblioteca de componentes. Tras estudiar las diferentes alternativas, se optó por aprovechar la formalización presente en los propios componentes como base para extraer una representación interna de los mismos que facilite su clasificación, almacenamiento, y posterior recuperación, evitando así los problemas de ambigüedad típicamente inherentes a cualquier tipo de descripción textual.

La proximidad funcional es el criterio subyacente en las relaciones de orden parcial y equivalencia definidas para organizar la biblioteca de componentes reutilizables, permitiendo así que componentes funcionalmente semejantes mantengan vínculos más estrechos en la biblioteca. Estos vínculos, aunque muy útiles, no son suficientes para estimar objetiva e independientemente la similitud funcional entre dos componentes cualesquiera, para ello se hace imprescindible el establecimiento de métricas que permitan cuantificar la proximidad funcional. El conjunto de relaciones de orden, de equivalencia, y métricas definido constituye la base para todas las tareas de administración de la información disponible en la biblioteca.

Tras el establecimiento de estas bases se definen los protocolos de clasificación, recuperación y adaptación de componentes con el objetivo de que bien su información de verificación, bien los propios modelos, sean reutilizados. Asimismo es necesario definir qué información de verificación será necesario almacenar y cómo la recuperación de esta información puede ayudar a minimizar las labores de verificación originales.

Abstract

Software reuse was set out, from the beginning, as a supplementary way to enhance system processes, with the aims of lightening these processes tasks and increasing the quality of the resulting systems. The researchers on this field agree on assuring that a systematic, automated and formal reusing program could reach all these goals, although for the moment the attempts for adding this kind of reusing plans have been slowed down by many different factors, among them the lack of suitable technologic environments may be the most distinguished one.

In the last years, different investigation lines have been emerged from this field, trying to provide suitable technical support to reusing environments with the aim of allowing systematic reuse. Nevertheless, and for the moment, it has been impossible to obtain a broad and common methodology to all development processes, and, as conclusion, each reusing environment must be specifically designed for each development process to be effective. This PhD is in line with this research field, providing theoretical and formal basis and describing an application methodology in order to achieve a reusing environment especially adapted to a previous development process.

The starting framework is the result of the previous work tackled by the Redes e Ingeniería del Software group, from the Departamento de Enxeñería Telemática of the Universidade de Vigo. The main contribution of these works is a design and development methodology for software systems, called SCTL-MUS, which joins on the one hand the totally formalization of the process, combining different FDTs, and, on the other hand, an incremental and iterative point of view. Using the manyvalued logic SCTL for the functional requirements specification, allows their formal description, without being too far from natural language semantic. The state-transition formalism MUS facilities the prototype and feed back with users, which are essential in an incremental and iterative life cycle model.

As verification is present over the whole software process, we have noticed that one of the main lacks in this kind of processes is the great computing resources needed to verify medium-large and large systems, so our proposal consists of reusing previous verification results —which are obtained from a model checking algorithm— to minimize these verification costs. From this proposal, it springs the possibility of reusing system specifications —possibly incomplete models— with the aim of reducing the specification efforts by picking already developed components that closely satisfy the properties required for the new specification.

On the whole, our proposal consists of reusing high abstract level components, like functional requirements and verification information, opposite of reusing low abstract level components, like code, which although being more easer and usual, it results a less profit and attractive alternative.

In order to reuse any kind of information it is essential being able to search and to retrieve it from a library in an efficient and effective way, and it is one of

the main lacks in software reuse: organizing large collections of reusable components. Because of this, the first part of the work is focused on studying and defining an appropriate classification and retrieval criteria which allows reusing software in a less expensive way than building it from scratch. After studying different alternatives, we have opted by a totally formalized management of the repository because of the formal character of the software process where the reuse environment is going to be incorporated into, and because of the advantages that formalization entails: minimization of the ambiguity, incompleteness and inconsistency inherent to natural language; and the possibility of automating the reuse process.

Since we are interested in recovering analogous behaviours to a given one, the retrieval criteria must be based on components' functionality relationships which enable defining software component hierarchies or lattices to classify and retrieve components in a properly way. With this aim we have defined four different functional criteria which define four partial order relations and four equivalence relations among components. Two components related by one of these functional relationships are assumed to have a functional closeness, but it is necessary to assess how closer are, and it is the reason to define functional metrics which enable us to quantify functional differences among components. The set of the four partial order relations and the metrics make up the criteria to manage the repository.

After establishing the lattices of components held by the repository, we must define how the components classification is done, how are the main tasks of the recovering process, how is selected the most suitable component, and how is adapted the selected component to satisfy the functional requirements asked.

Verification information reuse represents the last part of the proposed work. We have summarized all the available verification information —agreement levels on each state of the model— making it easy-to-reuse. We have opted for storing the agreement levels of a property on each trace of the system and over the whole model. Doing this, it is possible to reuse this knowledge in order to know the agreement levels of one property on a model, knowing the agreement levels of the property on other models maintaining a functional relationship with it.

Contenido

PARTE I

Reutilización de software: antecedentes y revisión bibliográfica

CAPÍTULO 1

Introducción

1.1 ¿Qué entendemos por reutilización de software?

La primera vez que se habló formalmente de reutilización en el ámbito de la ingeniería del software fue en el año 1968, cuando McIlroy propuso la creación de fábricas de elementos software análogas a las ya existentes de componentes hardware. Los foros de discusión de aquella época se centraban en localizar alguna solución viable a la llamada *crisis del software*: la demanda de creación y mantenimiento de sistemas software crecía vertiginosamente sin que las empresas del sector fueran capaces de hacerle frente. En este contexto fue donde surgió la propuesta de McIlroy como factor a tener en cuenta en la resolución de dicha crisis.

Desde entonces, y aunque los procesos de obtención de sistemas software han evolucionado notablemente mejorando e incrementando la producción de aplicaciones software, todavía queda mucho para hacer frente a las necesidades de una sociedad cada vez más consumidora y más dependiente del software y, consiguientemente, menos tolerante a sus fallos. La programación automática donde, partiendo de un conjunto de requisitos normalmente vagos, poco coherentes, y puede que incompletos, se genere el código apropiado a las necesidades de un usuario, está todavía bastante lejos (si es que llega a alcanzarse algún día). Ante esta situación, la reutilización de procesos y productos software se divisa como una alternativa realista y técnicamente factible.

Se puede decir que, entre los especialistas del ámbito, es ampliamente aceptado describir la reutilización de software como (Krueger, 1992):

"Software reuse is the process of creating software systems from existing software rather than building them from scratch"

precisamente por la flexibilidad de la definición: que no acota el tipo de elemento a reutilizar, su grado de abstracción o granularidad, y si se realiza algún tipo de modificación en él o no. Sin embargo, también es cierto que otros autores matizan esta descripción. Por ejemplo Lim (Sametinger, 1997) defiende que sólo es correcto hablar de reutilización cuando el software se aplica a los nuevos sistemas sin sufrir ningún tipo de modificación. Otros incluso son mucho más restrictivos, acotando también el tipo de elemento software a reutilizar, así para B. Stroustrup (Stroustrup, 1996) se habla de reutilización cuando un fragmento de código es utilizado literalmente en, al menos, dos programas.

Nosotros nos acogeremos a la definición que hace Krueger (Krueger, 1992) y nos referiremos a la *reutilización de software* como a la aplicación de productos software ya existentes (código, diseños, documentación, especificaciones,...) para la creación de un sistema nuevo, es decir, de la utilización de *"cualquier tipo de información ya existente que necesite el desarrollador de sistemas para la elaboración de un nuevo sistema software"* (Prieto-Díaz, 1993).

La reutilización casual y no sistemática de **recursos materiales** (básicamente código) es contemporánea a la programación; en los últimos años, sin embargo, se están potenciando las incursiones en la reutilización de **recursos humanos** (ideas, diseños, ...) que, aunque más compleja, es mucho más rentable y atractiva.

El objetivo de este capítulo introductorio es el acercamiento a los motivos que aconsejan la reutilización y las razones que históricamente han impedido su expansión, además de una breve introducción a las metodologías existentes.

1.2 Impacto de la reutilización de software

El impacto de la reutilización de software en la industria del sector ha sido analizado en la literatura especializada en repetidas ocasiones. Como nexo a todos los estudios se concluye que la aplicación de un buen programa de reutilización en cualquier organización dedicada a la producción de software ha de revertir en beneficios económicos a medio o largo plazo. Estos beneficios son consecuencia directa de una mejora en la calidad del software generado y del ahorro en tiempo y costes de producción. El hecho de que, hasta el momento, la reutilización no se aplique de forma sistemática, automatizada y formal ha de atribuirse a un conjunto de factores no sólo de origen tecnológico, sino también a factores culturales y económicos.

1.2.1 Reutilización de software y calidad

La reutilización de software incide directamente en la mejora de la mayor parte de los factores que afectan a la calidad de un sistema[1]:

- **Funcionalidad**
  Se puede definir la funcionalidad de un sistema software como el conjunto de prestaciones que ofrece al usuario, así que ésta guarda relación directa con los requisitos iniciales. Es muy habitual que el establecimiento de estos requisitos iniciales precise de una realimentación hacia el usuario, debido a que éste suele tener una idea vaga del sistema en sus primeras fases de diseño. La reutilización proporciona una base para el establecimiento de prototipos rápidos (Sametinger, 1997) de forma que permite al usuario estimar la funcionalidad del sistema y corregirla en las primeras etapas de su ciclo de vida (apartado 2.6.2).

- **Fiabilidad**
  La fiabilidad de un sistema mide la capacidad de que éste mantenga sus prestaciones, bajo una serie de condiciones de aplicación, a lo largo de un período de tiempo. Cuando se reutiliza software, éste ha sido verificado y validado, pudiendo estar en muchos casos avalado por utilizaciones anteriores, mejorando así la fiabilidad del sistema global.

- **Eficiencia**
  La eficiencia de un sistema software relaciona su rendimiento con la cantidad de recursos que consume durante su vida útil, además de los que se han invertido en su creación. Puede que la creación de software reutilizable provoque que la eficiencia se resienta: el software creado particularmente para un entorno suele ser poco eficiente si se aplica en otro distinto, además éste requiere una inversión mayor en tiempo y costes para que sea reutilizable.

- **Capacidad de mantenimiento**
  Es más fácil el mantenimiento de aplicaciones construidas a partir de software reutilizable. Las modificaciones que sea preciso realizar, en muchos casos, puede que ya estén previstas, la documentación está mucho más cuidada y, además, los costes serán compartidos, ya que el mantenimiento se hará centralizado y no en cada aplicación particular.

- **Portabilidad**
  Un sistema software será más portable cuanto más fácil sea su transferencia de un entorno a otro. Como ésta es una de las características de mayor peso a lahora de generar software reutilizable, la portabilidad se verá incrementará en sistemas donde se ha aplicado reutilización.

[1] según la norma de calidad ISO 9126 (ISO, 1991)

De todas estas características se puede deducir que, en general, la reutilización tiene una incidencia beneficiosa en el incremento de la calidad del software generado (Basili et al., 1996). El hecho, por otro lado esperado, de que se decremente la eficiencia no es más que el precio que es necesario pagar, y que es razonable en la mayoría de los casos —salvo en situaciones con graves restricciones temporales o de recursos—, por la obtención de elementos software reutilizables. La mejora de la calidad del software tiene una clara repercusión económica ya que se abaratan los costes de mantenimiento y del proceso de desarrollo (Karlsson, 1996), debido a que:

- se trabaja más rápido;
- se trabaja de forma más eficiente por estar el proceso de desarrollo más controlado (estimaciones más precisas en cuanto a los costes, mejoras en el propio proceso, en la evaluación y mejora de los sistemas, ...); y
- se trabaja de forma más hábil, en cuanto a que se evita la duplicación de trabajo, con software realizado por especialistas en cada área, reduciéndose así el tamaño de los equipos de trabajo y facilitando su interoperabilidad.

1.2.2 Impedimentos a la reutilización de software

A pesar de todos los beneficios que presenta la reutilización para el desarrollo de sistemas software, las organizaciones suelen adoptar una postura reticente ante la posibilidad de aplicarla para la elaboración de sus productos. Las justificaciones son varias pero pueden resumirse en:

➢ **Motivos económicos**
La implantación de un proyecto de reutilización requiere una elevada inversión inicial de la que, además, sólo va a ser posible obtener beneficios a medio y largo plazo. Se necesita financiación para:
  - la obtención de software reutilizable,
  - reutilizar el software obtenido, y
  - definir e implementar el proceso de reutilización.

  Es decir, se necesitan al menos inversiones en infraestructura, metodología y soporte técnico. Además el software generado ha de satisfacer unos requisitos de calidad más restrictivos, encareciéndose así todo el proceso.

➢ **Motivos técnicos**
Hasta el momento no existen herramientas que faciliten o automaticen las tareas propias de un programa de reutilización como pueden ser: la clasificación y búsqueda de elementos software, y la adaptación e integración de dichos elementos en los nuevos sistemas.

- **Motivos de organización**
  La reutilización de software a gran escala afecta a todo su ciclo de vida y requiere una reestructuración de la organización tradicional: puede ser más útil, por ejemplo, la creación de equipos que se ocupen sólo de la generación y mantenimiento de software reutilizable. En esta situación se hace imprescindible una gestión global. La actitud conservadora de los gestores ante estos cambios y el "síndrome NIH" (N*ot* I*nvented* H*ere*), colaboran al retraso en la aplicación de una política de reutilización en las empresas y organizaciones productoras de software.

En otros ámbitos de la ingeniería, sin embargo, la reutilización forma parte inherente del proceso de creación de un sistema. Para el diseño de un sistema electrónico, por ejemplo, el grupo de ingenieros parte de una serie de requisitos y especificaciones. Normalmente, llegarán a una solución de compromiso entre los requisitos pedidos y las características de los componentes electrónicos ya disponibles, de forma que el desarrollo *ad-hoc* se realiza en contadas ocasiones. Sin embargo, la dinámica de trabajo en la industria del software es radicalmente distinta: se parte también de un conjunto de especificaciones, pero una vez que éste es fijado, el trabajo consiste en satisfacerlas y completarlas. Este enfoque favorece la obtención de soluciones particulares y dificulta la reutilización. Así que parece que una de las principales características del software, su adaptabilidad y flexibilidad, es también su principal obstáculo a la hora de generalizar en esta industria una política de reutilización efectiva y eficiente.

1.3 Perspectivas en la reutilización de software

Un programa de reutilización puede ser considerado desde diferentes perspectivas dependiendo de la característica que se analice, Prieto-Díaz identificó seis (Prieto-Díaz, 1993), sin embargo este trabajo se estructurará atendiendo a los cinco puntos que se indican en la tabla 1.1:

1.3.1 Ámbitos de aplicación

La cantidad de software potencialmente reutilizable depende del grado de funcionalidad común que haya entre los sistemas que lo comparten (Sametinger, 1997). Definiendo **dominio** como área de aplicación o campo de desarrollo de sistemas, se entiende que el grado de reutilización es elevado entre sistemas pertenecientes al mismo dominio. La reutilización de software entre sistemas de un mismo ámbito de aplicación recibe el nombre de *reutilización vertical* o dependiente del dominio. La *reutilización horizontal*, o de ámbito general, se establece entre sistemas que no pertenecen al mismo dominio, lo que motiva que ésta sea, en general, más compleja y, consecuentemente, el grado de reutilización mucho menor.

Perspectiva	Ejemplos
Ámbitos	vertical, específica del dominio, horizontal, de propósito general
Grados	sistemática, ad-hoc, individualizada, institucionalizada, planeada, oportunista
Metodología	generación, composición
Integración	adaptación, modificación, sin modificaciones
Elementos	documentación, código, especificaciones, arquitectura, requisitos

Tabla 1.1. Enfoques de la reutilización de software

Tanto en un caso como en el otro, el **análisis de dominio** (capítulo 4) se ha convertido en una etapa fundamental dentro de los nuevos ciclos de vida, con el propósito de identificar, recoger y organizar software para que pueda ser reutilizado (Prieto-Díaz, 1990).

1.3.2 Grado de aplicación

¿Qué actitud adopta una organización ante la reutilización? Las distintas respuestas a esta pregunta establecerán la catalogación de la reutilización atendiendo a cómo aplicarla. Desde esta perspectiva, se puede determinar una clasificación según el grado de madurez que dicha organización haya alcanzado en la inclusión de la reutilización dentro de su proceso software: reutilización sistemática, individual, planificada, oportunista, etc.

El grado de madurez que presenta una organización ante la reutilización está directamente relacionado con el grado de madurez que presente su proceso software, es decir, el conjunto de actividades, métodos y prácticas que se utilicen para el desarrollo y mantenimiento del software (Paulk et al., 1997). Para establecer una categorización del grado de madurez del proceso de reutilización en una organización se ha utilizado como base el modelo CMM (C*apability* M*adurity* M*odel*) (Karlsson, 1996). En el capítulo 5 se detalla este modelo y su extensión.

Además hay que tener en cuenta que la actitud de una organización ante la reutilización se ve condicionada también por otro tipo de factores, como pueden ser factores económicos, legales y de organización.

1.3.3 Metodologías de reutilización

Tradicionalmente se han diferenciado dos metodologías para enfocar la reutilización de software; la primera se basa en la obtención de un nuevo sistema por

composición de elementos ya existentes y la segunda en la **generación** del nuevo sistema utilizando como base una estructura o modelo.

La reutilización por composición es bastante intuitiva, se trata de conjugar elementos o componentes para construir un sistema nuevo. Aunque conceptualmente es sencilla, requiere un soporte técnico complejo como métodos de clasificación y selección optimizados según los componentes, y soporte para su adaptación e integración en el nuevo sistema.

La reutilización por generación es conceptualmente más compleja, ya que no es posible definir los componentes como entes autocontenidos y concretos. En este caso se reutilizan procesos de generación obtenidos como resultado de una *codificación de estructuras*. Se suelen distinguir tres subtipos de metodologías: generadores de aplicación, generadores basados en el lenguaje y sistemas transformacionales.

Aunque en los capítulos 2 y 3 se detallarán ambas metodologías, sólo notar aquí una breve diferencia. La generación de sistemas se ha orientado hacia la reutilización de elementos pertenecientes a las primeras fases del ciclo de vida del software como son diseños, arquitecturas o requisitos, lo que la hace más apetecible en cuanto a su posible rentabilidad pero, desde luego, mucho más difícil de aplicar. Sin embargo, la reutilización por composición fue la primera en utilizarse, centrándose en elementos pertenecientes a las últimas fases del proceso software, como es el caso del código.

1.3.4 Integración de componentes

En el caso de aplicar reutilización por composición, es necesario concretar cómo se pretenden integrar los componentes en el nuevo sistema, para eso será preciso hablar de: visibilidad de componentes y modificación de componentes.

Se denominan *cajas negras* a aquellos componentes de los que sólo se conoce su interfaz y su funcionalidad; como no se tiene acceso a su interior es imposible realizar ningún tipo de modificación, así que en este caso sólo es posible hablar de reutilización íntegra. Aquellos componentes que permiten el acceso a su interior son llamados *cajas blancas*, en ellos es posible realizar modificaciones para adaptar su funcionalidad a la requerida. Un tipo intermedio son las *cajas transparentes* (Sametinger, 1997), este tipo de componentes permiten que se conozca su interior, como en las cajas blancas, pero su modificación no está permitida, como en el caso de las cajas negras.

La reutilización de cajas negras, aunque mucho más difícil, presenta muchas más ventajas ya que se incrementa la calidad y la fiabilidad del software generado, los componentes de este tipo han sido verificados y suelen estar certificados (Dunn y Knight, 1993; Knight y Dunn, 1998). Sin embargo, cuando se reutilizan cajas blancas es muy probable que se realicen modificaciones que provoquen que haya que repetir las tareas de verificación, decrementándose su fiabilidad.

La reutilización por generación puede ser vista como un tipo de reutilización de cajas negras, ya que, aunque no se dispone del componente en sí, se tiene un programa generador y éste es reutilizado como si fuera una caja negra.

1.3.5 Elementos reutilizables

Según la definición que aquí se ha adoptado para la reutilización de software, toda información necesaria para el diseño y desarrollo de un sistema se considerará potencialmente reutilizable. Así se tendrán tanto código como documentos, diseños, especificaciones, textos, etc. Las diferencias entre la reutilización de unos u otros radica, principalmente, en su grado de abstracción y la granularidad del componente, ambas características son fuente de controversia a la hora de decidir qué reutilizar (Prieto-Díaz, 1985; Prieto-Díaz, 1996). Es necesario adoptar una posición de compromiso entre los beneficios que aporta la reutilización de determinado elemento, directamente proporcionales al nivel de abstracción y a su tamaño, y la facilidad con que éste es reutilizado, inversamente proporcional a los mismos factores.

1.4 Conclusión

El objetivo de esta primera parte de revisión bibliográfica es la ubicación de los procesos de reutilización dentro de la ingeniería del software, estudiar sus diferentes metodologías —analizando cómo afectan a los ciclos de vida más tradicionales—, y evaluar su impacto: análisis de beneficios y costes. Así, esta primera parte se ha organizado de la siguiente forma:

– en el capítulo 2 se presenta la metodología de reutilización por composición de elementos reutilizables, introduciendo también sus principales problemas, como la clasificación y recuperación de componentes en bibliotecas organizadas;

– en el capítulo 3 se trata la reutilización por generación, comentando los tres paradigmas insignia de esta metodología: generación de aplicaciones, compiladores de sistemas y entornos transformacionales;

– el análisis de los dominios de aplicación (ingeniería de dominio) ha cobrado fuerza en los últimos años y, de hecho, hoy en día no se plantea la posibilidad de afrontar un programa de reutilización sistemático sin una fase de análisis de los ámbitos de aplicación. En el capítulo 4 se comentarán diferentes metodologías de ingeniería de dominio y su impacto en los ciclos de vida;

– desde el punto de vista industrial no se afronta la reutilización de software como una alternativa real a los problemas de producción y calidad de las

aplicaciones. Parte de este escepticismo está promovido por la carencia de métodos evaluadores de los beneficios y costes de los programas de reutilización, así que últimamente una gran parte de la comunidad *reutilizadora* enfoca su trabajo en este sentido. El objetivo es la consecución de métodos fiables que estimen el impacto económico de la reutilización y que proporcionen pautas para la inclusión de programas de reutilización dentro de un proceso software. Todos estos temas serán tratados en el capítulo 5.

CAPÍTULO 2

Reutilización por composición

2.1 Introducción

El desarrollo de sistemas por composición de elementos es una metodología bastante intuitiva y, quizá por eso, fue la primera en la que se basaron los procesos de reutilización. Sin embargo, su aplicación reúne conceptos de implementación complejos, como son la definición y obtención de los componentes; y la definición y mantenimiento de sistemas de clasificación y recuperación eficientes.

La reutilización por composición consta de dos actividades complementarias: la **obtención de elementos reutilizables** y el **desarrollo con componentes reutilizables**. El nexo entre ambas es la **biblioteca de componentes** donde éstos son almacenados, según un criterio de clasificación, para ser recuperados posteriormente.

Este capítulo está estructurado de la siguiente forma: en el apartado 2.2 se definirá qué puede entenderse por elemento software reutilizable, posteriormente se hablará sobre diferentes técnicas para la obtención de dichos elementos reutilizables (apartado 2.3), cómo certificar su calidad (apartado 2.4), y diferentes formas de clasificarlos y catalogarlos para su posterior recuperación (apartado 2.5). Por último, se comentará la influencia que tiene la reutilización de componentes en el proceso software y cómo se refleja en el ciclo de vida (apartado 2.6).

2.2 Elemento software reutilizable

Para ser coherentes con la definición que hemos adoptado para el proceso de reutilización de software (apartado 1.1), habrá que considerar que **elemento software** es cualquier tipo de información que se necesite a lo largo del ciclo de vida de un sistema. Así se incluirá, no sólo código (Sametin-

ger, 1997) (funciones, clases, objetos, etc), sino también especificaciones, requisitos, diseños, documentación, ... Pero no todos los elementos software van a poder ser reutilizados, para esto han de satisfacer las condiciones siguientes:

"Un elemento software reutilizable ha de ser autocontenido, describiendo una funcionalidad clara y documentada, proporcionar una interfaz apropiada y un registro o estado de reutilización"

El hecho de que sea autocontenido implica que tenga la menor dependencia posible de factores externos, tanto si éstos son de entorno como de implementación. El cumplimiento de esta característica facilita la *transferencia* y la *adición* del componente a otros sistemas, en el mismo o diferente ámbito de aplicación, sin provocar *efectos secundarios*. Otra característica deseable, aunque no imprescindible, es la **formalización** del elemento, es decir, si se define utilizando alguna técnica de descripción formal o semiformal, además de una definición unívoca, proporciona una base útil para la realización de tareas de *verificación* e incluso para la automatización de las tareas de clasificación y búsqueda (Ouyang y Carver, 1997)(apartado 2.5.1).

2.2.1 Modelo de un componente: proyecto REBOOT

Aunque existen tantos modelos de componente reutilizable como de entornos de reutilización, en este apartado vamos a definir el que podría considerarse el componente reutilizable ideal, es decir, aquél que reúne todas las características deseables que facilitan las tareas propias de un entorno de desarrollo basado en reutilización. Es por ello que nos hemos centrado en el modelo de componente definido dentro del proyecto REBOOT (RE*use* B*ased on* O*bject*-O*riented* T*echniques*, figura 2.1). Este proyecto —enmarcado dentro del SER ESPRIT PROJECT 9809 (SER, 1996)— trata de definir un marco metodológico y organizador que facilite la implantación de programas de reutilización en organizaciones o empresas software. El modelo de componente propuesto dentro de este proyecto tiene asociada, implícita o explícitamente, la siguiente información:

- Una representación interna o **información de clasificación** para que sea posible su inclusión en una biblioteca de componentes y su localización posterior.
- **Información de calificación** que garantiza su calidad y evalúa su grado de reutilización.
- **Documentación** que incluye información de dos tipos, documentación descriptiva del componente que ayudará a su posterior reutilización y documentación asociada al sistema donde éste será incluido. La documentación

se hace imprescindible para evaluar el componente más adecuado a los intereses de la aplicación, comprender su funcionalidad y permitir su adaptación a los nuevos sistemas. Algunas técnicas de clasificación y búsqueda automáticas (apartado 2.5.1) la utilizan como soporte para establecer las consultas y el almacenamiento de elementos en la biblioteca.
Se puede englobar aquí la **información de administración**, que incluye datos como el nombre del creador del componente, del gestor actual, fecha de creación, etc. Es especialmente útil, sobre todo si hablamos de reutilización *interempresarial*, incluir datos sobre los precios y permisos de acceso a los componentes.

➢ La fiabilidad en un componente aumenta si se tiene información sobre las **pruebas** que se han realizado y sus resultados, de forma que estos datos se hacen imprescindibles.

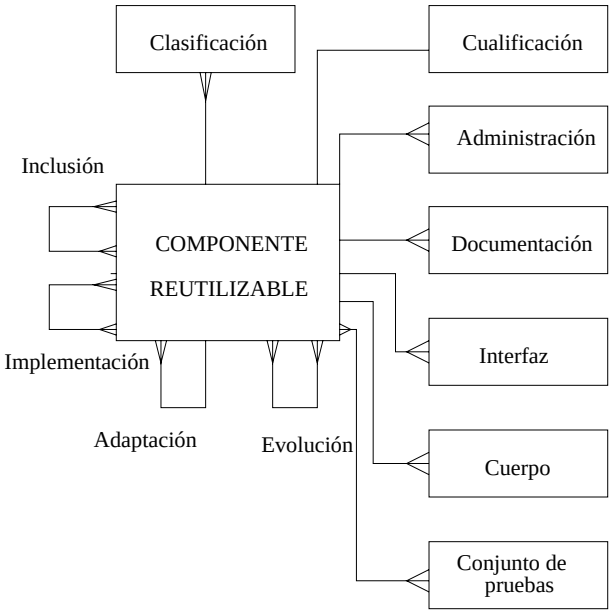


Figura 2.1. Modelo REBOOT de un componente reutilizable (notación entidad-relación)

Entre los componentes almacenados en la biblioteca se pueden establecer cuatro tipos de relaciones:

➢ **Relación de implementación**
Se establece entre aquellos componentes que tienen la misma funcionalidad pero que se encuentran en una fase de desarrollo diferente: análisis, diseño y codificación.

- **Relación de inclusión**
  "La reutilización de un objeto implica, implícitamente, la reutilización concurrente de un conjunto de elementos relacionados con él" (Caldiera y Basili, 1991). Esta situación es típica, sobre todo, para elementos de elevada granularidad, normalmente la relación de inclusión se establece entre éstos y aquéllos que los componen.

- **Relación de adaptación**
  Enlaza elementos que han sufrido alguna modificación (para que sus requisitos se ajusten a los solicitados) con aquel elemento *padre* del que proceden.

- **Relación de evolución**
  Proporciona un histórico de las sucesivas versiones del mismo componente, incluyendo las funcionalidades que le han sido añadidas o eliminadas.

2.3 Obtención de elementos reutilizables

¿Cómo obtener elementos software que puedan ser reutilizados para el desarrollo de un nuevo sistema?

El enfoque de esta actividad puede hacerse de dos formas radicalmente opuestas: bien partiendo de sistemas software ya existentes o bien generando elementos con la intención de que éstos sean reutilizados.

La reutilización esporádica, no planificada ni automatizada, era practicada por los programadores, a nivel de código, extrayendo los componentes de sistemas antiguos. Este proceso es imposible de automatizar porque estos sistemas no han sido obtenidos siguiendo las pautas de ningún proceso de reutilización (Prieto-Díaz, 1996).

Hoy día, dentro de la **ingeniería inversa**, se han establecido procesos de identificación y calificación de elementos partiendo de estos sistemas antiguos. Tanto estos procesos, como los de generación de componentes, pueden ser automatizados, consiguiendo que la reutilización pueda llegar a realizarse sistemáticamente.

De lo dicho antes, se deduce que la obtención de elementos reutilizables no debe ser considerada como una fase o etapa dentro del proceso software, sino que ha de ser vista como una actividad paralela al ciclo de vida.

2.3.1 Ingeniería inversa y reutilización

La ingeniería inversa trata de la reconstrucción de elementos propios de las primeras fases del ciclo de vida (especificaciones, requisitos, etc) partiendo de

elementos propios de las últimas fases, por ejemplo código. Su actividad proporciona las bases para lograr: la representación de un sistema en otro nivel de abstracción mucho mayor y la identificación de componentes, caracterizando las relaciones entre ellos. Esta última línea de trabajo es la que nos interesa desde el punto de vista de la reutilización.

El proceso de obtención de componentes en un sistema ya creado consta de dos pasos: la identificación y la calificación de elementos (Caldiera y Basili, 1991).

1. *Identificación:*
   Durante esta fase, se extraen aquellas unidades o módulos que sean potencialmente reutilizables. Para que el proceso pueda automatizarse habrá que definir un conjunto de atributos o características de los elementos (modelo) que, normalmente, se establecen en función de los resultados obtenidos tras la aplicación de un conjunto de métricas que evalúan su potencial de reutilización.

   Si el elemento supera con éxito la aplicación de dichas métricas, entonces se puede pasar a la siguiente etapa.

2. *Calificación:*
   En esta fase, se realiza un estudio de cada unidad para comprender su funcionalidad, documentarla y representarla formalmente. Una vez hecho ésto habrá que agregar toda esta información al componente para que su reutilización sea viable y almacenar el elemento en la biblioteca de componentes. Esta actividad puede dividirse en las etapas siguientes:

   (a) Obtención de una especificación funcional: partiendo de la documentación asociada y del código fuente se analiza el comportamiento del elemento y se especifica formalmente.

   (b) Generación de un conjunto de pruebas: utilizando la especificación obtenida se generarán un conjunto de pruebas para comprobar la corrección del elemento (apartado 2.4).

   (c) Inclusión de documentación: cada elemento adjuntará información sobre sus características, cómo reutilizarlo e información de administración.

   (d) Clasificación y almacenamiento: cada elemento se caracterizará de forma unívoca y será almacenado en una biblioteca de componentes para su posterior recuperación.

2.3.2 Desarrollo de componentes reutilizables

De forma independiente y complementaria a la aplicación de ingeniería inversa, se puede aplicar alguna metodología que permita la creación de componentes

desarrollados de forma que sea posible su reutilización posterior. Así, éstos habrán de obtenerse reuniendo las características enunciadas en el apartado 2.2.

¿Cómo surge la iniciativa de generar un componente que implemente una funcionalidad determinada y que éste, además, sea reutilizable?

Esta necesidad puede surgir bien durante el desarrollo de un sistema o bien tras el estudio, realizado por un analista de dominio, de las necesidades potenciales en un campo de aplicación determinado (capítulo 4). Independientemente de la motivación, se parte de un conjunto de requisitos funcionales que habrá que satisfacer y de un modelo de componente preestablecido que hay que respetar para que el programa de reutilización sea satisfactorio.

Las diferentes metodologías que pueden ser adoptadas presentan las siguientes etapas en común:

1. Estudio de las **diferentes soluciones** que pueden ser adoptadas para generar dicho componente. Dentro de ellas, se optará por la que sea más fácil de comprender y validar, facilitando así su reutilización.

2. Identificar posibles funcionalidades que se puedan añadir, estudiando su impacto económico y su potencial de reutilización.

3. Se realizará un estudio de posibles generalizaciones del componente y una tarea de **abstracción** que permita elaborar una interfaz adecuada.

Posteriormente, se obtendrán todas las características definidas en el modelo de componente reutilizable (documentación, información para su clasificación, batería de pruebas, etc).

2.3.3 Propuestas para la obtención de elementos reutilizables

Basili y Caldiera (Caldiera y Basili, 1991) proponen un proceso de obtención de componentes mixto, es decir, en la biblioteca habrá elementos desarrollados con el objetivo de ser reutilizados y otros que son el resultado de adaptar aquellos identificados en sistemas ya existentes.

En la figura 2.2 se esquematiza el proceso propuesto. La fábrica de componentes se encarga de la obtención de los elementos que se necesitan para el desarrollo de un sistema nuevo. Esta actividad se realiza de dos formas distintas con dos submódulos diferentes: uno síncrono y otro asíncrono. El primero se encarga de la generación de componentes y el segundo realiza las tareas de ingeniería inversa.

Karlsson (Karlsson, 1996) propone un proceso, también mixto, de obtención de elementos que se esquematiza en la figura 2.3. Del análisis realizado sobre el dominio de aplicación se extraen elementos de sistemas ya existentes y se generan otros según las características y necesidades de dicho ámbito de aplicación.

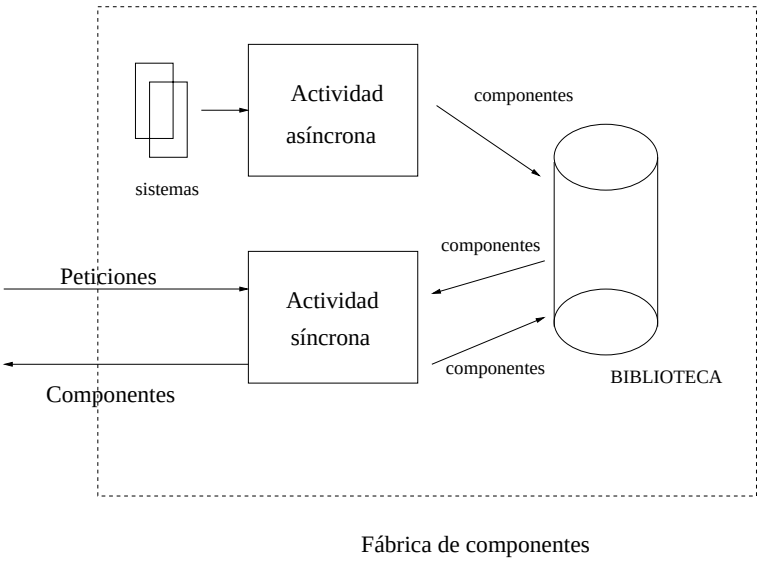


Figura 2.2. Modelo de la fábrica de componentes

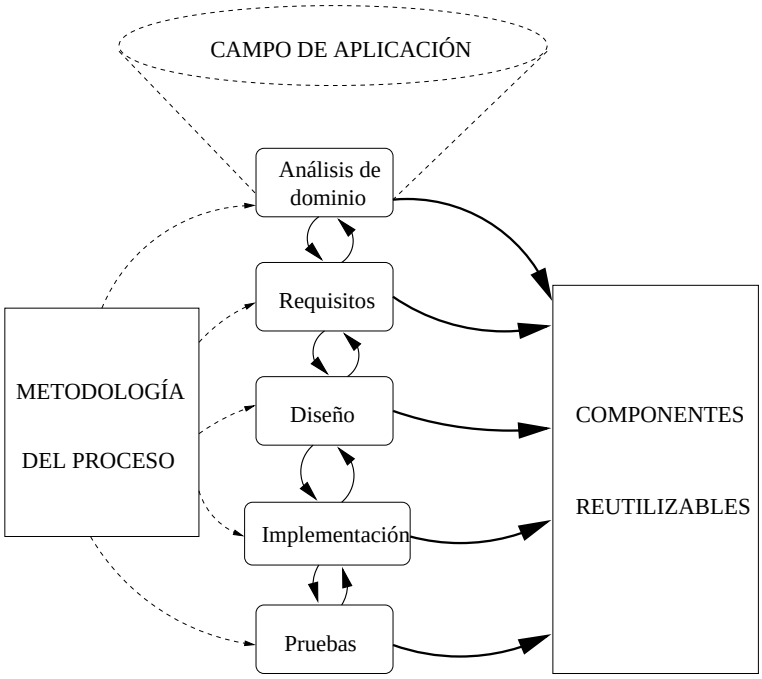


Figura 2.3. Ciclo de vida de un proceso de obtención de componentes, modificación de la propuesta de Karlsson (Karlsson, 1996)

La diferencia fundamental de ambos procesos radica en la generación de elementos. Según el modelo de la fábrica de componentes (Caldiera y Basili) éstos se generan bajo demanda, es decir, sólo cuando son necesarios para la creación de un nuevo sistema. En el caso del modelo propuesto por Karlsson, la generación de componentes se realiza independientemente de que éstos sean demandados para la obtención de un sistema nuevo o no. Es decir, con los resultados obtenidos tras el análisis del ámbito de aplicación, se detectan un conjunto de necesidades potenciales y éstas se cubren creando nuevos elementos.

2.4 Certificación de componentes

Antes de almacenar los componentes obtenidos, según alguno de los procesos definidos en el apartado 2.3, es necesario ofrecer una garantía de que dichos elementos son *fiables*, es decir, que realmente cumplen unos requisitos mínimos de calidad. Si un componente fue obtenido siguiendo unas normas o guías de desarrollo que garanticen su calidad, es mucho más fácil verificar la calidad del sistema obtenido utilizando componentes de este tipo.

Es difícil establecer los criterios que lleven a la obtención de componentes de calidad o certificados (Dunn y Knight, 1993):

– un exceso de celo en este sentido revertirá en un incremento de tiempo innecesario en la fase de obtención de componentes que elevará su coste sin que las mejoras sean perceptibles, y

– si el componente no está totalmente comprobado, se corre el riesgo de reducir su fiabilidad y no disfrutar de todos los beneficios asociados a la reutilización.

Existen diferentes técnicas para certificar componentes:

➢ **Certificación por niveles**
Este método establece la definición de un conjunto de niveles, cada uno satisfaciendo una serie de requisitos de calidad, de forma que cada componente reutilizable será evaluado según estos criterios y le será asignado un nivel de certificación. La dificultad de esta técnica es el establecimiento de los criterios de evaluación que definirán los niveles: éstos han de ser tal que permitan la catalogación de componentes procedentes de cualquier tipo de sistema y, al mismo tiempo, estar lo suficientemente definidos, de forma que, conociendo el nivel de certificación de un componente, esté claro qué calidad se puede esperar de él. Una de la bibliotecas que utiliza este tipo de certificación es ASSET (A*sset* S*ource for* S*oftware* E*ngineering* T*echnology*).

➢ **Modelos de certificación estadísticos**
Esta técnica se centra en la evaluación de la *fiabilidad* de un componente. Es necesario definir un modelo del sistema (componente): estados de comportamiento y las probabilidades de transición entre estados (modelo de Markov). De esta manera se podrá estimar qué trazas son más probables y, consecuentemente, cuáles precisarán ser más fiables. La certificación estadística tiene una aplicabilidad reducida y, además, no considera otros aspectos relacionados con la calidad, como por ejemplo la eficiencia.

➢ **Modelos de certificación local**
En el marco del proyecto RESOLVE (Weide, 1999; Weide y Hollingsworth, 1994) se ha introducido el concepto de certificación local o certificación independiente del contexto: se verifica la corrección de cada elemento sin tener en cuenta su relación con otros, de esta manera se logra facilitar las tareas de verificación para la composición de elementos. Las técnicas formales son el fundamento de este proyecto ya que son utilizadas tanto para la especificación de componentes como para su verificación.

➢ **Certificación orientada al dominio de aplicación**
Este tipo de certificación, introducida por Knight y Dunn (Knight y Dunn, 1998), se basa en la dependencia que tienen los componentes del entorno de trabajo donde se enmarquen. Es decir, los requisitos de calidad de cada componente son fuertemente dependientes de los requisitos de calidad de las aplicaciones, de las especificaciones, diseños, documentación, ... La implantación de este tipo de certificación requiere de los siguientes pasos:

1. *Determinar los requisitos de calidad*: es necesario detallar las propiedades y características deseables en los elementos del dominio (especificaciones, diseños, etc.).
2. *Definir la instancia de certificación*. Una instancia de certificación constará del conjunto de propiedades que ha de tener un componente y de las técnicas utilizadas para demostrar que las satisface.
3. *Desarrollo de una biblioteca*. La biblioteca de componentes ha de ser creada de forma que satisfaga los requisitos de calidad pedidos, es decir, ha de ser certificada.
4. Emplear este tipo de certificación para obtener aplicaciones obtenidas usando componentes reutilizables.

Existe un conjunto de normas y modelos más generales para evaluar software como la norma ISO 9126 (ISO, 1991) o el modelo CMM (en el apartado 5.3.1 se realiza una breve introducción a este último).

La norma internacional ISO 9126 —*Software Products Evaluation Quality Characteristics and Guidelines for their Use*— nace en el año 1991 como un intento de unificar criterios y de establecer una serie de pautas que garanticen la

obtención de software de calidad. La norma incide en un conjunto de características que afectan a la capacidad del software para satisfacer las necesidades o requisitos impuestos: funcionalidad, fiabilidad, eficiencia, capacidad de mantenimiento, portabilidad y capacidad de ser utilizado. El modelo CMM establece una escala para evaluar la madurez de los procesos software y, consecuentemente, la de la calidad del software obtenido.

2.5 Bibliotecas de componentes reutilizables

Las bibliotecas de componentes surgen como respuesta a la necesidad de organizar y almacenar los componentes reutilizables para su posterior recuperación. Existen diversidad de bibliotecas, creadas en entornos académicos, comerciales y gubernamentales, y con diferentes formas de acceso a sus contenidos: libremente o tras la obtención de una licencia. Atendiendo al tipo de elementos que contenga, se puede hacer una clasificación general de las bibliotecas de componentes (Sametinger, 1997):

- Las bibliotecas **locales** suelen almacenar componentes de propósito general, comunes para diferentes aplicaciones.
- Las bibliotecas **específicas de un dominio** contienen, como su nombre indica, componentes propios de aplicaciones pertenecientes al mismo ámbito.
- Las bibliotecas de **referencias** suelen utilizarse como páginas amarillas. No contienen componentes, en su lugar mantienen referencias a las localizaciones de dichos elementos.

Un aspecto que está cobrando importancia es la **cooperación** entre bibliotecas (Browne et al., 1996), de este modo podría reducirse la redundancia en el almacenamiento y aumentar al mismo tiempo la efectividad en las búsquedas. Para que esta colaboración sea posible es necesaria la definición de un estándar que regule el flujo de información entre bibliotecas y que normalice los tipos de datos almacenados y los catálogos de contenidos en cada biblioteca. Con este objetivo surgió en el año 1991 el grupo RIG (R*euse Library* I*nteroperability* G*roup*) formado por grupos de investigación pertenecientes a entornos académicos, privados y estatales. Su propósito es la elaboración de un estándar que defina la cooperación entre bibliotecas y los tipos de datos que se utilizarán para ello. En diciembre del año 1995 el IEEE aprobó el primer estándar, 1420.1, del RIG: el BIDM (B*asic* I*nteroperability* D*ata* M*odel*).

2.5.1 Clasificación de componentes

Clasificación es la actividad consistente en la agrupación de elementos similares, de forma que todos aquellos que pertenecen al mismo grupo tienen, al menos,

una característica en común que no comparten con el resto de los elementos. El criterio de clasificación es crucial para la recuperación de los componentes almacenados en la base de datos, sobre todo cuando su número es elevado.

¿Qué características son deseables en un criterio de clasificación de componentes?

- La **efectividad** es uno de los requisitos imprescindibles, de forma que las búsquedas han de tener (Frants et al., 1997; Mili et al., 1997):
  - elevada capacidad para rechazar la información no relevante: alta proporción de valores significativos dentro de los valores significativos disponibles en la biblioteca (*recall*),
  - elevada capacidad para encontrar información relevante: alta proporción de valores significativos dentro de los valores obtenidos tras la búsqueda (*precision*), y
  - tiempo de respuesta bajo: relación entre el número de componentes inspeccionados en la búsqueda y el número de componentes disponibles en la biblioteca (*response time*).

- Es imprescindible un criterio **eficiente**, de forma que el tiempo de localización sea bajo. Mili et al. (Mili et al., 1998) citan una posible catalogación de esta eficiencia en función de la complejidad temporal de la búsqueda, evaluando el orden de pasos de computación requeridos para resolverla en función de su tamaño:
  1. Muy bajo: si es constante.
  2. Bajo: si sigue una función lineal.
  3. Medio: si se puede expresar como un polinomio.
  4. Alto: si la complejidad temporal se puede expresar como un crecimiento exponencial con el tamaño de la búsqueda.
  5. Muy elevado: si no existe cota superior a la complejidad.

- La **flexibilidad** es otra de las características esperables en un criterio de clasificación, de esta manera la inclusión o eliminación de uno o varios conjuntos no provocará la modificación de todo el esquema de organización.

- Es deseable que sea **fácil** de comprender y de utilizar para que las tareas de clasificación y búsqueda no sean excesivamente complejas. Mili et al. (Mili et al., 1998) establecen una métrica para evaluar la complejidad lógica de la búsqueda clasificándola en función de la composición lógica requerida para resolverla:
  1. Simple: si se puede expresar como un booleano.
  2. Baja: si se puede resolver como una composición de booleanos.

3. Media: si la resolución se reduce a una expresión lógica (predicado).
4. Elevada: si se puede expresar como una composición de predicados lógicos.
5. Muy elevada: si es necesario expresarla como un predicado lógico de segundo orden.

➢ También es interesante que sea posible la **automatización** del proceso, reduciendo al máximo posible la presencia de personal especialmente formado para realizar las tareas de localización y almacenamiento, es decir, se debería mejorar la **transparencia** del algoritmo de recuperación.

Los sistemas de clasificación de componentes son muy variados, desde aquellos que emulan las técnicas más tradicionales utilizadas en la gestión de bibliotecas (Prieto-Díaz, 1985), hasta aquellos que se basan en tecnologías más recientes como la inteligencia artificial (Ouyang y Carver, 1997) o el hipertexto. Existen diversas catalogaciones de estos sistemas de clasificación, realizadas por autores distintos y atendiendo a características diferentes (Arnold y Frakes, 1992; Sametinger, 1997; Karlsson, 1996; Mili et al., 1997). Aquí utilizaremos la clasificación siguiente:

- Basadas en técnicas bibliotecarias
  - Vocabulario acotado
    - Enumerativa
    - Facetada
    - Palabras clave
  - Vocabulario no acotado
    - Texto libre
    - Palabras clave
- Basadas en técnicas de inteligencia artificial
  - Algoritmos divisores
  - Algoritmos aglutinadores
- Basadas en técnicas de especificación formal
- Basadas en hipertexto

Es posible la combinación de estas técnicas para lograr un criterio adaptado a las características de los componentes de cada biblioteca; Atkinson (Atkinson, 1995; Atkinson, 1996) ha estado trabajando en este tema para integrar y unificar diferentes técnicas de clasificación (utilizando ficheros índice, cotejo de estructuras, clasificación facetada, ...).

Sistemas basados en el análisis de conjuntos

El análisis de conjuntos (*clusters*) suele incluirse dentro del área de la inteligencia artificial. Los algoritmos de clasificación basados en el análisis de conjuntos agrupan el grueso de elementos iniciales según su similitud. Es decir, cada elemento tiene asociado un valor que indica su parecido con cada uno de los demás componentes. Los valores de similitud se asignan según un criterio preestablecido, en el caso que nos ocupa podrían ser parecidos en funcionalidad.

Los sistemas de clasificación que se obtienen son muy flexibles, ya que no se realiza ninguna suposición inicial sobre el número de grupos resultante, éste es uno de los valores que se obtiene tras la aplicación del algoritmo. Tradicionalmente estos algoritmos de clasificación se engloban en dos clases (Everitt, 1986):

- *Algoritmos divisores.* Éstos parten de un único conjunto formado por todos los elementos, tras la realización de divisiones sucesivas e iterativas se obtienen los grupos resultantes.
- *Algoritmos aglutinadores.* El enfoque de estos algoritmos es radicalmente opuesto al anterior, se parte de tantos conjuntos como elementos se tengan (cada conjunto consta de un único elemento) y la clasificación se realiza iterativamente aglutinándolos hasta obtener los grupos resultantes.

Las condiciones de parada que se pueden imponer a los algoritmos aglutinadores (número de conjuntos, número de elementos por grupo, valor de similitud) los hacen mucho más flexibles frente a los algoritmos divisores. Los trabajos propuestos por Jung-Jang Jeng y Betty H. C. Cheng (Jeng y Cheng, 1993) se realizaron partiendo de un algoritmo de este tipo, algoritmo de Kruskal; de la misma forma se enfocaron los propuestos por Youwen Ouyang y Doris L. Carver (Ouyang y Carver, 1997) aunque en este caso el algoritmo seleccionado fue el HAC (H*ierarchical* A*gglomerative* C*lustering*).

Sistemas basados en técnicas bibliotecarias

La agrupación de componentes en la biblioteca se hará atendiendo a sus características, éstas pueden describirse en un documento agregado a cada uno de los elementos en su fase de desarrollo, o bien pueden extraerse de la documentación asociada. En cualquiera de los dos casos, se obtendrá una representación interna de los componentes (índices de clasificación) que será utilizada para las tareas de clasificación y búsqueda. La forma de obtener estos índices marcará la diferencia entre las técnicas de clasificación.

➢ **Texto libre**
En este caso, la representación interna del elemento se obtiene partiendo de su documentación asociada. Se realiza un procesado del texto, extrayendo

información léxica, semántica y sintáctica, obteniéndose así los índices de clasificación. Este mismo procesado se aplicará a la descripción que realice el usuario del elemento que desea localizar para obtener los patrones de búsqueda. Un ejemplo de un sistema de clasificación basado en texto libre es ROSA (R*euse* O*f* S*oftware* A*rtifacts*), propuesto por Rosario Girardi (Girardi y Ibrahim, 1994).

Como ventajas presenta una gran flexibilidad en las descripciones (no se tiene vocabulario acotado) y la posibilidad de automatizar el proceso. Como desventajas una efectividad reducida y baja eficiencia, derivada precisamente de la flexibilidad y ambigüedad del lenguaje natural.

➢ **Palabras clave**

En este caso la representación interna de los componentes y los patrones de búsqueda consiste en un conjunto de términos que describen al componente. La elección de estos términos puede que esté restringida (vocabulario acotado) o no (vocabulario libre). Las labores de clasificación y localización se facilitan, ya que se reducen a un proceso basado en el cotejo de términos.

Estos sistemas suelen ser menos complejos y más eficientes, si bien la precisión se ve reducida en el caso de que los términos no pertenezcan a un conjunto preestablecido. La selección de las claves ha de ser realizada por personal cualificado, lo que impide la total automatización de los procesos.

➢ **Clasificación enumerativa**

Muy utilizada en la organización de bibliotecas, se basa en el establecimiento de un conjunto de categorías, y sucesivas subcategorías, dividiendo el espacio de clasificación en regiones. Los elementos pertenecientes a una de estas regiones tendrán unas características comunes predefinidas.

La clasificación enumerativa es conceptualmente sencilla y de fácil implementación. Para que el sistema sea eficiente se requiere que el usuario conozca el esquema de organización. La jerarquía de regiones ha de definirse *a priori*, resultando esquemas inflexibles que no permiten las modificaciones sin una reestructuración global del sistema. En este tipo de sistemas, el diseñador tendrá que tomar una decisión de compromiso entre el número de grupos y el de componentes por grupo.

➢ **Clasificación por facetas**

La organización por facetas fue propuesta por primera vez como aplicación a las bibliotecas de componentes por Prieto-Díaz (Prieto-Díaz, 1985). En este caso, el espacio de clasificación se ordena bajo una serie de facetas o puntos de vista, de forma que cada una de estas dimensiones se define según un número prefijado de categorías. De esta manera, cada elemento tendrá como representación interna un valor (categoría) por cada una de las facetas existentes en el sistema.

Es deseable que las facetas sean lo más ortogonales posible de forma que el valor que presente un elemento en una de ellas sea independiente de los valores que tome en las otras. De esta forma, además, se gana en flexibilidad ya que la modificación de una faceta es independiente de todas las demás. Al igual que ocurría en la clasificación enumerativa, la eficiencia en la recuperación es directamente dependiente del grado de conocimiento que tenga el usuario del esquema de clasificación.

2.6 Impacto en el ciclo de vida del software

¿Cómo recuperar los elementos de la biblioteca de componentes y utilizarlos en el desarrollo de un sistema nuevo?

Las actividades asociadas a este proceso, al contrario de lo que ocurría con la obtención de componentes reutilizables, sí han de integrarse en el ciclo de vida del sistema. En la figura 2.4 se recoge el modelo propuesto por Karlsson, quien propone una forma de incluir estas actividades dentro del tradicional proceso software en cascada, permitiendo la reutilización de componentes en cualquiera de sus fases.

Independientemente de cuál sea el ciclo de vida adoptado por una organización, la metodología consta de las siguientes pautas comunes:

1. Tras la obtención de las unidades necesarias para el desarrollo de un sistema, habrá que **identificar** aquellas que pueden implementarse a partir de elementos ya existentes y almacenados en la biblioteca de componentes.

2. Una vez definidas las características del elemento buscado, se establece un **patrón de búsqueda** para su localización en la biblioteca de componentes. Es muy posible que se obtenga más de un elemento que se ajuste a las características pedidas.

3. Para poder tomar una decisión sobre cuál de los elementos disponibles se va a seleccionar para ser integrado en el nuevo sistema, es necesario llegar a **comprender** las características de cada uno. Para ello se estudian los aspectos **funcionales** y **no funcionales** de cada componente.

   Las características no funcionales de un componente pueden tener tanto o más peso que aquellas referentes a su funcionalidad, ya que establecen un conjunto de condicionantes o restricciones de aplicación. Sin éstas se podría llegar a una implementación correcta desde un punto de vista funcional, pero inservible para las condiciones de entorno de la aplicación. Los aspectos que se suelen tener en cuenta al definir las características no funcionales de un componente, son su eficiencia, capacidad de reutilización, facilidad de mantenimiento y fiabilidad (Franch, 1998; Franch y Botella, 1998).

4. Una vez conocidos todos los elementos disponibles se procede a **evaluar** las posibilidades de cada uno. Puede ser útil que, llegados a este punto, se establezca una **negociación** de requisitos con el usuario, ya que puede que éste decida sacrificar algunos aspectos funcionales para lograr un abaratamiento de los costes de producción.

5. Se **seleccionará** el mejor candidato atendiendo a la evaluación realizada.

6. Posteriormente se procederá a la **adaptación e integración** de dicho componente en el nuevo sistema. En el apartado 1.3.4 se han introducido diferentes formas de hacerlo.

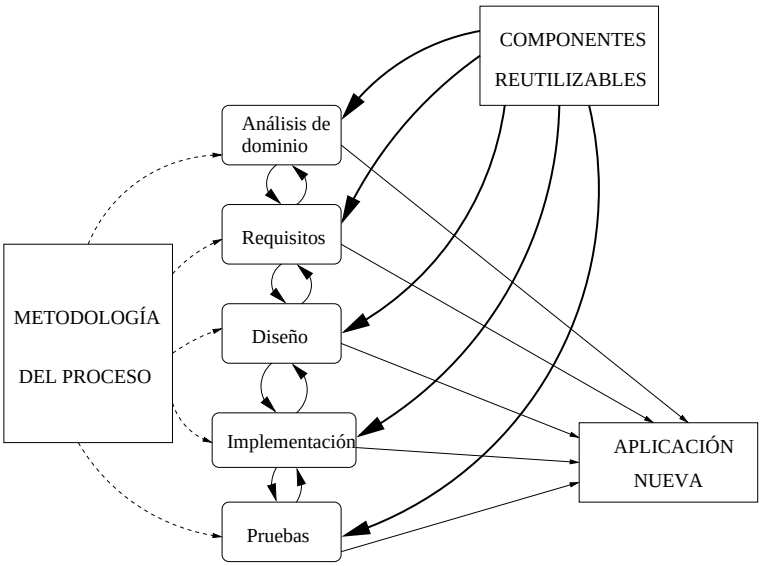


Figura 2.4. Ciclo de vida general del desarrollo con componentes reutilizables, modificación de la propuesta de Karlsson (Karlsson, 1996)

2.6.1 Inclusión de componentes abstractos

La reutilización de componentes de elevado nivel de abstracción provoca una disyuntiva a la hora de definir un programa de reutilización (Karlsson, 1996).

Un componente de este tipo tendrá un conjunto de relaciones de *implementación* (apartado 2.2) de forma que se enlazan componentes con la misma funcionalidad que sólo se diferencian por la fase de desarrollo en la que se encuentran: requisitos, diseño o implementación. Así la evolución del elemento desde su especificación hasta su implementación viene dada por sucesivas relaciones de im-

plementación entre componentes reutilizables. Para integrar estos componentes en un nuevo sistema se puede optar por dos vías alternativas:

- ➢ A medida que el proceso software avanza, independientemente del modelo de ciclo de vida seguido, se selecciona la fase adecuada del componente y se integra en el sistema. Como ventaja notar que la integración del componente se realiza siempre en un contexto de desarrollo coherente con su estado. La principal desventaja radica en la necesidad de repetir algunas actividades en todas y cada una de las fases, como por ejemplo, la comprensión sobre la funcionalidad y comportamiento del componente.

- ➢ Realizar la adaptación del componente de golpe, es decir, adaptar todas sus fases a la vez. Como principal ventaja se observa la eliminación de repeticiones de tareas que se producían en el caso anterior y, como inconveniente, el incremento de dificultad a la hora de integrar el elemento en las fases a las que el sistema en desarrollo todavía no ha llegado.

Quizá la mejor solución sea un desarrollo mixto donde se combinen ambas filosofías:

- un desarrollo *top-down* donde, en cada fase, se realice un pequeño estudio sobre las etapas posteriores para predecir la posible reutilización de algún componente en las mismas, y

- un desarrollo *bottom-up*, de forma que dichos componentes tengan información no sólo relativa a su fase de integración sino a etapas anteriores.

Un paradigma que se ajusta a este método es el desarrollo orientado a objetos, avalado por una gran parte de la comunidad *reutilizadora*.

2.6.2 Modelos de procesos software incluyendo reutilización

Dependiendo del proceso de desarrollo software por el que se haya optado, las posibilidades de integración de elementos reutilizables varían.

En un modelo de desarrollo en cascada es posible integrar componentes reutilizables en cualquiera de las fases del ciclo de vida. En el caso particular que se indica en la figura 2.4, la integración de los elementos de alto nivel de abstracción se realiza gradualmente, es decir, a medida que el contexto (sistema nuevo) evoluciona.

En un modelo de ciclo de vida utilizando prototipos es especialmente útil el desarrollo con componentes reutilizables, precisamente para el establecimiento del prototipo inicial: se construye un sistema aproximado, integrando componentes ya existentes de funcionalidad semejante, para negociar las necesidades de los usuarios.

El modelo de ciclo de vida en espiral (Boehm, 1988) necesita la evaluación de un conjunto de alternativas para progresar en el proceso software, una de las opciones es la realización de un prototipo. Para la obtención de un prototipo se pueden integrar componentes reutilizables.

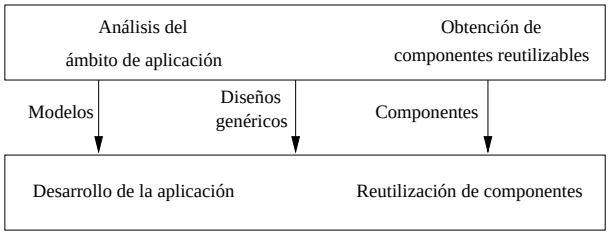


Figura 2.5. Modelo de desarrollo en espejo

Un modelo de ciclo de vida especialmente ideado para el desarrollo con reutilización de componentes es el **modelo en espejo**. En él se incluyen grupos de trabajo propios de una organización donde la reutilización ya esté integrada con un nivel elevado de madurez: grupos de analistas de dominio, generadores de componentes reutilizables, grupos de ingeniería inversa, etc. Karlsson (Karlsson, 1996) define las etapas de trabajo, quién interviene en cada etapa y cómo interactúan dichas fases (figura 2.5).

En este modelo se introduce la ingeniería de dominio, aunque en el apartado 4.4 se detalla un modelo de ciclo de vida expresamente adaptado a las técnicas de esta ingeniería.

2.7 Ejemplo de un sistema actual

Aunque existen diversidad de entornos de reutilización basados en la composición de elementos reutilizables, vamos a destacar aquí GenVoca (Biggerstaff, 1998) como ejemplo, principalmente por su versatilidad.

GenVoca permite el diseño y la construcción de sistemas software jerárquicos. Un sistema de este tipo es aquel que puede comprenderse como una sucesión de subsistemas superpuestos —cada vez más complejos— donde cada uno de ellos implementa una funcionalidad propia del dominio de la aplicación. Cada una de estas capas puede ser vista como un componente software de granularidad elevada que se puede combinar con otras capas o niveles para obtener una nueva aplicación, implementándose así un sistema de reutilización a gran escala. Cada vez que se construya una nueva aplicación, ésta se organizará en capas que son almacenadas en la biblioteca de componentes. La funcionalidad de cada nivel se especifica de forma abstracta y puede ser implementada de diferentes formas,

permitiendo así una analogía con los paradigmas orientados a objetos (clases y objetos).

En un principio, Batory et al. (Batory y O'Malley, 1992) trabajaron en dos entornos diferentes:

- uno para la elaboración de protocolos de comunicación, *Avoca*, y
- otro para la creación de sistemas de gestión para bases de datos, *Genesis*;

pero, tras observar las similitudes de ambos, se decidió unificarlos en uno solo que permitiese el diseño y la implementación de aplicaciones jerárquicas, surgiendo así **GenVoca**.

Tal y como ocurre en todos los entornos de composición, uno de los principales problemas es la validación de la composición, es decir, sabiendo que son válidos los componentes individuales asegurar que será válida también la composición resultante. En trabajos más recientes (Batory y Gerardi, 1996), se han introducido:

- *Precondiciones*: funcionalidad que ha de satisfacer el sistema para la incorporación de un nuevo componente (nivel); y
- *Postcondiciones*: funcionalidad que exporta un componente para las capas superiores.

facilitando de esta forma las tareas de validación de composiciones.

CAPÍTULO 3

Reutilización por generación

3.1 Introducción

La reutilización por generación es conceptualmente más compleja que la reutilización por composición, ya que es difícil definir componentes como entes autocontenidos y concretos. En este caso, se reutilizan procesos de generación que han sido obtenidos como resultado de una *codificación de estructuras*. Los generadores de aplicaciones, los compiladores de sistemas y los procesos transformacionales son las tres ramificaciones características dentro de esta metodología.

La generación de sistemas se ha orientado hacia la reutilización de elementos pertenecientes a las primeras fases del ciclo de vida del software, como son diseños, arquitecturas o requisitos, lo que la hace más apetecible en cuanto a su posible rentabilidad aunque mucho más difícil de aplicar.

En este capítulo se tratarán los tres paradigmas representativos de la reutilización por generación: la compilación de sistemas (apartado 3.2), la generación de aplicaciones (apartado 3.3) y los sistemas transformacionales (apartado 3.4). Por último, se realiza una comparación entre sus prestaciones y cómo afectan al ciclo de vida del software.

3.2 Compiladores de sistemas

Este tipo de generación de sistemas parte de una especificación expresada en un lenguaje VHLL (V*ery* H*igh*-L*evel* L*anguage*), y de un compilador que la interpreta y genera el ejecutable que se comporta de la forma especificada por el programador.

Los VHLLs emulan el comportamiento de los HLLs pero a un nivel de abstracción superior. Así estos lenguajes, también llamados lenguajes de cuarta ge-

neración, proporcionan una sintaxis y semántica que permite la especificación de sistemas de propósito general. En general este tipo de lenguajes proporcionan abstracciones matemáticas que no suelen utilizarse en los procesos software convencionales. Como principales ejemplos se tienen:

- **SETL** es un lenguaje de cuarta generación desarrollado en la universidad de Nueva York. Es un lenguaje imperativo, secuencial y ligeramente tipado que se basa en la teoría de conjuntos y proporciona una base matemática fuerte para la especificación de sistemas a muy alto nivel (Dubinsky et al., 1989).

- **PAISLey** presenta similitud con SETL en cuanto a su base matemática, aspectos relacionados con conjuntos. El especificador puede beneficiarse, además, de la posibilidad de definir comunicaciones entre procesos asíncronos y restricciones de tiempo real a sus sistemas. Una de las principales ventajas de especificar en PAISLey es que tiene un entorno de trabajo muy potente que permite la depuración y prueba de los sistemas, incluso aunque se encuentren inacabados (Zave, 1991).

- **MODEL** es un lenguaje no imperativo que permite la especificación de un sistema como un conjunto de restricciones. De esta manera el sistema vendrá definido como la solución simultánea de dicho conjunto de restricciones. El flujo de datos y de control no está definido explícitamente por el especificador, sino que viene determinado automáticamente por el compilador (Cheng et al., 1991).

3.3 Generadores de aplicaciones

Un generador de aplicaciones opera de forma similar a los compiladores tradicionales: partiendo de la especificación del sistema se obtiene su ejecutable. Sin embargo, en este caso, las aplicaciones (sistemas) a generar han de pertenecer al mismo ámbito o dominio y la especificación se realiza a un nivel de abstracción mucho mayor, en un DSL (D*omain* S*pecific* L*enguage*). Aunque todos los compiladores pueden verse como generadores de aplicaciones, éstos tienen otras características diferenciales:

– la posibilidad de realizar extensiones en el lenguaje para ampliar su capacidad expresiva, y

– realizar transformaciones de estructuras del programa.

Los ejemplos más cercanos de este tipo de generadores son, quizá, los generadores de analizadores léxicos (*lexer*) y sintácticos (*parser*) como Lex y Yacc (Mason y Brown, 1991).

El hecho de tener que obtener primero un generador de aplicaciones para después generar un sistema no es algo que parezca rentable a primera vista. En realidad, la obtención de un sistema partiendo de un generador de aplicaciones sólo es apropiada si:

- es necesaria la obtención de sistemas software semejantes,
- se sabe que el sistema software que se va a generar será modificado frecuentemente durante su ciclo de vida, o
- es necesario obtener una cantidad considerable de prototipos antes de converger al sistema final.

Para la obtención de un generador de aplicaciones se siguen los pasos siguientes:

1. **Identificar un dominio** o campo de aplicación realizando un estudio exhaustivo del mismo para delimitar firmemente sus fronteras. Así, normalmente, es necesario un analista de dominio (capítulo 4). Es deseable que el ámbito de las aplicaciones no sea muy extenso ya que su tamaño es directamente proporcional a la dificultad de utilización y las aplicaciones así generadas pueden ser de baja eficiencia.
2. Una vez analizado el ámbito de aplicación se **extraerán sus características**, pudiendo obtenerse así aquellas invariantes y aquellas susceptibles de ser parametrizadas. Esta caracterización del dominio es la que determinará la cobertura del generador.
3. Se establecerán las **pautas** que debe seguir el usuario para **especificar un sistema** de forma correcta y completa.
4. Por último, la codificación del programa (generador) que interprete las especificaciones dadas y genere su ejecutable.

3.4 Generación de sistemas en entornos transformacionales

En un entorno transformacional se obtiene un sistema tras la aplicación de transformaciones sucesivas a su especificación. Uno de los pioneros en este campo fue Cheatham (Cheatham, 1989) y, desde entonces, un amplio sector dentro de la ingeniería del software ha enfocado sus actividades hacia esta área.

El trabajo en este tipo de entornos permite la reutilización de modificaciones realizadas a nivel de especificación, y de las transformaciones; además del propio entorno transformacional (Krueger, 1992).

Pero, ¿qué es una transformación?. Una transformación puede definirse como una substitución de un programa por otro. Ésta es una definición poco restrictiva que podría adaptarse a diferentes contextos. En el que nos ocupa, las transformaciones suelen clasificarse en:

- **Refinamientos**: este tipo de transformaciones añaden algún tipo de detalle de implementación a una especificación abstracta, relacionando así diferentes niveles de abstracción.

- **Optimizaciones**: aquí se engloban aquellas transformaciones que persiguen una mejora de las prestaciones de un programa, normalmente su eficiencia, y que se producen dentro del mismo nivel de abstracción conceptual.

Aunque ésta es una clasificación comúnmente aceptada, las transformaciones admiten categorizaciones más detalladas y muy diversas. Nosotros hemos adoptado la clasificación que realizan Smaragdakis y Batory en (Smaragdakis y Batory, 2000). Así, dentro de los refinamientos de especificaciones, tenemos dos subgrupos:

- **Refinamientos de algoritmos**: Permiten la obtención de procedimientos operativos a partir de sus especificaciones y son el fundamento de todo sistema transformacional. Por ejemplo, la localización de un procedimiento que tenga la misma funcionalidad que un operador (incluido dentro de la especificación) se podría realizar mediante un motor de cotejamiento guiado por un conjunto de heurísticos.
  En la práctica, la obtención de algoritmos directamente desde una especificación abstracta es una línea de investigación relativamente reciente; sin embargo, algunos entornos transformacionales utilizan *patrones de algoritmos*: esqueletos genéricos de algoritmos que permiten la especialización para algunos tipos de datos y operaciones concretas. Desde luego, esta última opción es mucho menos ambiciosa pero más viable en este momento.

- **Refinamientos de tipos de datos**: Son complementarios a los anteriores, se realiza una selección de la mejor implementación del algoritmo para el tipo de dato especificado.

Las transformaciones realizadas con el objetivo de optimizar la eficiencia de los sistemas resultantes son consecuencia directa de la tecnología de los compiladores convencionales. Sin embargo, es en estos entornos transformacionales donde han cobrado mayor importancia y donde, consecuentemente, se han desarrollado un mayor número de dichas transformaciones. Estas transformaciones para la optimización pueden clasificarse en los grupos siguientes (Smaragdakis y Batory, 2000):

- **Evaluación parcial**: Este tipo de transformaciones realizan una especialización de un fragmento de código general, para que pueda ser aplicado en un contexto determinado. Esta modificación se activará bajo la satisfacción de las condiciones impuestas sobre un conjunto de parámetros.

- **Incrementales**: El objetivo de este tipo de transformaciones es enfocar aquellas computaciones inherentemente complejas como una sucesión de transformaciones incrementales, unificando así las transformaciones que se hayan realizado por evaluación parcial. La técnica más popular recibe el nombre de *diferencias finitas*.

- **Reestructuración de código**: En esta categoría se enmarcan aquellas transformaciones que realizan una mejora del flujo de datos o de control. Algunas de las más conocidas son: unificación de bucles, eliminación de código no utilizado, etc.

Todas las transformaciones anteriores podrán ser resueltas por el entorno transformacional de uno de los modos siguientes:

- **Resolución basada en patrones**: El patrón substituto se localiza en una biblioteca partiendo del patrón origen que actúa como índice. Este método es conceptualmente sencillo, aunque en la práctica no se obtienen buenos resultados en transformaciones complejas.

- **Resolución programada**: Un programa se encarga de manipular el código para obtener el resultante de la transformación. Permite una mayor flexibilidad y resoluciones más complejas que el anterior.

3.5 Comparación de los métodos de generación de sistemas: hacia la unificación

Se han descrito brevemente tres métodos de generación de sistemas, la elección entre unos y otros dependerá de las características que deseemos obtener. Cuando se habla de compiladores de lenguajes de cuarta generación se está hablando de **generalidad**, se pueden aplicar para la descripción de cualquier tipo de sistema, debido precisamente a ello su **eficiencia** se ve reducida.

Los generadores de aplicaciones son mucho más eficientes que los compiladores de VHLLs, son sistemas especialmente diseñados para la obtención de aplicaciones con una serie de características comunes —pertenecen al mismo dominio.

Sin embargo, la mayor eficiencia se logra con la aplicación de transformaciones, ya que precisamente estas pequeñas modificaciones son realizadas para

mejorar la respuesta del sistema final. Si las transformaciones desde un espacio origen a uno destino definiesen una aplicación que llevase todo elemento del primer conjunto a uno y sólo uno del segundo y, además, en el espacio origen estuviesen contemplados todos los casos posibles, se evitarían los dos problemas que impiden la **automatización** del proceso: el no determinismo y la no completitud de las transformaciones. La automatización del proceso se logra en los generadores de aplicación y en los intérpretes de VHLLs a costa de una reducción de la eficiencia, la calidad o la restricción de los dominios de aplicación.

En realidad, el mundo de la reutilización por generación es un *collage* donde suelen reunirse ideas diversas, y es difícil encontrar un entorno de este trabajo que corresponda exactamente a uno de los tres tipos anteriormente citados. Lo más habitual es que estos entornos combinen influencias de todos ellos para poder obtener las características más beneficiosas de cada uno (apartado 3.7).

El ejemplo más destacado es el de los generadores de aplicaciones, que suelen incluir en su núcleo un sistema de transformaciones.

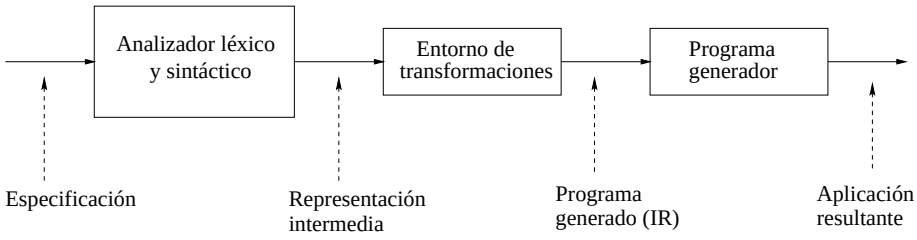


Figura 3.1. Arquitectura de un generador de aplicaciones

La arquitectura de estos entornos se esquematiza en la figura 3.1, y está organizada en tres grandes módulos:

1. Módulo analizador: realiza un análisis léxico-sintáctico de la especificación —realizada en un DSL— y obtiene una equivalente, pero expresada en un lenguaje intermedio más manejable por el motor de transformaciones.

2. Módulo de transformaciones: es en este bloque donde se entremezclan ambos paradigmas y aquí se realizan las transformaciones precisas para obtener un programa que funcione tal y como se ha especificado de un modo correcto y expresado en un lenguaje intermedio.

3. Módulo traductor: partiendo del programa en representación interna obtiene la aplicación generada en un lenguaje de programación. Este módulo puede ser múltiple, permitiendo la traducción a diferentes lenguajes de programación.

3.6 Impacto en el ciclo de vida del software

En general, la construcción de un sistema software usando alguno de estos paradigmas acorta el ciclo de vida tradicional.

La utilización de un generador de aplicaciones modifica radicalmente el proceso software, que se reduce a una única fase de especificación. El ejecutable se obtiene de forma inmediata y se prescinde de la etapa de verificación, ya que se presupone que, tras la implementación del generador, se ha comprobado que éste siempre genera código consistente con la especificación. En realidad es el propio generador quien propaga de forma automática los cambios que sufre la especificación a lo largo del ciclo de vida.

En el caso de aplicar intérpretes de VHLLs o de trabajar en entornos transformacionales se obvia la necesidad de realizar las tareas de diseño, codificación y prueba. En esta situación, sin embargo, es imprescindible la formalización de la especificación del sistema. Por ejemplo, en un entorno transformacional, el ciclo de vida se modifica de la forma siguiente:

1. *Fase de especificación del sistema en un lenguaje de alto nivel.* Normalmente estos lenguajes suelen ser de un nivel de abstracción mayor que el que presentan los VHLLs. Martin Feather propone una metodología de transformaciones partiendo de la especificación en lenguaje Gist (Feather, 1989).

2. *Fase de transformaciones sucesivas hasta la obtención del sistema final.* Se realizan ligeras variaciones sobre el programa hasta obtener un ejecutable eficaz y eficiente. No es posible su total automatización ya que es necesaria la ayuda de personal cualificado que tome decisiones de transformación para conseguir una buena realización. Normalmente los entornos de trabajo proporcionan ayuda para estas tomas de decisión.

3.7 Ejemplos de sistemas actuales

Existe una gran diversidad de entornos de trabajo basados en generadores de aplicaciones, compiladores de sistemas y entornos de transformación incluyendo reutilización. La tendencia actual, además, es la integración, de forma que se tiende a combinar influencias de cada uno de estos paradigmas con la intención de aunar sus beneficios. En este apartado se reseñan brevemente algunos de los entornos de trabajo más destacados actualmente.

Sistema DRACO

DRACO es un entorno transformacional propuesto originalmente por Neighbors (Neighbors, 1984) y en el que ahora están trabajando investigadores de la Pontifícia Universidade Católica de Río de Janeiro (Draco-PUC).

La base de este entorno es el modelado de los dominios de aplicación (ver capítulo 4). Se realiza un estudio exhaustivo de cada dominio y se crea un lenguaje específico que permita al programador expresar todas sus particularidades. DRACO parte de la especificación de la aplicación y realiza una *compilación* de la misma hasta obtener el código fuente en C, C++ o Java.

El principal mecanismo de compilación que utiliza es el **refinamiento**, de forma que se tienen transformaciones de AST a AST (A*bstract* S*yntax* T*ree*). Éstas se establecen entre especificaciones en un dominio a especificaciones en otro conceptualmente de menor abstracción (p.ej. trabajo en red a bases de datos). Estos refinamientos no se realizan en un orden preestablecido, de hecho suelen hacerse simultáneamente.

Una vez que se ha *compilado* la especificación, tras la realización de todo un conjunto de refinamientos, se ha obtenido un código pero que es muy poco eficiente. Para limpiarlo de todas las ineficiencias que presenta se hace imprescindible la aplicación de **transformaciones de optimización**. Se caracterizan por ser internas a cada dominio (p.ej. las bases de datos).

DRACO se ayuda de un conjunto de **tácticas** que guían al entorno a la hora de realizar las transformaciones (tanto refinamientos como optimizaciones). Son instrucciones de operación que favorecen una transformación u otra dependiendo de las condiciones de aplicabilidad.

Se ha utilizado para desarrollar aplicaciones dentro de dominios como el trabajo en red o la telefonía (Neighbors, 1992; Neighbors, 1996) (dominios superiores) y para ello ha sido necesario estudiar algunos subdominios como las bases de datos y las estructuras de datos entre otros.

Sistema SciNapse

SciNapse (Akers et al., 1997) es un generador para software matemático que utiliza un núcleo de transformaciones, tanto basadas en patrones como programadas, y que realiza el refinamiento de los algoritmos utilizando esquemas globales que posteriormente son particularizados. Es capaz de generar código en Fortran77, C ó CM Fortran.

Originalmente fue pensado para obtener programas que pudiesen resolver ecuaciones diferenciales para el modelado de ondas sonoras, aunque también se han encontrado otros campos de aplicación: estudio de seísmos, propagación de vertidos de base aceitosa, y, más recientemente, para las fluctuaciones financieras.

Sistema Mousetrap

Mousetrap (Dietz et al., 1998) es un sistema transformacional desarrollado por Motorola y que se aplica para la obtención de código eficiente relacionado con los productos de radiofrecuencia. Parte de una especificación, un árbol de sintaxis abstracta, y realiza las transformaciones necesarias basadas en patrones para la obtención de código utilizable. Sobre este código se realizarán diversas optimizaciones.

CAPÍTULO 4

Ingeniería de dominio

4.1 Introducción

El conjunto de actividades relacionadas con la obtención de una infraestructura que proporcione las bases para la reutilización sistemática recibe el nombre de **ingeniería de dominio**. Dentro de este proceso se han diferenciado dos etapas: **análisis de dominio** e **implementación del dominio** (Frakes et al., 1998), aunque en un principio el concepto que introdujo Neighbors (Neighbors, 1981) fue el de análisis de dominio.

Al igual que con muchos términos utilizados en la ingeniería del software, para el análisis de dominio existen múltiples acepciones. Así que comenzaremos por definir a qué le vamos a llamar dominio y, consecuentemente, qué entendemos por análisis de dominio.

Partiendo de que **dominio** es un área o ámbito de aplicación de productos software, el **análisis de dominio** consistirá en el estudio y formalización de las características de un dominio. La principal diferencia con el análisis de sistemas es que este último se realiza sobre un único sistema o aplicación, no sobre un conjunto de ellos, aunque el análisis de dominio puede ser el punto de partida para el análisis de sistemas.

Como resultado de un análisis de dominio se obtiene información interesante a la hora de reutilizar software dentro del mismo ámbito de aplicación, de hecho algunos autores, como Prieto-Díaz, particularizan la definición de análisis de dominio como el proceso por el cual se captura, identifica y organiza información de un área de aplicación con el propósito de hacerla reutilizable (Prieto-Díaz, 1990).

Tradicionalmente las actividades propias de un análisis de dominio se asocian a tareas humanas cuyo éxito, además, depende en gran medida de la experiencia del analista. Para lograr la automatización y sistematización de estas tareas se trabaja con herramientas y metodologías de captura, gestión y almacenamiento de

información, por ejemplo las proporcionadas por diferentes técnicas de inteligencia artificial.

Cuando las actividades relacionadas con el análisis de dominio se enfocan hacia la reutilización se distingue, dentro de la ingeniería de dominio, una segunda fase: la implementación de dominio. Si se realizase una analogía entre la ingeniería de dominio y la obtención de una línea de ensamblaje o manufactura en una fábrica tradicional, la fase de análisis de dominio correspondería al diseño de la línea de producción y la fase de implementación a la obtención y puesta en marcha de la línea de ensamblaje.

En este capítulo se introducirán los conceptos relativos al análisis, a la implementación de dominio y en el apartado 4.4 se detallará cómo afectan estos procesos al ciclo de vida del software. Por último se comentarán tres metodologías: DARE, FODA y FORE que utilizan la ingeniería de dominio como base para el desarrollo de aplicaciones software.

4.2 Análisis de dominio

El análisis de un dominio es realizado por los analistas e ingenieros de dominio partiendo de implementaciones o aplicaciones ya existentes y toda la información que éstas conllevan (código, documentación de requisitos y funcionalidad, diseño, manuales de usuario, ...), de requisitos actuales y futuros y de otras documentaciones técnicas relacionadas con el ámbito de aplicación. Siguiendo unas pautas establecidas para el análisis, se extrae toda la información relevante y, tras realizar tareas de abstracción, se encapsula y organiza para obtener estándares, arquitecturas de dominio y modelos de dominio.

Como resultados de la realización de un análisis (figura 4.1) de un dominio se obtienen los **modelos del dominio** y la **arquitectura del dominio**. Los modelos de dominio recogen los requisitos propios del ámbito estudiado, señalando las necesidades y características comunes y aquellas que son diferentes para las aplicaciones del dominio. La arquitectura de dominio representa un *esqueleto* o patrón común a las aplicaciones del dominio, de forma que caracteriza los elementos de la aplicación y la relación entre los mismos, además de un conjunto de pautas o guías (metodología) que permita gestionar su diseño y desarrollo.

Es necesario notar que el análisis de dominio es un proceso iterativo, ya que tanto los requisitos como las fronteras del ámbito de aplicación no suelen ser estáticos.

4.3 Implementación del dominio

Dependiendo de la metodología seleccionada para implementar un proceso de reutilización de software (sección 1.3.1), los objetivos que se persiguen con

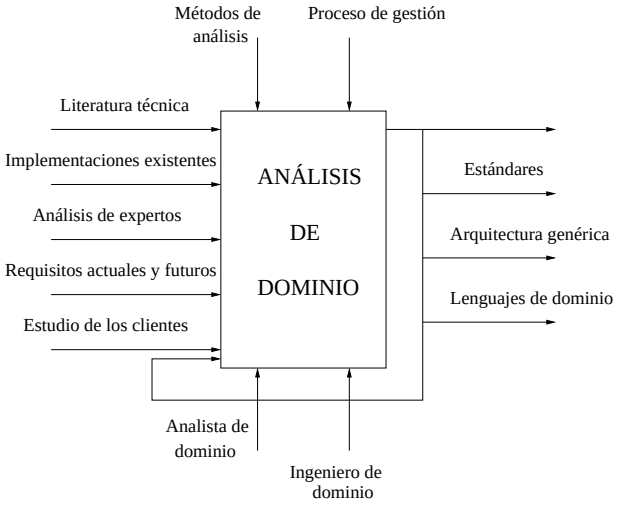


Figura 4.1. Resultados tras un análisis de dominio (Prieto-Díaz, 1990)

el análisis de dominio varían, y éstos se muestran claramente en las actividades propias de la implementación del dominio.

En el caso de la reutilización por generación el principal objetivo es conseguir una estructura común a todas las aplicaciones del ámbito de forma que ésta sea reutilizable: una arquitectura del dominio. Esto es especialmente claro en los generadores de aplicación, donde es inviable la implementación sin un conocimiento exhaustivo previo del dominio.

En la reutilización por composición se intentará localizar aquellos componentes que tengan interés desde el punto de vista reutilizable, bien partiendo de sistemas antiguos vía ingeniería inversa, bien definiendo necesidades (preferentemente comunes) todavía no cubiertas con el objetivo de generar aquellos componentes necesarios.

4.4 Impacto de la ingeniería de dominio en el ciclo de vida del software

Una vez que se ha decidido reutilizar elementos software, el proceso de creación de los sistemas se ve radicalmente modificado. En el apartado 2.6.2 se comentaron brevemente algunas modificaciones que habría que realizar en los ciclos de vida tradicionales. Hoy en día, sin embargo, no es factible enfocar la inclusión de un proceso de reutilización sistemático sin la inclusión de las actividades propias de la ingeniería de dominio. En este apartado se recoge uno de los modelos propuestos para adaptar el proceso software a estas nuevas necesidades.

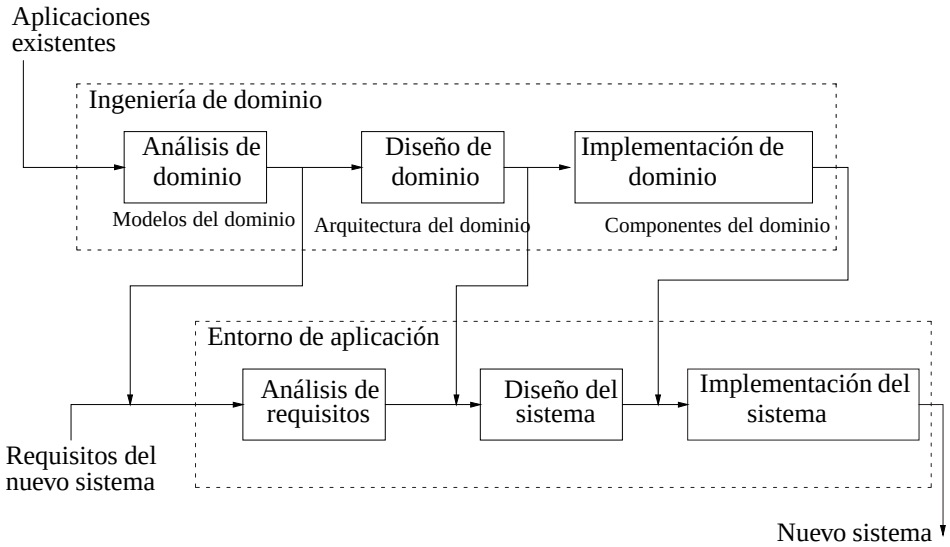


Figura 4.2. Modelo de proceso software incluyendo actividades propias de la ingeniería de dominio

Como resultado del proyecto STARS, financiado por el Departamento de Defensa de los EE.UU., se propuso el modelo de ciclo de vida esquematizado en la figura 4.2 (Addy, 1998) (STARS Two Life-Cycle Model). Las actividades asociadas a este proceso se dividen en dos fases o ciclos diferentes pero complementarios:

- Actividades propias de la ingeniería de dominio: es en esta fase donde se analiza el ámbito de aplicación, se realiza un modelo o modelos del dominio, se define la arquitectura del mismo y se crean los componentes reutilizables propios del ámbito.

- Actividades propias de la creación de un sistema software: la obtención de una nueva aplicación propia del dominio parte de un conjunto de requisitos, aquellos que sean propios del nuevo sistema se desarrollarán partiendo de cero, sin embargo aquellos que son *comunes* o característicos del ámbito de aplicación pueden ser reutilizados sin más que aprovechar el trabajo realizado en la fase anterior.

4.5 Metodologías de análisis de dominio

4.5.1 FODA: Feature-Oriented Domain Analysis

FODA (Kang et al., 1990) es una metodología desarrollada por el SEI[1] para la aplicación del análisis de dominio, definiendo las etapas del método y los resultados obtenidos en cada una de ellas. A continuación se describen brevemente las etapas del método:

1. **Análisis del contexto**: el principal objetivo de esta etapa es la completa caracterización del dominio con la consecuente definición de las fronteras. Como resultado se obtiene un modelo del contexto consistente en un diagrama de estructura y un flujo de datos. El diagrama de estructura establece la organización jerárquica entre los distintos módulos localizados en el dominio: superdominios, subdominios, etc; y el diagrama de flujo de datos establece las comunicaciones entre los módulos. El trabajo realizado en esta fase es crucial para la realización satisfactoria de las siguientes.

2. **Modelado del dominio**: una vez acotado el entorno de trabajo se procede a la localización de semejanzas y diferencias dentro de las aplicaciones pertenecientes al ámbito de trabajo:

   – análisis de características,

   – modelado entidad-relación, y

   – análisis funcional.

3. **Modelado de la arquitectura del dominio**: esta fase proporcionará una base o *modelo de arquitectura* que será particularizada para cada una de las aplicaciones del dominio. Consistirá en un diseño de alto nivel donde se han identificado los componentes comunes y las relaciones entre ellos para poder ser reutilizado en la generación de diferentes sistemas del dominio.

FORM (F*eature*-O*riented* R*euse* M*ethod*) (Kang et al., 1998) surge como una ampliación de FODA. La principal característica que añade FORM es una descripción exhaustiva del análisis de características, realizado sobre las aplicaciones ya existentes en el dominio, y cómo utilizar los resultados para la obtención de arquitecturas y componentes reutilizables.

4.5.2 DARE: Domain Analysis and Reuse Environment

DARE (Frakes et al., 1998) es una herramienta CASE que proporciona un entorno de trabajo y una metodología que facilita las tareas propias del análisis de

[1] Software Engineering Institute, fundación de investigación y desarrollo financiado por el Departamento de Defensa de EE.UU.

dominio: extracción de información sobre el ámbito de aplicación y adquisición y almacenamiento de dicha información. Básicamente sigue el comportamiento esquematizado en la figura 4.1.

Las principales ventajas respecto a otros entornos de trabajo como FODA y ODM (Simos et al., 1995; Simos, 1999)[2] es su elevada modularidad que la hace compatible con dichos entornos y un mayor esfuerzo en la automatización de estas actividades.

El objetivo de DARE es la consecución de una arquitectura genérica del dominio, compuesta por los componentes y su interrelación, de forma que se destaquen sus parecidos y diferencias en funcionalidad. Como resultado del análisis de dominio se obtendrá, también, documentación sobre éste y un sistema de clasificación automatizado organizado por facetas (Prieto-Díaz y Freeman, 1987; Prieto-Díaz, 1991).

4.5.3 FORE: Family Of REquirements

FORE (Lam, 1998) establece una metodología para el análisis de requisitos dentro de un ámbito de aplicación o área con el objetivo de crear *requisitos generales* que puedan ser instanciados para la obtención de aplicaciones concretas dentro del dominio. Es decir, se trabaja con la idea de *productos o aplicaciones genéricos* que engloben los rasgos comunes e instanciables de un subtipo de sistemas dentro del ámbito. FORE define los pasos siguientes:

1. **Establecimiento de los grupos de trabajo.** Normalmente estos grupos constarán de: especialistas del dominio, que contribuirán en los aspectos más técnicos del trabajo; diseñadores que colaboran en la obtención de productos genéricos; gestores para la administración del trabajo entre los grupos y clientes que, aportando su punto de vista, colaboren en el análisis de los productos.

2. **Análisis del producto.** Se define un producto genérico que sea común para un conjunto de clientes o usuarios dentro del dominio.

3. **Estructuración de los requisitos.** Para cada producto genérico se identifican y formalizan los requisitos comunes y aquellos instanciables. Se almacenarán de forma jerárquica, donde se incluyen también aquellos requisitos opcionales para una aplicación concreta. En esta fase es necesario realizar un análisis de variabilidad que impida la explosión de estados en la jerarquía. Este análisis se basa, principalmente, en un estudio detallado de cada requisito de forma que las variaciones permitidas en el mismo sean propias exclusivamente de un alto nivel de definición.

[2]Metodología desarrollada durante los últimos diez años dentro del proyecto DARPA STARS (Software Technology for Adaptable, Reliable Systems; financiado parcialmente por Defense Advanced Research Projects Agency)

4. **Generación de un sistema.** Una vez que se tienen los requisitos propios del producto genérico se realizará una instanciación para obtener los requisitos propios de un producto concreto.

5. **Análisis de cambios.** Teniendo en cuenta que los dominios son entes dinámicos puede ser preciso: redefinir un dominio, revisar las necesidades de los clientes, y reestructurar los requisitos obtenidos en la tercera fase para reflejar fielmente los cambios anteriores.

CAPÍTULO 5

Madurez y costes de un programa de reutilización

5.1 Introducción

Uno de los principales temas de debate en la introducción de programas de reutilización de software es su impacto económico. Aunque existe consenso respecto a los beneficios que podría obtenerse debidos a:

- un incremento en la producción, y
- la mejora de la calidad de los productos, lo que conlleva un ahorro en tareas de mantenimiento y una mayor satisfacción de los usuarios con los productos,

también es cierto que no es fácil evaluar el grado de mejora que puede llegar a conseguirse. Es necesario el desarrollo de modelos económicos y métricas de software apropiadas para cuantificar los costes y los beneficios de la reutilización de software: tanto para su estimación (*a priori*) como para su evaluación (*a posteriori*). Precisamente de los resultados de estos estudios dependen decisiones como (Mili et al., 1995):

- comenzar o no un programa de reutilización dentro de una empresa u organización software,
- desarrollar o no un elemento software para que pueda ser reutilizado, y
- utilizar un elemento software reutilizable para el desarrollo de una determinada aplicación.

Está claro que los mayores beneficios revertirán en organizaciones que tengan un programa de reutilización maduro y firmemente asentado, tanto estructural como tecnológicamente hablando, pero ¿cómo evaluar el grado de madurez de un programa de reutilización? y, lo que puede ser más interesante, ¿cómo contribuir a mejorarlo?

Partiendo de que es determinante la elección de un modelo de producción y una metodología de reutilización que se adapte a él, es evidente que la madurez de un programa de reutilización estará íntimamente relacionada con la madurez del proceso software al que está asociado. Es decir, cuanto más automatizado y definido se encuentre éste, mejor y más rentable será la reutilización dentro de él.

Existen varias normas que permiten la evaluación de la calidad y madurez del proceso software, de todas ellas quizá la más popular sea el CMM (C*apability* M*adurity* M*odel*). Este capítulo aborda la introducción de un programa de reutilización dentro de una organización software, y después se realiza una breve revisión de las diferentes normas, prestando especial atención al CMM ya que basándose en él se ha realizado una extensión para la evaluación de la madurez de la reutilización en una organización productora de software. Por último, se tratará sobre la cuantificación de costes asociados a la implantación de un programa de reutilización.

5.2 Implantación de un programa de reutilización

La implantación de un programa de reutilización en una empresa provoca una serie de cambios, no sólo técnicos sino de estructura y organización. Encontrar la respuesta a cómo enfocar estos cambios para que sean lo menos traumáticos posible permitiendo, además, un mayor margen de beneficios, es la clave.

Existe diversa bibliografía donde se estudia en profundidad este campo dentro de la reutilización de software, menos técnico que otros pero de gran relevancia en la actualidad. Normalmente los autores difieren en las fases en las que estructurar la implantación del programa de reutilización.

Karlsson (Karlsson, 1996) propone una metodología para introducir un programa de reutilización en una empresa influenciado sobre todo por el *Reuse Adoption Guidebook* del SPC (S*oftware* P*roductivity* C*onsortium*), y define las siguientes fases:

1. *Inicialización:*
   Antes de particularizar un programa adaptándose a las necesidades de una organización, será necesario informarse sobre aspectos técnicos y de gestión implicados en la reutilización. Además deberá precisarse cómo puede ayudar a la consecución de los objetivos de la empresa y estudiar experiencias en organizaciones semejantes.

2. *Definición de la estrategia de reutilización y su evaluación:*
   Ya dentro del contexto particular que define cada empresa, es necesario estudiar los diferentes productos que ofrece la compañía y analizar su mercado para localizar puntos de reutilización potenciales. Con toda esta información en la mano y el tipo de gestión y estructura propias, se puede definir una estrategia de reutilización adaptada a las necesidades de la empresa.

3. *Plan de implementación:*
   Es necesario decidir cómo implementar el soporte técnico preciso para llevar a cabo la estrategia de reutilización seleccionada. Dependiendo de la organización y la gestión de cada organización, se definirán los equipos de trabajo que mejor se adecuen a estas necesidades.

4. *Implementación y monitorización:*
   La implementación del programa de reutilización deberá realizarse gradualmente, normalmente comenzando con un proyecto piloto. A partir de aquí, se extenderá al resto de los proyectos de la empresa.

En (Mili et al., 1995) se recoge la aportación de Davis (Davis, 1993) que coincide con la anterior en los tres últimos puntos. Sin embargo el programa de reutilización se plantea sólo en aquellos dominios de aplicación donde resulte rentable, pudiendo además implementar estrategias de reutilización diferentes en dominios distintos:

1. *Estudio del desarrollo del programa:*
   En esta fase es necesario identificar aquellos dominios de aplicación con mayor potencial para la reutilización. Para ello se evaluará la estabilidad, su rango, etc.

2. *Definición del programa de reutilización:*
   Partiendo de la información anterior, se seleccionarán aquellos dominios que ofrezcan un mayor potencial de reutilización, menor riesgo, y mayor rentabilidad. Una vez hecho esto será preciso definir un conjunto de objetivos razonables e identificar las diferentes alternativas de reutilización para cada dominio. Los objetivos a lograr se definen en función de tres medidas:

   - Habilidad de reutilizar: mide la proporción de reutilización potencial que está siendo realmente explotada.
   - Eficiencia de la reutilización: mide qué proporción de las oportunidades de reutilización, objetivo de la organización, están siendo realmente aprovechadas.
   - Eficacia en la reutilización: relación entre los beneficios obtenidos gracias a la reutilización y los costes que ocasiona.

Es difícil evaluar el potencial de reutilización de un dominio determinado así que la habilidad de reutilización es un valor meramente orientativo. La medida de la eficiencia, sin embargo, es de gran interés y suele enfocarse como el porcentaje de código reutilizado dentro de la elaboración de un nuevo sistema. El principal problema al realizar medidas basadas en el número de líneas de código fuente es que existen fragmentos del componente que no van a ser utilizados y contabilizan igual. Además, es difícil separar las instanciaciones de utilización: puede que un objeto o componente se utilice un número elevado de veces incluso con funcionalidades diferentes y sólo contabilice una vez (Lim, 1999).

5.3 Estándares de evaluación del proceso software

Desde que se produjo la llamada crisis del software han surgido una serie de iniciativas, normalmente estatales, con la intención de conseguir un conjunto de pautas que mejoren el potencial de los procesos software[1] y, consecuentemente, la calidad del software obtenido.

Una organización inmadura no realiza la gestión de los proyectos a largo plazo, sino que las soluciones adoptadas suelen ser *ad-hoc* y, aunque tenga un proceso software especificado no se aplica de forma habitual. Esto provoca que, cuando se tienen plazos estrictos, suela sacrificarse la funcionalidad y la calidad del producto. Sin embargo, una organización con un proceso software maduro y asentado consigue:

– mejorar la predecibilidad, ya que las estimaciones realizadas acotan la distancia entre objetivos y resultados de una forma fiable y realista; y

– mejorar la efectividad, ya que aumenta la productividad y la calidad reduciendo costes y tiempos de desarrollo.

Watts Humphrey fue uno de los pioneros en este campo y su trabajo fue publicado en el libro *Managing the Software Process* en el año 1989 (Humphrey, 1989) . Tomando como guía los resultados de Humphrey surgen diferentes grupos de trabajo con la intención de proporcionar un conjunto de pautas que guíen a las organizaciones para mejorar su proceso software, evaluando también el estado de madurez en que dicho proceso se encuentra. Los más significativos son:

➢ El Departamento de Defensa de los EE.UU. (DoD) financia el SEI (S*oftware* E*ngineering* I*nstitute*), donde se desarrollará el estándar CMM

[1] Se define *capacidad de un proceso software* como el rango de resultados que es posible alcanzar cuando se sigue dicho proceso software.

(sección 5.3.1). En (Paulk et al., 1994) se detalla la evolución de este modelo.

- El Ministerio de Defensa del Reino Unido (MoD) financiará, a través de la DRA (D*efense* R*esearch* A*gency*), los trabajos en este ámbito que tendrán como resultado SPICE (S*oftware* P*rocess* I*mprovement and* C*apability d*E*termination*) en el año 1993 (Eman et al., 1997). Esta contribución será la base del estándar ISO/IEC 15504, cuya primera versión se publica en 1997 (ISO, 1997). Este estándar abarca tanto la definición de las actividades propias del proceso software como la mejora de la capacidad del propio proceso en sí, definiendo seis niveles o fases de madurez del proceso.

- La Fuerzas Armadas Alemanas financiaron el desarrollo del V-Model (IABG, 1992). Este trabajo se orienta hacia la definición del ciclo de vida del software y las actividades paralelas a éste.

- TRILLIUM (Trillium, 1994) es un modelo obtenido por el consorcio entre los siguientes operadores canadienses de telecomunicación: Bell Canada, Bell Northem Research y Northem Telecom. Se basa en la primera versión del CMM y en (Popel y Wise, 1996) se realiza una comparativa entre ambos métodos.

- Como resultado del proyecto ESPRIT financiado por la CE, desarrollado entre septiembre de 1991 y febrero de 1993, surge BOOTSTRAP (Kuvaja y Bicego, 1994) como una metodología de desarrollo de software. Basado en el CMM, alguno de sus puntos ha sido incluido en la familia ISO 9000 y en el PSS-05-0 (ESA, 1991). Este último fue desarrollado por la ESA (E*uropean* S*pace* A*gency*).

- ISO (I*nternational* S*tandards* O*rganization*) desarrolló un conjunto de estándares denominados ISO 9000, de los cuales son propios del campo del software el ISO 9000-3 (ISO, 1994), ISO/IEC 12207 (ISO, 1995) e ISO/IEC 15504.

5.3.1 Capability Madurity Model (CMM): enfoque hacia la reutilización

El *Capability Madurity Model* proporciona una descripción de los estados a través de los cuales evoluciona una organización software, que permiten definir, implementar, medir, controlar y mejorar su proceso software (Zahran, 1998).

El modelo proporciona un conjunto de pautas que pueden ayudar a una organización a mejorar su proceso de producción de software. Todas estas pautas están agrupadas en cinco etapas o niveles que, además, pueden ser utilizados como una escala que permita medir el grado de madurez del proceso software de

una compañía. Fue desarrollado por el SEI y, desde que fue propuesto en el año 1987, se ha producido una continua mejora del modelo, provocada principalmente por la colaboración de empresas privadas y organizaciones estatales. Las ideas utilizadas para la ingeniería del software fueron exportadas hacia otras áreas de aplicación, lo que ha provocado la aparición de otros modelos CMM: SA-CMM (S*oftware* A*dquicisition*), SE-CMM (S*ystem* E*ngineering*), IPC-CMM (I*ntegrated* P*roduct* M*anagement*), etc. En esta sección se hablará exclusivamente del SW-CMM (S*ofWare*).

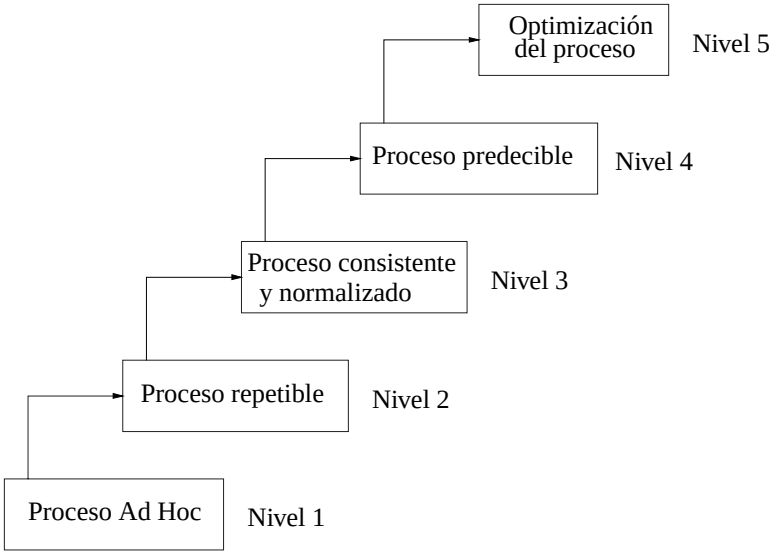


Figura 5.1. Niveles del modelo CMM

En la figura 5.1 se han esquematizado los cinco niveles de madurez que, según el modelo CMM, definen los posibles estados de una organización software. Cada etapa o nivel consta de un conjunto de criterios o guías comunes para el mantenimiento, gestión y desarrollo del software mejorando sus características y las del proceso en sí. La evaluación de la madurez de un proceso software se obtendrá tras la resolución de unos cuestionarios.

Estos niveles propios de un proceso software sirven como base para establecer otros paralelos para un programa de reutilización. Para poder hablar de un programa de reutilización, éste ha de estar asociado a un proceso software, de modo que podemos suprimir el primer nivel: el correspondiente a un proceso software inexistente. Teniendo esto en cuenta, los niveles restantes son:

- *Nivel 2: reutilización dentro de un proyecto.*
  En esta fase no existe un programa de reutilización institucionalizado y tampoco existe coordinación entre proyectos. Se caracteriza, entre otros, por incluir:

  - Gestión de requisitos: se estudia cada requisito y se negocia con el usuario la posibilidad de reutilizar trabajo con el consiguiente abaratamiento en los costes, renunciando a algunos de los requisitos iniciales.
  - Gestión del proyecto: introduciendo tareas propias de un programa de reutilización dentro de las actividades de desarrollo de una aplicación.

- *Nivel 3: institucionalización de la reutilización.*
  En esta fase se establece una coordinación formal entre proyectos, lo que conlleva la institucionalización del programa de reutilización y nuevas necesidades como la gestión de una biblioteca de componentes:

  - definiendo la política de utilización de la biblioteca,
  - adecuando los componentes a las necesidades de la organización,
  - documentando los elementos, evaluando la efectividad de la biblioteca, etc.

- *Nivel 4: evaluación del programa de reutilización.*
  En esta fase, la compañía tiene un control cuantitativo sobre las actividades de reutilización: se evalúa la calidad del software obtenido, la capacidad de que éste sea reutilizado y el proceso de gestión.

- *Nivel 5: optimización del programa.*
  En esta fase, el trabajo está claramente enfocado a la mejora del programa de reutilización detectando puntos débiles, mejorando el soporte tecnológico y variando, si es preciso, la gestión dentro de la compañía.

5.3.2 Reuse Madurity Model

El SPC (S*oftware* P*roductivity* C*onsortium*) ha desarrollado otro modelo que permite evaluar el grado de madurez de un programa de reutilización (Favaro, 1999). Una de las diferencias con el CMM es el número de niveles de madurez que define, que, en este caso, son cuatro:

1. *Reutilización a nivel de proyecto:* se reutilizan elementos software pero sólo entre los equipos de trabajo implicados en el desarrollo de una aplicación o proyecto concreto y sólo elementos obtenidos para esa misma aplicación.

2. *Reutilización a nivel de organización:* se han definido bibliotecas de elementos, se tiene un programa de reutilización y dichos elementos pueden ser compartidos por aplicaciones diferentes y grupos de trabajo independientes.

3. *Reutilización estructurada:* se ha definido una línea de producción donde se contempla la reutilización en cada etapa. En esta fase cobran más importancia las métricas relacionadas con la reutilización ya que se intenta evaluar su incidencia dentro de la producción.

4. *Reutilización predecible:* en esta etapa la infraestructura de reutilización es lo suficientemente flexible para que pueda ser adaptada rápidamente a las exigencias del mercado. Es posible, además, anticiparse y predecir necesidades que puedan ser cubiertas obteniendo elementos reutilizables.

5.4 Cuantificación de costes asociados al desarrollo con reutilización

5.4.1 Costes asociados al desarrollo con reutilización

En esta sección nos centraremos en la influencia que tiene en el coste global de un programa de reutilización, la adopción de elementos reutilizables para el desarrollo de una aplicación. En (Barnes y Bollinger, 1991) se estima este coste medio según el tipo de elementos a reutilizar: reutilización de elementos no modificables (cajas negras) o la reutilización de elementos modificables (cajas blancas).

Tal y como se comentó en el apartado 2.6 para poder incluir un elemento reutilizable en el desarrollo de un nuevo sistema es preciso: localizar el elemento adecuado en la biblioteca y, si se trata de uno modificable, adaptarlo para que pueda ser integrado en el nuevo entorno.

➢ Costes asociados a la reutilización de un elemento no modificable.
En este caso el elemento a reutilizar ha de satisfacer exactamente los requisitos pedidos ya que no es posible adaptar el componente. De esta manera si se denota como:

- p a la probabilidad de que se localice el elemento buscado en la biblioteca,
- B al coste asociado a la búsqueda del elemento, y
- D al coste del desarrollo del componente desde el principio, sin partir de ningún trabajo anterior

se puede expresar el coste medio deseado como: $B + (1 - p) \cdot D$.

Está claro que, para que el proceso compense, será necesario que este coste sea inferior al coste de desarrollar el elemento desde cero:

$$B < p \cdot D$$

➢ Costes asociados a la reutilización de un elemento modificable.
Para este tipo de elementos habrá que tener en cuenta, además, los costes derivados de las adaptaciones necesarias para hacer que sean integrados en las nuevas aplicaciones. La expresión del coste medio es la siguiente:

$$B + (1 - p)(B_{aprox} + q \cdot A + (1 - q) \cdot D)$$

donde:

- p es la probabilidad de que se localice el elemento buscado en la biblioteca,
- q la probabilidad de que se localice algún elemento próximo al buscado,
- B_{aprox} el coste asociado a la búsqueda de un elemento parecido en la biblioteca,
- A el coste de adaptación del componente,
- B el coste asociado a la búsqueda del elemento, y
- D el coste del desarrollo del componente desde el principio sin partir de ningún trabajo anterior

Así que para que la reutilización resulte rentable en este caso, será necesario que este coste sea inferior al coste de desarrollar el elemento desde cero:

$$B + (1 - p) \cdot Baprox + q \cdot (1 - p) \cdot A \leq (p + q \cdot (1 - p)) \cdot D$$

5.4.2 Costes asociados a la construcción de un elemento reutilizable

Los costes asociados a la construcción de un elemento reutilizable dependen, sobre todo, del tipo de elemento. Cuando se trata de construir un generador, por ejemplo, las cifras son bastante más elevadas que si estuviésemos ante la elaboración de un componente de código.

Siguiendo el paradigma de la reutilización de generadores de aplicación se necesitaría, como mínimo, inversión en un buen análisis de los dominios de desarrollo de la empresa. De estos estudios se podrían deducir aquellos campos de aplicación donde la reutilización podría ser más viable y la cobertura del generador para que su construcción fuese rentable. El que un generador cubra todo el espectro de funcionalidad de un entorno de aplicación no lo convierte necesariamente en un generador rentable, al contrario, se habrá invertido en cubrir particularizaciones que escasamente revertirán en un aumento de los ingresos. Además habrá que tener en cuenta otros factores como la esperanza de vida del generador, que normalmente dependerá de la estabilidad del dominio al que esté asociado.

Sin embargo, la construcción de cada componente reutilizable, como los que se han definido en el apartado 2.2, requiere un desembolso menor. También es preciso analizar los dominios de aplicación, desarrollar criterios de clasificación y búsqueda de componentes en bibliotecas, y un entorno de reutilización adecuado, pero todas estas inversiones se dividen entre la multitud de componentes resultantes. En este caso, de reutilización por composición, los costes de desarrollo de un componente reutilizable son mucho mayores que los de la obtención de componentes comparables en funcionalidad, pero que no se construyen con el objetivo de que puedan volver a ser utilizados. Y ésta es una opinión unánime.

En este contexto será necesario tener en cuenta el dinamismo dentro de la biblioteca de componentes: tras evaluar el porcentaje de *usos* de un componente si éste no es lo suficientemente elevado deberá ser retirado de la biblioteca. Es preciso llegar a una solución de compromiso en el número de componentes almacenados, ya que su aumento indiscriminado provoca que el funcionamiento de los motores de búsqueda y clasificación se resientan innecesariamente.

PARTE II

Introducción y objetivos del trabajo de tesis

CAPÍTULO 6

Punto de partida

El principal objetivo de esta tesis es definir un proceso de reutilización de componentes software de elevado nivel de abstracción, especialmente adecuado a un proceso de desarrollo software ya existente. Dicho proceso se caracteriza por ser iterativo, incremental y totalmente formalizado y, dado que el trabajo descrito en este documento está íntimamente ligado a dicho proceso, se hace imprescindible una breve descripción de sus peculiaridades.

6.1 Introducción

Aunque la ingeniería del software es considerada un campo de investigación relativamente reciente y poco maduro, su rápido crecimiento, en apenas cuatro décadas, se ha producido condicionado, fundamentalmente, por la satisfacción de una demanda creciente de sistemas software cada vez mayores y más complejos. El conjunto de teorías, métodos y herramientas aportadas por los investigadores del ámbito persiguen precisamente mejorar la eficiencia y la eficacia de los procesos de creación de software, logrando así sistemas de mayor calidad.

Si entendemos como calidad de un sistema software el grado de satisfacción de los requisitos y expectativas planteadas *a priori* por los usuarios, quizá pueda considerarse que la inclusión de las técnicas de descripción formal FDTs (F*ormal* D*escription* T*echniques*) dentro del proceso de desarrollo de dichos sistemas sea una de las principales aportaciones realizadas para mejorarla. Íntimamente ligada a la aplicación de FDTs camina la verificación formal, ya que permite comprobar que, efectivamente, se satisfacen los requisitos del usuario en cualquiera de las fases de desarrollo del sistema.

Por otro lado, la reutilización de software dentro de los procesos de desarrollo, además de favorecer también la calidad de los sistemas generados (ver apartado

1.2), permite mejorar la eficiencia del propio proceso, evitando la realización de tareas redundantes y permitiendo que éstas sean competencia exclusiva de especialistas de cada ámbito.

El trabajo desarrollado en esta tesis fusiona ambas líneas, integrando un proceso de reutilización de componentes software dentro de un proceso de desarrollo totalmente formalizado, ya existente, intentando así mejorar su eficiencia y calidad.

Es objetivo de este capítulo realizar una descripción breve de los conceptos que van a ser manejados repetidamente a lo largo de este trabajo (apartados 6.2 y 6.3), e introducir el proceso de desarrollo software sobre el que se van a definir las tareas de reutilización (apartado 6.4).

6.2 La aplicación de FDTs en el ciclo de vida

Para que un sistema software satisfaga plenamente los requisitos expresados por el usuario, es de vital importancia asegurar la corrección de la fase inicial de captura y análisis. La utilización de lenguaje natural para expresar las necesidades del usuario ha llevado, tradicionalmente, a que se parta de especificaciones incompletas, imprecisas y, a veces, inconsistentes, redundando así en sistemas de baja calidad.

La formalización de todo el proceso de especificación, diseño y desarrollo de un sistema software, evita los problemas de imprecisión y ambigüedad en las especificaciones y proporciona una base que permite afrontar labores de verificación en fases más tempranas del proceso. De esta manera se consigue eliminar el "efecto bola de nieve", producido al arrastrar errores desde las primeras fases hasta las últimas.

Pero no son la verificación, ni la especificación de requisitos formales las únicas ventajas de las FDTs, además éstas proporcionan un lenguaje intermedio entre el diseñador y el computador, que permite dar a este último un papel más relevante que el que tenía hasta ahora en el proceso de diseño y desarrollo. El diseñador podrá encargarse entonces de la parte más creativa del proceso, como la toma de decisiones, y dejar que el computador tome parte activa en la manipulación y análisis de datos, y en el desarrollo de documentación.

A pesar de todas las ventajas que aporta la inclusión de técnicas formales en los procesos de desarrollo software —respecto a lo cual existe un amplio consenso— la industria del sector no recurre a ellas todavía. Los motivos son diversos, pero quizá los siguientes sean los más representativos:

- La descripción formal dificulta la comprensión de las características de un sistema a un usuario no familiarizado.

- El elevado coste computacional de cualquier verificación formal puede im-

posibilitar la aplicación de estas técnicas en sistemas de medio o gran tamaño, precisamente donde son más necesarias.

- Al ser un área relativamente reciente, todavía no se han definido procesos de desarrollo software que integren correctamente dichas técnicas, y

- siendo además indispensable la ayuda del computador, todavía no existen herramientas adecuadas que proporcionen un entorno de trabajo amigable y eficaz.

El proceso software sobre el que se va a trabajar (consultar el apartado 6.4) nació con el objetivo de cubrir las carencias anteriormente citadas: procurando integrar diferentes técnicas de descripción formal para tomar sus mejores prestaciones, intentando salvar la distancia conceptual con el usuario y facilitar la transferencia tecnológica. El único punto que quedó por solventar, el elevado coste computacional asociado a la utilización de técnicas de verificación formal, es precisamente el que se intentará solucionar con este trabajo de tesis.

6.3 Verificación formal

Una aplicación telemática típica consta de un conjunto de procesos (normalmente reactivos) que, aún ejecutándose en máquinas autónomas e independientes, es necesario sincronizar. A la hora de comprobar su corrección hay que añadir a la problemática típica de cualquier sistema, los problemas que añade la ejecución paralela y sincronizada de procesos. La casuística de comportamientos es tan elevada en esta situación, que imposibilita la verificación dinámica clásica, basada en la comprobación del sistema en un conjunto de escenarios. Así que, prácticamente, se puede decir que la única opción restante para afrontar la verificación de este tipo de sistemas es la aplicación de técnicas de verificación estática: demostradores de teoremas y/o *model checking*.

Demostradores de teoremas

La utilización de demostradores de teoremas es el enfoque más tradicional dentro de la verificación formal y, aún así, su aplicación es escasa debido principalmente al elevado grado de formalización que se precisa. Esta técnica se basa en expresar tanto el sistema como las propiedades en alguna lógica matemática apropiada, que defina un conjunto de axiomas y reglas de inferencia. El cumplimiento o no de dichas propiedades se demuestra aplicando, en un proceso deductivo, dichos axiomas y reglas. El proceso concluye con la obtención de una relación de equivalencia u orden entre la especificación del sistema y la propiedad formulada.

Es una metodología muy rigurosa que asegura la corrección de los resultados e incluso permite trabajar con espacios de estados infinitos. Sin embargo la

dificultad en la obtención de los mismos y el escaso grado de automatización logrado, hace que su aplicación se reduzca fundamentalmente a sistemas con graves restricciones de seguridad.

Precisamente es el grado de automatización del demostrador el que permite clasificarlos en dos grandes grupos: aquellos más automáticos que requieren una mínima intervención del usuario, donde se ubican *Nqthm* (Boyer y Moore, 1979) y sus descendientes como *ACL2* (Kaufmann y Moore, 1995); y el conjunto de demostradores que precisan de una mayor colaboración del usuario para llevar la demostración a buen término, grupo en el que se incluye *LCF* (Gordon et al., 1979) y sus descencientes, como *HOL* (Gordon, 1988) y *Nuprl* (R. Constable et al., 1986). Normalmente los demostradores más automáticos son empleados para sistemas generales, pero cuando es necesario evaluar características más específicas resulta prácticamente imposible no recurrir a la ayuda de los usuarios, así que debido a estas dificultades de aplicación, algunos autores han propuesto la combinación de demostradores de teoremas con técnicas de *model checking*, surgiendo así *PVS* (Owre et al., 1992) o *STeP* (N. Bjorner et al., 1996).

Model checking

Más reciente que los demostradores de teoremas, la aplicación de *model checking* se basa en la obtención de un modelo finito del sistema, y la comprobación, por exploración exhaustiva en su espacio de estados, de la propiedad deseada. Dentro de esta técnica se pueden diferenciar dos tendencias:

- *Verificación heterogénea.* También conocida como *temporal model checking* fue desarrollada de forma independiente por Clarke y Emerson (Clarke y Emerson, 1981) y por Queille y Sifakis (Queille y Sifakis, 1982). Se caracteriza porque, en este caso, el conjunto de propiedades es expresado utilizando alguna lógica temporal o modal, mientras que el sistema sobre el que se realizan las tareas de verificación suele modelarse utilizando un autómata de estados finitos.

- *Verificación homogénea.* En este caso, por el contrario, tanto el conjunto de propiedades como el sistema a verificar son expresados utilizando la misma notación, normalmente máquinas de estados finitos. El criterio de corrección se basa en el establecimiento de algún tipo de relación de equivalencia entre ambos: equivalencias de orden (Cleaveland et al., 1993; Roscoe, 1994), equivalencia observacional (Cleaveland et al., 1993; Fernández et al., 1996), etc.

Esta técnica presenta numerosas ventajas respecto a la utilización de demostradores de teoremas, entre ellas destacan: su rapidez, la posibilidad de trabajar con sistemas incompletos, y la generación de contraejemplos.

Su principal inconveniente radica precisamente en la raíz del método: la exploración exhaustiva del conjunto de estados del sistema. Esto obliga a que el modelo con el que se trabaje sea un modelo de estados finito. El establecimiento de este tipo de modelos, evitando la explosión de estados, es uno de los aspectos en los que se centran las investigaciones actuales en este campo. Como antecedentes, los trabajos de MacMillan, utilizando los BBDs (*ordered Binary Decision Diagrams*) obtenidos por Bryant (Bryant, 1986) que permitían una representación eficiente de las transiciones entre estados; la reducción de estados propuesta por (Kurshan, 1994); y la minimización semántica (Elseaidy et al., 1997) que elimina los estados no necesarios del modelo de un sistema.

Simulación y banco de pruebas

Otros enfoques menos formales ante la verificación de sistemas son la simulación y los bancos de pruebas. Se han englobado estas dos técnicas en un mismo apartado porque comparten la misma filosofía, aunque las diferencia un importante matiz: la primera trabaja sobre un modelo del sistema, y la segunda sobre el propio sistema.

Ambas técnicas trabajan con los resultados de comportamiento obtenidos tras someter al modelo del sistema o al propio sistema a un conjunto de pruebas o escenarios predefinidos. Dado que es prácticamente imposible abarcar todos los escenarios representativos de comportamiento del sistema, no puede asegurarse en ningún caso que éste sea correcto. Sin embargo, estas técnicas son un buen complemento al *model checking* y a los demostradores de teoremas, ya que son más rápidas, en principio menos costosas, y proporcionan una buena orientación a propósito de la calidad del sistema, entendiendo como calidad el grado de satisfacción de los requisitos funcionales del usuario.

6.4 El proceso software sin reutilización

En este apartado se resumirán las características más destacadas del modelo de proceso software ya existente (Pazos-Arias, 1995; Gil-Solla, 2000; García-Duque, 2000) sobre el que se plantea la inclusión de un proceso de reutilización. Dicho proceso consta de las tres fases indicadas en la figura 6.1 en las que se han integrado FDTs orientadas a propiedades y FDTs constructivas para lograr un ciclo de vida totalmente formalizado.

La captura de requisitos es realizada en la primera fase —fase de requisitos iniciales— y como resultado se debería obtener una especificación consistente y completa. Dado que el usuario no suele saber con exactitud la funcionalidad que desea en esta primera fase, se le proporciona un prototipo rápido del sistema especificado hasta el momento para que, viendo así su funcionalidad, pueda completarla o modificarla. Para evitar la ambigüedad inherente a la utilización

de lenguaje natural, comúnmente utilizado en esta fase, se hace imprescindible la formalización del proceso. Dentro de las técnicas formales existentes, se ha optado por la utilización de una FDT orientada a propiedades, más apropiada en esta etapa ya que de este modo el usuario expresa la funcionalidad del sistema como un conjunto de propiedades que éste debe satisfacer.

La segunda fase —fase de refinamiento— parte de la especificación de la funcionalidad obtenida de la primera. Es, en este punto, donde se hace necesaria una FDT constructiva que permita diseñar la arquitectura del sistema: sus componentes, las interfaces entre ellos, y su interacción. Una vez obtenida una arquitectura inicial se entra en un proceso de refinamientos sucesivos, realizados en el entorno LIRA (L*otos* I*terative* R*easoning* A*id*) (Pazos-Arias, 1995; Gil-Solla, 2000), que concluye con una arquitectura ya muy detallada que puede ser traducida cuasidirectamente a código fuente.

Por último, la fase de mantenimiento puede ser asumida como una fase de desarrollo propiamente dicha, donde pueden modificarse los requisitos iniciales del sistema y refinar de algún modo su arquitectura, es por esto que puede decirse que no tiene entidad propia.

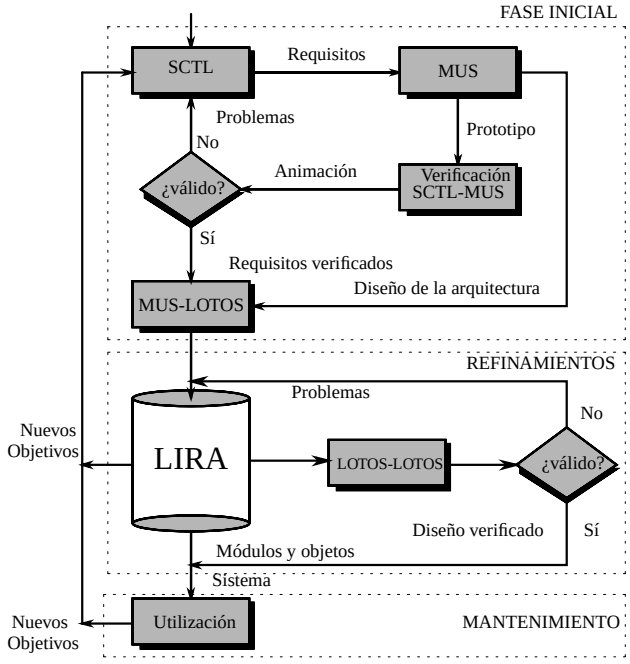


Figura 6.1. Metodología SCTL-MUS

Especificación formal de requisitos

Tradicionalmente los requisitos funcionales son expresados por el usuario utilizando lenguaje natural. Con el objetivo de formalizar esta captura de especificaciones, sin alejarse demasiado de la semántica del lenguaje natural, se ha utilizado una lógica temporal especialmente ideada para este entorno llamada SCTL (S*imple* C*ausal* T*emporal* L*ogic*) (García-Duque, 2000; Pazos-Arias y García-Duque, 2001). La inclusión de los conceptos de temporalidad y causalidad permiten expresar al usuario proposiciones genéricas muy cercanas al lenguaje natural del tipo:

"*Si es posible (premisa), entonces (en un instante temporal) debe ser posible (consecuencia)*"

Pero la verdadera aportación de esta lógica radica en la inclusión de la idea de **subespecificación**, acercándose así al carácter incompleto típico de esta fase de desarrollo. De esta forma es posible capturar como *subespecificados* todos aquellos eventos sobre los que el usuario no tenga claro todavía si deben ser permitidos o no, expresando así la posibilidad de evolución hacia un estado *verdadero* o *falso*. En el apartado 9.1 se realiza una descripción más detallada de esta FDT orientada a propiedades.

Prototipado rápido

Con el conjunto de propiedades o requisitos expresados por el usuario en SCTL se generará un modelo del sistema actual que actuará como prototipo sobre el que trabajar en un proceso iterativo e incremental. De esta forma el usuario podrá añadir, eliminar o modificar la funcionalidad que crea apropiada hasta llegar a un prototipo adecuado. Este modelo o prototipo se construye utilizando un formalismo llamado MUS (M*odel of* U*nspecified* S*tates*) (García-Duque, 2000; Pazos-Arias y García-Duque, 2001) basado en una máquina de estados finitos.

La principal aportación de este formalismo es la inclusión del concepto de subespecificación recogido en la etapa de captura de requisitos. Básicamente un modelo MUS consiste en un autómata o grafo dirigido expresando el conjunto de estados del sistema, el conjunto de transiciones entre dichos estados y los eventos posibles —etiquetas de las transiciones—, los eventos no posibles y aquellos que todavía están por especificar. Además de expresar de una forma amigable el comportamiento del sistema, MUS permite también realizar verificación formal, utilizando técnicas de *model checking* heterogéneo, de propiedades o requisitos sobre el modelo del sistema. Esta labor de verificación es vital en las iteraciones producidas en la fase de captura de requisitos iniciales. Cada vez que un usuario desee añadir una nueva propiedad al sistema, se verificará dicha propiedad en el prototipo actual, pudiendo obtenerse como resultados:

– la inconsistencia de dicha propiedad con el modelo actual, con lo que será necesaria su reformulación, o la de alguno de los requisitos precedentes;

– podría ocurrir que el prototipo ya satisfaga esta propiedad, con lo que no se produciría ningún avance en la especificación de requisitos, o bien

– estamos ante un sistema incompleto, con lo que será necesario añadir la propiedad al prototipo, sintetizándolo de nuevo.

Una descripción más detallada de este formalismo puede consultarse en el apartado 9.2.

Refinamientos sucesivos

Una vez que el usuario considere que el prototipo MUS obtenido es el más apropiado a sus necesidades, el proceso continuará con la definición de la arquitectura del sistema. Para ello es necesario recurrir a una FDT constructiva, como es el caso de LOTOS (ISO, 1989; Bolognesi y Briksma, 1989). La traducción MUS–LOTOS es automática, sirviendo así el formalismo MUS como un paso intermedio entre los dos tipos de FDTs utilizados: orientada a propiedades (SCTL) y constructiva (LOTOS).

La arquitectura inicial expresada en LOTOS se someterá a una serie de refinamientos sucesivos en el entorno transformacional LIRA.

CAPÍTULO 7

Ubicación del trabajo y objetivos

Hoy en día nadie discute las ventajas de la reutilización de componentes software, especialmente de aquella información de mayor nivel de abstracción. Las discrepancias radican a la hora de implementar un entorno que permita el manejo eficiente y eficaz de dicha información: ¿qué información reutilizar? y ¿cómo gestionarla?. En la actualidad es difícil dar una respuesta general a estas preguntas, haciéndose necesario particularizarlas para cada uno de los ámbitos de aplicación posibles, lo que se conoce como reutilización vertical (apartado 1.3).

7.1 ¿Qué información reutilizar?

Tras el estudio de las peculiaridades del modelo de ciclo de vida sobre el que se va a trabajar (apartado 6.4), se ha constatado que uno de los puntos débiles del modelo es la elevada frecuencia de ejecución del algoritmo de verificación. La utilización de técnicas de *model checking* para la verificación implica la exploración exhaustiva del espacio de estados del modelo del sistema en cada etapa del proceso. Además, esta multiplicación de ejecuciones, respecto a otros ciclos de vida, incrementa notablemente la cantidad de recursos necesarios para el proceso de desarrollo, revertiendo en un detrimento de su eficiencia. Las razones expuestas justifican la decisión de reutilizar la información de verificación asociada a modelos de sistemas (completos o incompletos), con el objetivo de aligerar esta carga computacional.

Pero ésta no es la única circunstancia donde la reutilización de información es deseable, sino que a la hora de comenzar con el proceso de creación de un nuevo sistema también sería conveniente poder partir de un modelo sobre el que se haya trabajado previamente y que sea similar al deseado. De esta forma se

incrementaría la eficiencia del proceso, ya que se estarían reduciendo las tareas de traducción SCTL-MUS, de síntesis y de verificación.

Así que se podría particularizar la respuesta a la primera de las preguntas realizadas —¿qué información reutilizar?— en nuestro entorno de trabajo:

– Reutilización de la información de verificación asociada a los modelos, completos o incompletos, sobre los que se haya trabajado con anterioridad.
  Objetivo: aligerar la elevada carga computacional que supone la verificación formal basada en *model checking* en un entorno iterativo e incremental como el propuesto.

– Reutilización de los propios modelos de sistema.
  Objetivo: reducir las labores de traducción SCTL-MUS, de síntesis y de verificación al comienzo de la creación de un nuevo sistema.

Consecuentemente, nosotros estamos planteando un entorno de reutilización basado en la composición de elementos, ya que esta metodología se adapta de una forma más natural a nuestro proceso de desarrollo software que la reutilización por generación.

Aunque podría pensarse que estamos proponiendo la reutilización de dos tipos de componentes, nosotros hemos reunido ambas características en uno solo, un único componente que almacene tanto su descripción funcional (SCTL), como un grafo MUS que modele su comportamiento, y el conjunto de propiedades verificadas sobre el modelo junto con sus correspondientes grados de satisfacción (consultar apartado 12.2).

Dado que la principal dificultad de todo entorno de reutilización radica en la gestión que se realice de la información a reutilizar, en este capítulo se detallan, en el apartado 7.3, las características más relevantes de distintos sistemas de gestión ya existentes. De esta forma, es posible cotejar nuestra propuesta (apartado 7.4) con los sistemas descritos y destacar cuáles son las principales aportaciones de la gestión que nosotros realizamos.

7.2 ¿Cómo gestionar la información?

A la hora de plantear la implementación de cualquier mecanismo de reutilización de información es imprescindible el establecimiento de un procedimiento eficiente y eficaz de gestión de dicha información. Hay que notar que en estas circunstancias estos condicionantes son vitales ya que es imprescindible poder satisfacer la premisa básica de la reutilización: *reutilizar ha de ser menos costoso que crear desde cero*. Aunque esta premisa puede considerarse obvia, lo que no lo es en absoluto es su satisfacción. Para poder cumplir este requisito básico no

basta con tener bien definida la información a reutilizar, sino que habrá que mantener dicha información clasificada y almacenada de forma tal que su recuperación presente una opción viable.

7.3 Enfoques de la gestión de componentes reutilizables

En la parte I de revisión bibliográfica de esta tesis se han explicado a grandes rasgos las principales líneas de investigación relacionadas con la reutilización de software: reutilización por composición, reutilización por generación, ingeniería de dominio, y análisis de la madurez y costes de la implantación de sistemas de reutilización software. Dentro de la reutilización por composición, es la gestión de la base de datos o biblioteca de componentes la tarea considerada el *talón de Aquiles*, y es por esto, que en este apartado vamos a sumergirnos precisamente en la descripción y análisis de otros trabajos que proporcionen al lector diferentes puntos de vista y enfoques, y que nos permitan a nosotros realizar una posterior labor de cotejo con nuestra propuesta.

La mayor parte de los mecanismos de recuperación se basan en el mismo principio, esto es, se intenta caracterizar mediante un patrón o perfil cada uno de los componentes reutilizables, de forma que éste reúna sus principales atributos. De esta manera, se realiza una representación interna que sentará las bases de su clasificación y posterior recuperación. Las principales diferencias entre los trabajos consultados radican en la forma en que estos perfiles son elaborados y su posterior aplicación, de hecho puede constatarse que actualmente existen propuestas que se pueden enmarcar en dos grandes líneas: aquellas que se basan en una descripción textual del componente y aquellas que utilizan una descripción formal del mismo.

7.3.1 Gestión basada en métodos semánticos

Girardi e Ibrahim (Girardi y Ibrahim, 1994) proponen el entorno ROSA (R*euse* O*f* S*oftware* A*rtifacts*). Este sistema de clasificación y recuperación de componentes reutilizables se basa en el procesado de las descripciones realizadas en lenguaje natural sobre cada componente. El motor de clasificación y búsqueda extrae información léxica, semántica y sintáctica que es utilizada para crear una representación interna de dichos componentes. El proceso de indexado es totalmente automático, reduciéndose así los costes de creación de las bibliotecas y evitando la presencia de personal especializado en las tareas de clasificación de componentes. Además de los problemas típicos inherentes a la utilización de lenguaje natural —ambigüedad, incompletitud e inconsistencia—, la eficacia de este sistema se basa totalmente en las descripciones y en el diccionario utilizado por el motor de búsqueda y clasificación.

Maarek et al. (Maarek et al., 1991) (proyecto GURU) construyen una biblioteca de componentes reutilizables en dos pasos: primero se extraen, de forma automática, los atributos de cada elemento, procesando para ello la documentación asociada a los mismos; y, una vez finalizada esta etapa, se genera, también automáticamente, una jerarquía de componentes utilizando técnicas de *clustering*. Como la representación interna de los componentes es obtenida por el análisis de textos, este sistema sigue manteniendo una fuerte dependencia de la calidad de la documentación, aunque bien es cierto que consigue evitar totalmente la interactuación con el usuario. Por último, notar que el establecimiento de una jerarquía de componentes mediante *clusters* provoca la total reorganización de la base de datos cada vez que un nuevo componente es almacenado, lo que revierte en una reducción de la eficiencia.

El método de facetas propuesto por Prieto-Díaz (Prieto-Díaz y Freeman, 1987; Prieto-Díaz, 1991) también se apoya totalmente en una descripción textual de las características de los componentes almacenados, aunque con una gran salvedad frente a las metodologías mencionadas anteriormente, en este caso el vocabulario utilizado para la caracterización interna está totalmente acotado. Según este sistema, un componente es descrito según los valores o atributos asociados a una característica determinada, o faceta. El hecho de restringir el vocabulario para las definiciones evita algunos problemas de ambigüedad, aunque la capacidad expresiva está claramente limitada por el conjunto de palabras reservado. En este entorno sí será vital la presencia de usuarios especializados, para poder así lograr una eficacia mayor en las búsquedas.

Ribeiro y Martins (Nestor-Ribeiro y Mário-Martins, 1995) proponen una representación lógica de los componentes, llamada AO (A*bstract* O*bjects*), basada en el método de facetas propuesto por Prieto-Díaz. La diferencia radica en que en este trabajo se propone modificar la estrategia de comparación, de forma que ésta no sea demasiado restrictiva y permita técnicas de *fuzzy logic*. Las consultas se abordan mediante un grafo conceptual donde los posibles valores de las facetas están relacionados entre sí a través de una distancia específica, llamada distancia semántica.

Los trabajos aquí detallados son sólo una pequeña muestra de la investigación más destacable realizada sobre este campo, que sigue siendo una línea floreciente aún a pesar de los problemas de equivocidad derivados de la utilización de lenguaje natural. Varios son los argumentos esgrimidos por sus defensores, de entre ellos podemos extraer algunos que favorecerían su implantación en entornos industriales: los usuarios de estos sistemas de reutilización no precisan una preparación especialmente costosa, y en algunos entornos se logra la total automatización de los procesos de clasificación y búsqueda, con la consiguiente reducción en los costes de implementación.

Como principal inconveniente, además de los ya citados, merece ser destacado que estas metodologías pueden ser aplicadas para la reutilización de componen-

tes de bajo nivel de abstracción, especialmente código, pero su utilización para la reutilización de componentes de un nivel abstracto más elevado es más que cuestionable.

7.3.2 Gestión basada en métodos formales

La descripción formal de los componentes reutilizables establece la base de actuación de la segunda gran línea de trabajo en este ámbito. Los entornos de reutilización de componentes software basados en métodos formales intentan solventar los problemas que tradicionalmente presenta la utilización de lenguaje natural, substituyendo las descripciones textuales y su análisis por descripciones totalmente formalizadas.

En todos los trabajos consultados hemos encontrado el mismo patrón de comportamiento. Cada componente mantiene asociada una descripción formal —donde se especifica su comportamiento más relevante— que será posteriormente utilizada para un cotejamiento formal de especificaciones (*specification matching*). El proceso de cotejo formal precisa de la caracterización de la relación entre un componente y la consulta mediante una fórmula lógica. Para comprobar la validez de dicha fórmula se recurre a un demostrador de teoremas y sólo si el resultado es positivo se considerará que el componente es adecuado. Tal y como es fácil deducir, el principal escollo para su implementación radica en el elevado número de pruebas que es preciso realizar en la base de datos, una por componente, así que normalmente los autores proponen un filtrado anterior que reduzca la cantidad de componentes a analizar.

Zaremski y Wing (Zaremski y Wing, 1997) proponen la reutilización de componentes de código (funciones y módulos) cuya descripción se realiza utilizando el lenguaje Larch/ML. Para la comparación formal de especificaciones recurren al demostrador interactivo Larch. Con el objetivo de reducir en lo posible las ejecuciones del demostrador incluyen un primer filtrado de los componentes que permita así trabajar con un subconjunto de los almacenados en la base de datos. Este primer filtrado excluye del grupo de trabajo a aquellos componentes que obviamente no encajan en la consulta. Para ello definen condiciones previas y condiciones de finalización para cada función (condiciones que han de satisfacerse antes y después de la ejecución de dicha función), que se hacen automáticamente extensibles a los módulos, que pueden verse como agrupaciones de funciones. De esta manera se eliminan del conjunto de estudio aquellos componentes que no satisfacen estas condiciones iniciales y finales.

El entorno de trabajo REBOUND (RE*use* B*ased* O*n* UND*erstanding*) fue propuesto por Penix y Alexander (Penix y Alexander, 1999; Penix y Alexander, 1997) para facilitar y automatizar las labores de recuperación y adaptación de componentes reutilizables. Como en el caso anterior este entorno está especialmente adaptado a la reutilización de componentes de código que mantienen una descripción formal en lenguaje Larch, como representación interna para las labores

de recuperación, sin embargo en este caso el demostrador de teoremas utilizado (HOL) permite que el cotejo formal se realice de forma cuasiautomática. Para reducir el número de componentes sobre los que realizar estas labores de cotejamiento y mejorar la eficiencia del motor de recuperación, los autores proponen una selección previa basada en un conjunto de heurísticos proporcionados por la semántica de las descripciones de los componentes. Esta preselección es realizada tras la clasificación de todos los componentes de la base de datos en función de la semántica expresada en la consulta. Aunque en este entorno se consigue la casi total automatización de las tareas de búsqueda, la eficiencia se ve reducida debido a una ardua tarea de clasificación dinámica, en función de la consulta, lo que obliga a ordenar la base de datos cada vez que se pretenda localizar un componente apropiado. REBOUND se ve completado con una parte de adaptación de componentes para que estos satisfagan totalmente las premisas de la consulta (Penix, 1999).

Jeng y Cheng (Jeng y Cheng, 1993; Cheng y Jeng, 1995) proponen una biblioteca de componentes de código basada en la especificación formal de los mismos y de las consultas utilizando OSPL (O*rder*-S*orted* P*redicate* L*ogic*). La clasificación de elementos se implementa utilizando dos jerarquías: una de bajo nivel que ordena los componentes según relaciones de generalidad (componentes cuyos métodos o funciones están totalmente incluidos en los de su inmediato superior); y una de más alto nivel que agrupa los elementos superiores, resultado de la ordenación anterior, según relaciones de similitud entre especificaciones, utilizando para ello técnicas de *clustering*. El proceso de recuperación se realiza precisamente a la inversa, comenzando por una primera identificación del *cluster* más conveniente y después refinando la búsqueda dentro de la jerarquía más inferior. El cotejo de especificaciones es llevado a cabo aplicando un demostrador de teoremas —desarrollado por Chang y Lee (Chang y Lee, 1973). Es necesario notar que el entorno propuesto se ha mejorado para que, además de la descripción funcional de los componentes, sea tenida en cuenta también las propiedades de su arquitectura, para ello los autores recurren a la aplicación de LOTOS como técnica de descripción formal más apropiada (Chen y Cheng, 1997).

Schumann y Fischer proponen el entorno de trabajo NORA/HAMMR (NORA H*ighly* A*daptive* M*ulti*M*ethod* R*etrieval* tool) (Schumann y Fischer, 1997) donde la recuperación de componentes de la base de datos es implementada como una concatenación de filtros que tratan de optimizar la compatibilidad del elemento recuperado con el entorno donde se pretende insertar. El proceso comienza con un primer estudio de compatibilidad, basado en la especificación VDM de los componentes (Fischer et al., 1995); a este primer filtrado le seguirán otros de rechazo, que intentan eliminar los componentes que claramente no se acoplen bien a la consulta, aplicando técnicas de *model-checking*; y, por último, se utilizarán filtros de confirmación utilizando demostradores de teoremas (Setheo). Los autores han optado, en este caso, por diversificar las técnicas de verificación formal utilizadas en el cotejo formal de especificaciones y reducir así la aplicación de

demostradores de teoremas. Uno de los principales problemas con los que se han encontrado es a la hora de tratar especificaciones recursivas, ya que en este caso el cotejamiento habría de hacerse aplicando técnicas de inducción que Setheo no soporta.

Los trabajos presentados en este apartado aunque no son los únicos, sí nos han parecido los más característicos en este ámbito. Como nexo a todos ellos, la aplicación de demostradores de teoremas, con los importantes problemas de automatización que conllevan, ya que ciertas relaciones que el razonamiento humano pasaría por alto —*trivial* o *análogo a*— han de ser explicitadas en la herramienta. Si se opta por una demostración interactiva en la que el usuario colabore, es vital que éste tenga una formación especializada; y, en otro caso, optando por una demostración totalmente automatizada, es prácticamente imposible cubrir todos los aspectos del proceso en un tiempo razonable.

7.3.3 Otros enfoques

Además de los trabajos que claramente encajan en alguna de las dos líneas detalladas anteriormente, hemos encontrado otros enfoques alternativos que plantean la gestión de la biblioteca de forma muy diferente.

El trabajo expuesto por Fischer en (Fischer, 1998) propone una alternativa a la automatización de las labores de gestión de la base de datos, en su lugar será el usuario quien, ayudado de un navegador especializado, recorra la biblioteca de componentes buscando aquél que mejor se adecue a sus necesidades.

Por otra parte, en (Atkinson, 1997) el autor propone una recuperación un tanto diferente, *behavioral retrieval*, basada en los resultados de ejecución de los componentes dentro del marco donde se pretende insertar aquél más apropiado. Es decir, no se establece una clasificación de componentes basada en su idiosincrasia sino que lo que se pretende es ordenar los resultados de ejecutar cada uno de ellos sobre el programa donde se desea insertar el más apropiado y, a partir de estos resultados, obtener aquél que sea más semejante al deseado. Por supuesto este método presenta varios inconvenientes: por un lado está especialmente ideado para la recuperación de componentes de código —con la consecuente reducción de beneficios—, además implica que cada consulta provoque la ejecución de todos y cada uno de los componentes de la biblioteca y, posteriormente, la ordenación de todos y cada uno de los resultados obtenidos. Esto provoca una ralentización excesiva, sobre todo teniendo en cuenta el mínimo grado de abstracción de los componentes recuperados. Además no es posible aplicar esta metodología sobre sistemas reactivos, de los que no se puede extraer un par *entrada-salida* definido.

Así como en la reutilización de componentes software es sencillo encontrar bibliografía y trabajos ya desarrollados, en el caso de la reutilización de información de verificación esta labor ya no es tan inmediata. De nuestras pesquisas sólo hemos podido concluir el trabajo expuesto por Keidar et al. en (Keidar et al.,

2000). Estos autores, aunque proponen la reutilización de resultados de verificación, se centran en la recuperación de resultados de simulación sobre componentes de código (algoritmos).

7.4 Nuestra propuesta de gestión

Dado que el proceso de desarrollo software sobre el que se va a trabajar está totalmente formalizado, también el proceso de reutilización que proponemos (Díaz-Redondo y Pazos-Arias, 2001) se enmarcará dentro de la línea de trabajo que opta por una representación formal de los componentes reutilizables. Sin embargo, a diferencia con los trabajos citados en el apartado 7.3.2, nuestra intención no es utilizar esta descripción formal como una mera representación interna que permita indexar en la base de datos para poder recuperar, en última instancia, componentes de bajo nivel de abstracción, sino que nosotros proponemos un criterio de recuperación **orientado a contenidos**. De esta manera, la descripción formal de la funcionalidad de un componente (modelo MUS) es, simultáneamente, índice y objetivo de la búsqueda, consiguiendo así reutilizar elementos software de un elevado nivel de abstracción.

Partiendo de que nuestra propuesta nace, en parte, como un intento de aliviar la carga computacional del proceso de verificación formal en el entorno de desarrollo software propuesto, sería totalmente incongruente incluir labores de verificación en el proceso de reutilización. En lugar de ello, nosotros proponemos una **recuperación de componentes aproximada** que permita la selección eficiente de un conjunto de ellos funcionalmente similares a la consulta. Será el proceso de adaptación quien, analizando los componentes recuperados, analice la conveniencia de reutilizarlos o no; y quien sugiera las modificaciones oportunas para hacerlo. Esta solución puede considerarse una opción intermedia entre la localización exacta y totalmente automática, proporcionada por la inmensa mayoría de las propuestas consultadas, y una localización totalmente dirigida por el usuario.

Aunque nuestra propuesta de trabajo parte de una localización aproximada de componentes, evitando la verificación formal en todo el proceso, también vamos a enfocar dicho proceso de localización en dos fases: una primera de recuperación gruesa y una segunda más refinada. El principal motivo de mantener una **localización escalonada** es simplemente una cuestión de eficiencia. Esto nos permitirá mantener una ordenación o clasificación estática de los componentes dentro de la base de datos o biblioteca, es decir, su *posición* dentro de la misma sólo depende del resto de los componentes almacenados, no de la consulta que se haga en cada momento. Tras la primera recuperación gruesa, obtendremos un conjunto relativamente pequeño de componentes susceptibles de ser apropiados y será en la segunda fase de la búsqueda, más refinada, donde sí se organicen éstos en función de la consulta. De esta forma, estamos amalgamando dos tendencias bien dife-

renciadas: la ordenación estática o independiente de la consulta y la ordenación dinámica o totalmente adaptada a la misma, evitando así el principal inconveniente de la ordenación dinámica —la reconfiguración del grueso de componentes en cada consulta— sin perder la elevada precisión que proporciona.

7.5 Identificación de objetivos

Para abordar el trabajo descrito someramente en el apartado anterior, se hace imprescindible la división del mismo en diferentes tareas u objetivos:

- ✗ Dado que la recuperación de componentes se realizará atendiendo a criterios de proximidad o similitud funcional, entonces su clasificación en la base de datos se realizará según estos mismos criterios. Según esto, se hace imprescindible la definición de una o más relaciones de orden que permitan comparar la funcionalidad de los componentes.

- ✗ Los criterios de cotejo de funcionalidad entre componentes no son suficientes, se hace necesario cuantificar las diferencias de funcionalidad entre ellos. Según esto, será necesario definir una o varias distancias que permitan realizar esta cuantificación.

- ✗ Dado que intentamos reutilizar componentes próximos en funcionalidad a la consulta, será necesario realizar tareas de adaptación sobre los componentes recuperados para que satisfagan la funcionalidad requerida.

- ✗ Se pretende reutilizar información de verificación asociada a los componentes, es decir, los resultados obtenidos de la verificación de propiedades sobre los modelos. Para ello será necesario extender los resultados de verificación que se tenían sobre un estado del sistema (García-Duque, 2000), para ser capaces de almacenar diferentes grados de satisfacción de una propiedad, pero ahora sobre un modelo MUS.

7.6 Organización del documento

Este texto se ha organizado, básicamente, según los objetivos definidos en el apartado anterior, así se ha dividido en las partes siguientes:

- Una primera parte, parte I, donde se resumen las principales líneas de trabajo dentro de la reutilización de software: reutilización basada en composición, basada en generación, ingeniería de dominio y estudio de métricas de madurez y estimación de costes de procesos de reutilización.

– En la parte II se realiza la introducción al trabajo de tesis y los objetivos planteados; se plantean las modificaciones pertinentes sobre el ciclo de vida para que éste incluya tareas de reutilización (capítulo 8), y se reseñan aquellos conceptos formales básicos que permitirán una lectura apropiada del resto del trabajo.

– En el siguiente grupo de capítulos, parte III, se detallan las relaciones de equivalencia y de orden definidas entre los componentes reutilizables, definiendo, a partir de éstas, las distancias adecuadas que permitirán la cuantificación de diferencias funcionales.

– En los capítulos de la parte IV se detallan las tareas propias de la reutilización de componentes software: clasificación, búsqueda, selección y adaptación. De esta manera, partiendo de un conjunto de funcionalidad especificada bajo SCTL se podrá obtener un modelo MUS ya existente que esté próximo, funcionalmente hablando, y adaptándolo se conseguirá la funcionalidad solicitada.

– En la parte V se tratará la reutilización de la información de verificación asociada a los componentes reutilizables. Para ello, en el capítulo 15, se explica la extensión realizada sobre los grados de satisfacción de una propiedad sobre un estado para que, ahora, expresen los grado de satisfacción de dicha propiedad pero sobre la totalidad del grafo de comportamiento. Además se detallará el proceso que gestiona esta información y permite aprovecharla para otras verificaciones formales.

– Con objeto de aclarar las aportaciones de esta tesis, en la parte VI se incluye un ejemplo más extenso, donde se pueden observar las peculiaridades del proceso de reutilización.

– En la parte VII se resumen las conclusiones que hemos alcanzado con este trabajo y se enumeran algunas de las posibles líneas futuras de investigación más destacables en este campo.

– Para finalizar se recogen las referencias bibliográficas consultadas para la realización de este trabajo.

CAPÍTULO 8

Ciclo de vida con reutilización

En el capítulo 6 se describió brevemente el proceso software sobre el que se pretendía incluir un mecanismo de reutilización. Dicho proceso estaba totalmente formalizado y se caracterizaba, además, por ser iterativo e incremental. Es objetivo del presente capítulo detallar cómo se verá afectado el ciclo de vida de un sistema al incluir en su desarrollo tareas de reutilización.

8.1 Etapas del proceso software afectadas

Las tareas propias de un entorno de reutilización de software pueden catalogarse básicamente en dos grandes grupos:

– clasificación y almacenamiento; y

– recuperación y adaptación de componentes.

Dependiendo del tipo de componentes y de cómo sea el proceso software sin reutilización, la inclusión de estas tareas afectará en diferente medida al proceso de desarrollo —en los apartados 2.6 y 4.4 se han reseñado algunos ejemplos.

En nuestro contexto, el proceso software del que hemos partido consta básicamente de dos grandes etapas (consultar apartado 6.4): en la primera —fase de requisitos iniciales— el objetivo es la consecución de una especificación consistente y completa; en la segunda —fase de refinamiento— se parte de la funcionalidad especificada en la etapa anterior y se realiza la traducción a una FDT constructiva, LOTOS, que permita la realización de transformaciones sucesivas hasta obtener una arquitectura lo suficientemente detallada como para poder ser traducida a código fuente.

Nuestro trabajo se centra en la reutilización de modelos incompletos y su información de verificación asociada, así que realmente la única fase que se verá

afectada será la primera, o fase de especificación de requisitos iniciales. Más concretamente, las labores de reutilización afectarán a dos de las tareas propias de esta fase, destacadas en tono más claro en la figura 8.1.

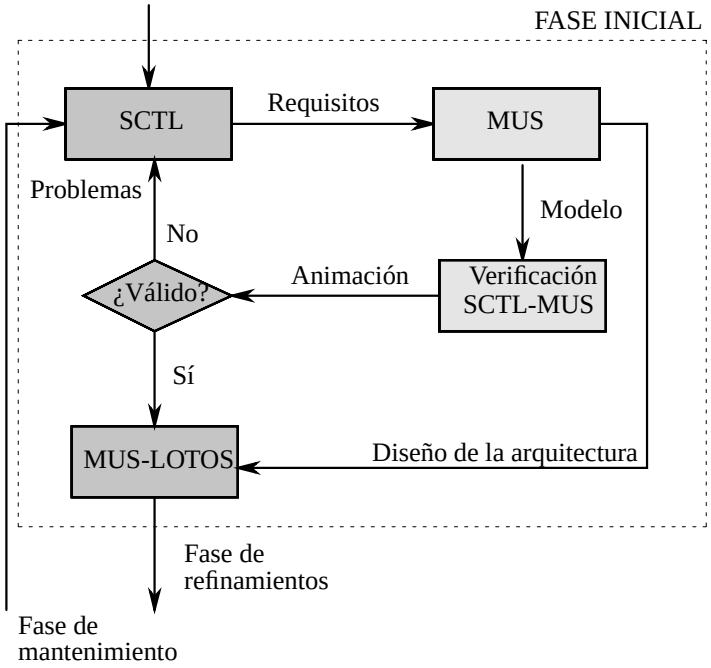


Figura 8.1. Fase inicial de especificación de requisitos

En los apartados siguientes se incorporarán al modelo indicado en la figura 8.1 las tareas propias del entorno de reutilización; en el apartado 8.2 se verá cómo afecta la clasificación y el almacenamiento de componentes reutilizables al ciclo de vida; y en el apartado 8.3 se detalla cómo se ve éste modificado por la reutilización de información de verificación y por la reutilización de modelos incompletos.

8.2 Tareas de clasificación y almacenamiento

La clasificación y almacenamiento de un componente reutilizable sólo podrá ser abordada, obviamente, una vez que se disponga de la información básica que lo conforma: modelo MUS e información de verificación.

El bloque *clasificación y almacenamiento* (en tono más claro en la figura 8.2) recibe como datos el modelo MUS del componente y su información de verificación asociada. Con esta información será necesario decidir si es conveniente su almacenamiento en la biblioteca y, si es así, se generará un componente reutilizable que será convenientemente clasificado y almacenado en la base de datos. Para

clasificar el componente se recurrirá a las relaciones de orden parcial definidas en el capítulo 10 y, una vez ordenado convenientemente, se almacenará en la biblioteca. Los detalles de este proceso se explican más extensamente en el capítulo 12.

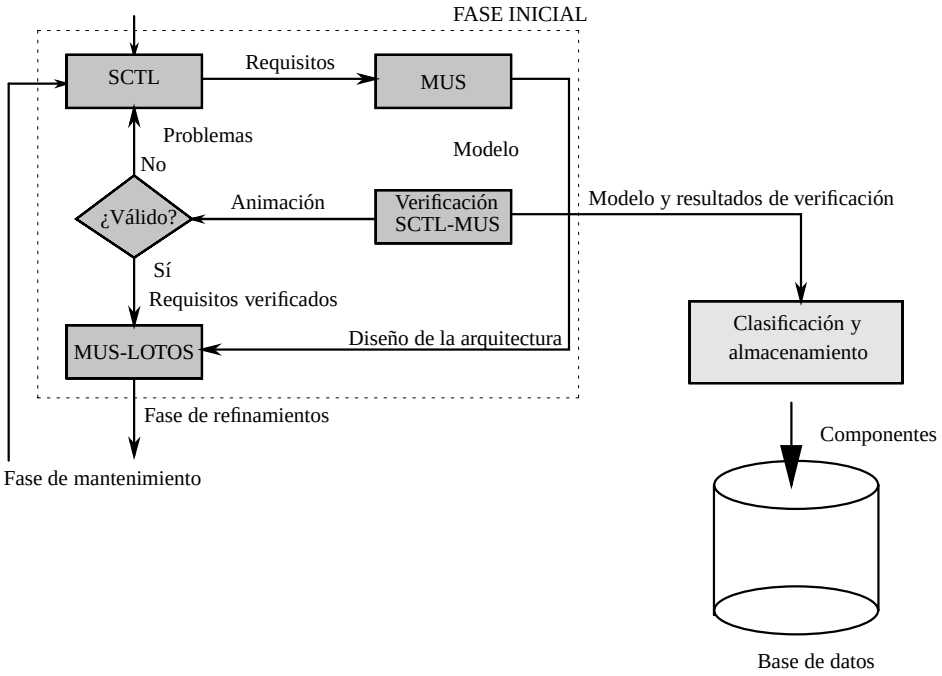


Figura 8.2. Clasificación y almacenamiento de componentes

8.3 Tareas de localización y adaptación

Así como las tareas de clasificación y almacenamiento pueden verse como un apéndice dentro del ciclo de vida, ya que no lo alteran en gran medida, las tareas de reutilización propiamente dichas —localización y adaptación—, por el contrario, sí lo modifican sustancialmente.

La reutilización de componentes se realiza en dos puntos diferentes dentro del proceso iterativo de la fase inicial que nos ocupa:

– en la verificación formal de propiedades, reutilizando información de verificación, y

– en la síntesis y generación de modelos MUS, reutilizando modelos incompletos.

Reutilización de modelos incompletos

En cada iteración del proceso, el usuario especifica nuevos requisitos funcionales que desea que satisfaga el sistema final. Estos requisitos deberán ser incorporados al modelo MUS que representa el comportamiento del sistema.

Al comienzo de este proceso no se dispone de ningún modelo MUS sobre el que trabajar y éste debe ser sintetizado a partir del primer conjunto de requisitos funcionales. Es en este punto donde se plantea la posibilidad de buscar un modelo MUS que sea funcionalmente semejante al deseado y que permita reducir la carga computacional en las labores de síntesis. En la figura 8.3 puede verse que se ha substituido el bloque *MUS* del ciclo de vida original (figura 8.1) por un conjunto de actividades que resumen este proceso de reutilización (cajas en tono más claro).

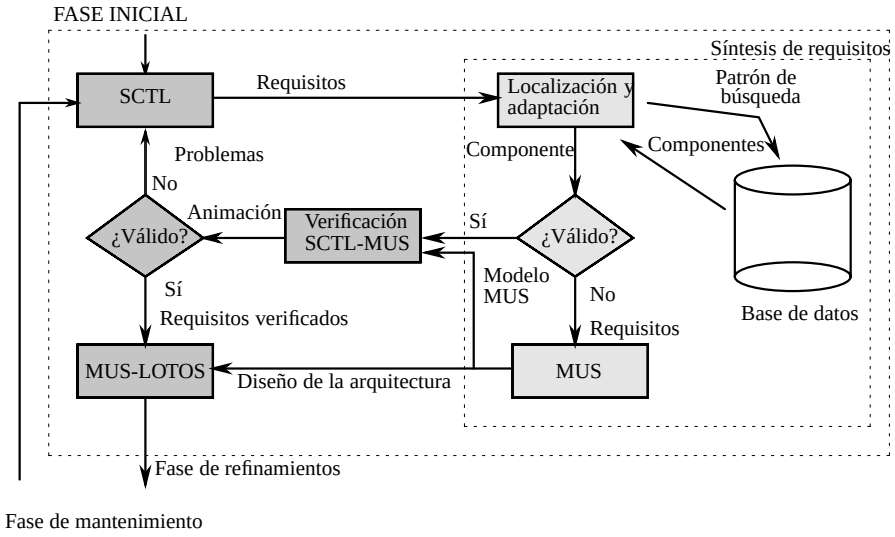


Figura 8.3. Reutilización de modelos incompletos

Partiendo de la funcionalidad especificada en los requisitos iniciales, se realizará la localización dentro de la base de datos de aquellos componentes más adecuados (en el capítulo 13 se detalla este proceso) a los que se someterá, posteriormente, a un proceso de adaptación (capítulo 14) hasta satisfacer la funcionalidad especificada. Si la localización y adaptación ha tenido éxito, el modelo adaptado es directamente el que se utilizará como prototipo inicial del proceso, en otro caso será preciso realizar la síntesis partiendo de cero.

Reutilización de información de verificación

Cada vez que se especifica un nuevo requisito funcional será necesario verificar que realmente éste es consistente con la funcionalidad acumulada hasta ese

momento. Es en este proceso de verificación formal donde se ha incluido la posibilidad de reutilizar información de verificación almacenada previamente. El bloque *verificación SCTL-MUS* de la figura 8.1, se ha substituido en la figura 8.4 por un conjunto de actividades que esquematizan este proceso (en tono más claro).

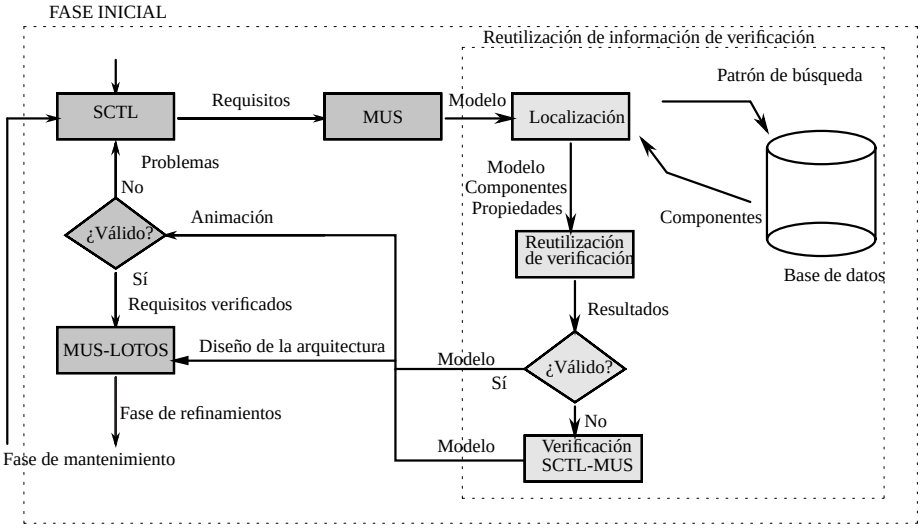


Figura 8.4. Reutilización de resultados de verificación

Partiendo del modelo actual del sistema y de la propiedad que se desea verificar se realiza una búsqueda en la base de datos con el objetivo de encontrar aquellos componentes sobre los que se haya verificado dicha propiedad anteriormente. Una vez obtenidos, éstos serán procesados para aprovechar toda la información de verificación útil. Si ha sido posible obviar las labores de verificación formal, el prototipo será directamente ofrecido al usuario para su validación, en otro caso será necesario abordar las labores de *model checking* complementarias. En la parte V se trata este tema más detalladamente.

8.4 Modelo de ciclo de vida con reutilización

Tras haber desglosado en los apartados anteriores el proceso de reutilización en tres partes diferentes, según la tarea a realizar y el punto del ciclo de vida donde se aplica, se pretende ahora reunir toda esta información para obtener un modelo de ciclo de vida incluyendo labores de reutilización. En la figura 8.5 puede verse la fase inicial de captura y especificación de requisitos incluyendo ya todas aquellas actividades propias de la reutilización de sistemas incompletos y su información de verificación.

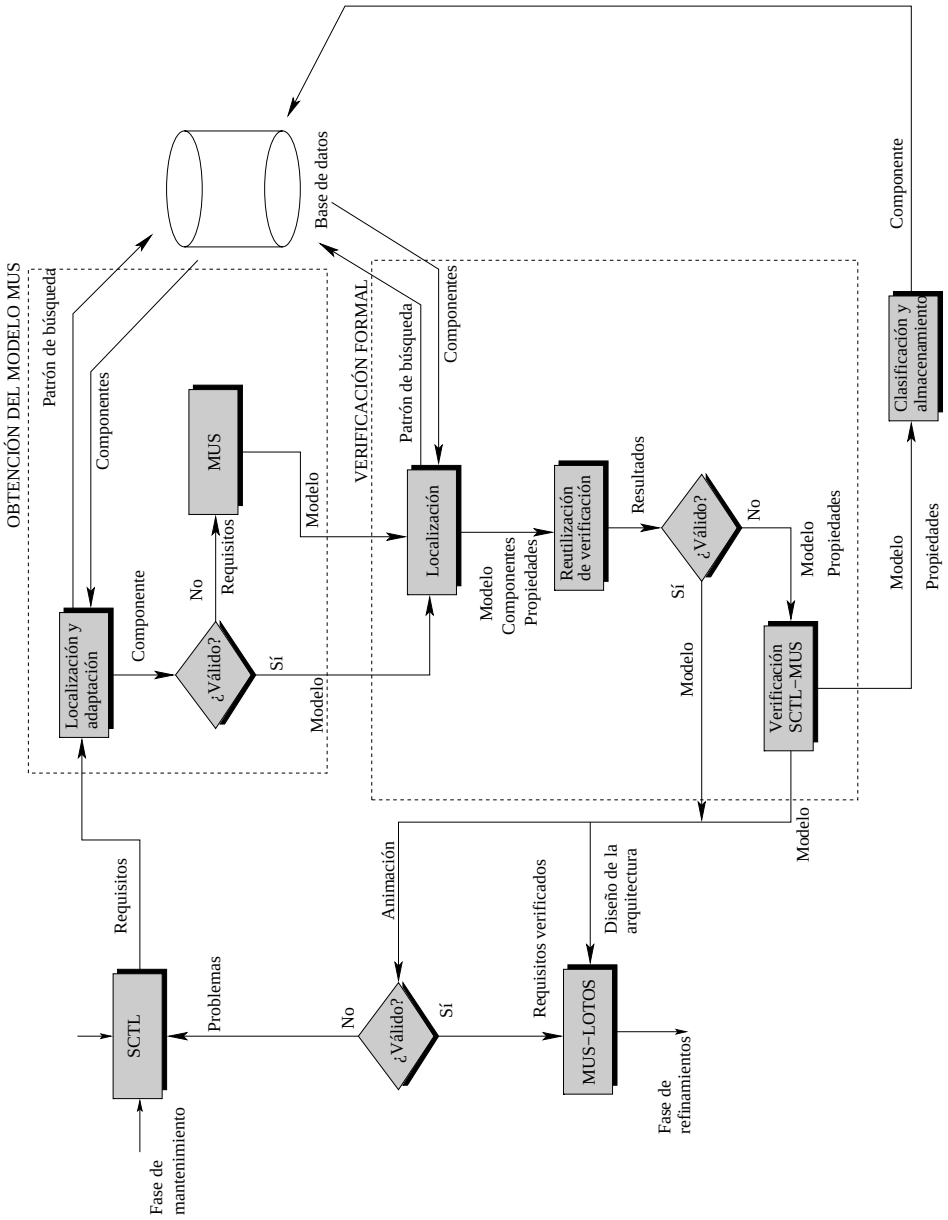


Figura 8.5. Modelo de proceso software incluyendo reutilización

CAPÍTULO 9

Bases formales: SCTL y MUS

En este capítulo se resumen brevemente los dos formalismos sobre los que se asienta el proceso de desarrollo software descrito en el capítulo 6. Esta reseña se hace imprescindible para la correcta lectura e interpretación de los capítulos posteriores. El capítulo se ha organizado en tres grandes subapartados: el primero dedicado íntegramente a la definición de la lógica SCTL; el segundo al modelo de estados subespecificados MUS; y un tercero donde se resumen las tareas de verificación de una propiedad SCTL sobre un estado de un grafo MUS y la colección y significado de los diferentes grados de satisfacción que es posible alcanzar.

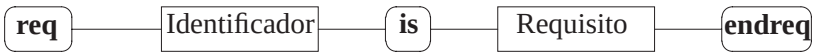
9.1 Lógica temporal causal simple: SCTL

Debido al carácter informal de la fase de especificación de requisitos y a que el usuario no suele especificar el sistema en una única etapa, sino que lo hace de forma gradual, es necesaria, en esta primera fase, una técnica de descripción formal orientada a propiedades, simple y cercana al lenguaje natural y que permita esta especificación incremental.

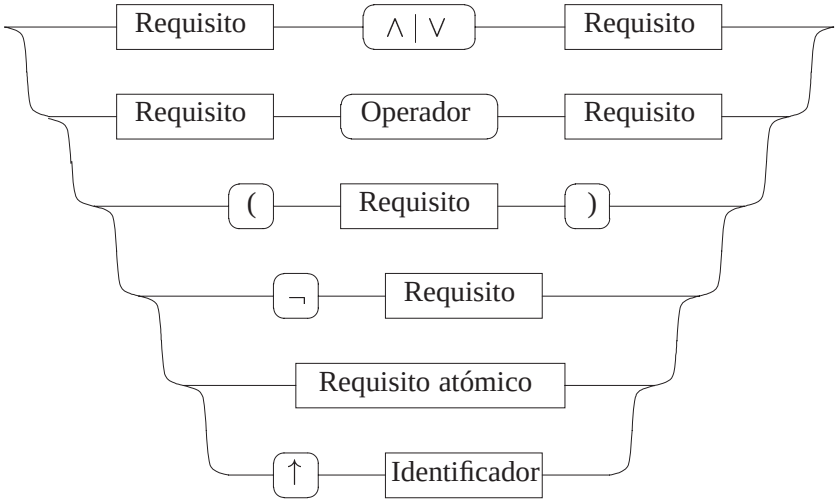
Tradicionalmente la utilización de una técnica de especificación formal hace necesario obtener en cada una de las fases de diseño especificaciones completas, esto supone considerar como *verdadero* aquello que el usuario especifica como cierto y *falso* aquello que especifica como tal y aquello sobre lo que no se ha pronunciado. Sin embargo, si se desea formalizar el proceso de especificación incremental es necesario que no sean considerados de igual forma aquellos requisitos no especificados que aquellos que han sido especificados como *falsos* o no posibles.

SCTL introduce entonces el concepto de subespecificación como un tercer estado frente a la lógica booleana clásica de dos estados. Así, un requisito no

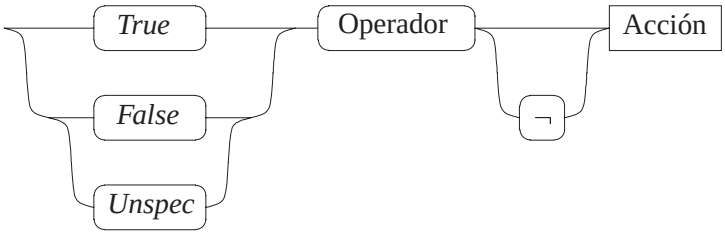
Requisito SCTL:



Requisito:



Requisito atómico:



Operador:

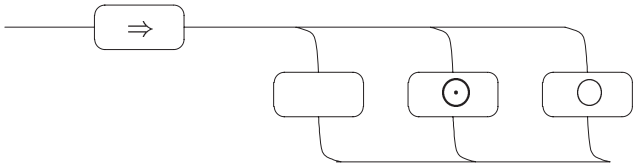


Figura 9.1. Sintaxis de la lógica SCTL

especificado adoptará un tercer estado, de subespecificación, con la peculiaridad de que su estado podrá evolucionar hacia *cierto* o *falso* a medida que la especificación del usuario progrese.

El proceso de especificación de requisitos, utilizando SCTL, parte de un sistema totalmente subespecificado, sin ningún tipo de restricción. A este sistema se le irán incorporando restricciones a medida que el usuario exprese requisitos funcionales. Una vez que el usuario considere que la especificación del comportamiento del sistema ha terminado, será necesario eliminar todas las subespecificaciones que puedan quedar. De esta forma, al final de esta primera etapa, se tendrá una especificación formal, consistente y completa.

9.1.1 Requisitos SCTL

Una fórmula SCTL R se construye combinando los elementos siguientes:

- Acciones pertenecientes a un conjunto $\Lambda = \{a_1, a_2, \ldots, a_m\}$.
- Operadores temporales pertenecientes al conjunto $\Theta \triangleq \{\Rightarrow, \Rightarrow\bigcirc, \Rightarrow\odot\}$.
- Operadores lógicos pertenecientes al conjunto $\Gamma \triangleq \{\wedge, \vee, \neg\}$.
- Constantes $\Psi = \{0, \frac{1}{2}, 1\}$.
- Identificadores de requisitos SCTL precedidos por el símbolo $\uparrow$.

Un requisito SCTL R se define como cualquier fórmula SCTL construida según la sintaxis definida en la figura 9.1. Generalmente se compone de una premisa, un operador temporal y una consecuencia que permiten combinar causalidad y temporalidad expresando condiciones del tipo: "*Si es posible (premisa), entonces (a la vez, antes o después) debe ser posible (consecuencia)*", donde tanto la premisa como la consecuencia son, a su vez, requisitos SCTL.

9.2 Modelo de estados subespecificados: MUS

Los diagramas de estados clásicos de modelado del comportamiento de un sistema se basan en el concepto matemático de grafo (Liu, 1985; Ross y Wright, 1992). Un grafo consta de un conjunto de estados o nodos, y un conjunto de arcos que unen dichos nodos. Cuando los arcos son dirigidos, es decir, especifican un estado de origen y uno de destino, se dice que el grafo es dirigido. Un sistema evoluciona de un estado a otro cuando se produce un determinado evento o acción observable. Así que, cada arco o transición, estará etiquetado con un evento o acción que, en caso de producirse, provocará la transición del sistema desde el estado origen al estado destino del arco.

El modelo de estados subespecificados, MUS, es básicamente un grafo dirigido clásico, al que se le ha añadido el concepto de subespecificación, enriqueciendo así su expresividad. De esta forma se recoge, además de los tradicionales valores *verdadero* o *falso*, uno nuevo, *subespecificado*, que asumen todos y cada uno de los elementos del grafo —estados, eventos y transiciones— en la fase inicial del diseño, y que evolucionará hacia los valores *verdadero* o *falso* a medida que el usuario refina la especificación.

Básicamente un grafo MUS g consta de un conjunto de estados que representan la evolución temporal del sistema modelado, $\mathcal{E} = \{E_0, E_1, \ldots, E_n\}$. La transición de uno a otro es provocada por la ocurrencia de una acción o evento $a_i \in \Lambda = \{a_1, a_2, \ldots, a_m\}$.

Cada evento observable del conjunto Λ puede ser especificado en todos y cada uno de los estados de g —denotándose como $E_j[a_i]$ a la especificación de la acción o evento a_i en el estado E_j— como un valor $\psi \in \Psi = \{0, \frac{1}{2}, 1\}$, donde 0 representa el valor *falso*, $\frac{1}{2}$ el valor *subespecificado* y 1 el valor *verdadero*. De esta forma, cada estado constará de un conjunto de eventos posibles, uno de eventos no posibles y un tercero de eventos subespecificados.

Se define entonces el conjunto $\mathcal{A}_t = \{(E_i, E_j, a_k) : E_i, E_j \in \mathcal{E}, a_k \in \Lambda \cup \{a_{sub}\}\}$ como el conjunto posible de arcos o transiciones en un grafo g, donde a_{sub} representa el hecho de que una transición no tenga todavía una acción o evento asociado, es decir, se trata de una transición subespecificada.

Definición 9.1. *Un sistema modelado según un grafo MUS $g \in \mathbb{G}$, siendo $\mathbb{G}$ el conjunto de todos los grafos posibles, se define como una tupla $(\mathcal{E}, \mathcal{T})$ donde $\mathcal{T} \subseteq \mathcal{A}_t$, satisfaciendo que no pueden existir dos arcos paralelos que unen el mismo par ordenado de estados y etiquetados con el mismo evento $a_i \in \Lambda$.*

9.3 Verificación de requisitos SCTL

En los sistemas clásicos, que utilizan lógicas binarias, el grado de satisfacción de una propiedad o requisito se reduce a un valor booleano —*verdadero* o *falso*. La lógica SCTL, al ser una lógica multivalorada donde se ha introducido el concepto de subespecificación, permite un mayor refinamiento en estos resultados, ampliando el abanico de posibilidades a seis, donde cuatro de estos valores son resultados intermedios entre la plena satisfacción y la no satisfacción de la propiedad. De esta forma se amplía la expresividad de la lógica y se permite la detección de errores en la versión actual del sistema e, incluso, errores potenciales, es decir, errores que se producirían con una determinada evolución en su diseño.

En el caso de una propiedad que no ha sido especificada en la versión actual del sistema, su grado de satisfacción no debería ser *falso* o *verdadero*, como ocurre en las lógicas binarias, sino que deberá ser *subespecificado* ya que, dependiendo

de cómo evolucione el diseño del sistema, podrá converger en un futuro hacia un grado de satisfacción *verdadero* o *falso*.

Dentro de estas propiedades subespecificadas, es posible diferenciar, a su vez, varios grados de subespecificación, ya que puede que la propiedad se encuentre parcialmente especificada, es decir, algunos elementos que la componen sí están plenamente especificados y otros no. El englobar toda esta casuística bajo un mismo criterio implicaría una reducción de la expresividad de la lógica y, es por esto, que se ha distinguido entre:

– Requisitos que no pueden llegar a ser *verdaderos*. Es decir, su grado actual de satisfacción es subespecificado, pero si se perdiese dicha subespecificación, evolucionaría hacia un grado de satisfacción *falso*.

– Requisitos que no pueden llegar a ser *falsos*. Totalmente equivalente al anterior, salvo que en este caso al perder subespecificación, se evolucionaría hacia un grado de satisfacción *verdadero*.

– Requisitos que no pueden ser *falsos* ni *verdaderos*. Estos requisitos no pueden ser estudiados, ya que no se satisface la condición de aplicabilidad de los requisitos SCTL. Este grado de satisfacción recibe el nombre de *contradictorio*.

– Requisitos que podrían llegar a ser *falsos* o *verdaderos* ya que están totalmente subespecificados y no se tiene ninguna información a propósito de su posterior evolución.

9.3.1 Álgebra de Incertidumbre del Punto Medio

Definición 9.2. *Se define el* **conjunto de grados de satisfacción de una propiedad** $\Phi = \{0, \frac{1}{4}, \widehat{\frac{1}{2}}, \frac{1}{2}, \frac{3}{4}, 1\}$*, y el conjunto de operadores lógicos internos a dicho conjunto* $\{\wedge, \vee, \neg\}$ *.*

Definición 9.3. *Se define la operación lógica* **or** *sobre el conjunto* Φ *(definición 9.2), denotándose como* $\vee$*:*

$$\begin{array}{ccc} \Phi \times \Phi & \longmapsto & \Phi \\ (a,b) & \longmapsto & a \vee b \end{array}$$

$\vee$	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
$\widehat{\frac{1}{2}}$	$\widehat{\frac{1}{2}}$	$\widehat{\frac{1}{2}}$	$\widehat{\frac{1}{2}}$	$\frac{3}{4}$	$\frac{3}{4}$	1
$\frac{1}{2}$	$\frac{1}{2}$	$\frac{1}{2}$	$\frac{3}{4}$	$\frac{1}{2}$	$\frac{3}{4}$	1
$\frac{3}{4}$	$\frac{3}{4}$	$\frac{3}{4}$	$\frac{3}{4}$	$\frac{3}{4}$	$\frac{3}{4}$	1
1	1	1	1	1	1	1

Definición 9.4. *Se define la operación lógica* **and** *sobre el conjunto* Φ *(definición 9.2), denotándose como* $\wedge$*:*

$$\begin{array}{ccc} \Phi \times \Phi & \longmapsto & \Phi \\ (a,b) & \longmapsto & a \wedge b \end{array}$$

$\wedge$	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	$\frac{1}{4}$	$\widehat{\frac{1}{2}}$	$\frac{1}{4}$	$\widehat{\frac{1}{2}}$	$\widehat{\frac{1}{2}}$
$\frac{1}{2}$	0	$\frac{1}{4}$	$\frac{1}{4}$	$\frac{1}{2}$	$\frac{1}{2}$	$\frac{1}{2}$
$\frac{3}{4}$	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 9.5. *Se define la operación monaria* **complemento** *sobre el conjunto* Φ *(definición 9.2), denotándose como* $\neg$*:*

$$\begin{array}{ccc} \Phi & \longmapsto & \Phi \\ a & \longmapsto & \neg a \end{array}$$

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

Definición 9.6. *Se define el álgebra IPM (álgebra de* Incertidumbre del *P*unto *M*edio*) como la cuádrupla* $(\Phi, \vee, \wedge, \neg)$*, pudiendo demostrarse que IPM es un álgebra de De Morgan.*

Definición 9.7. *Dados dos elementos* ϕ_1*,* $\phi_2 \in \Phi$*, se define la operación interna* **causal***, denotándose* $\rightarrow (a, b) = a \rightarrow b$*, como sigue:*

$$\begin{array}{ccc} \Phi \times \Phi & \longmapsto & \Phi \\ (a, b) & \longmapsto & a \rightarrow b \end{array}$$

$\rightarrow$	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

Esta operación interna satisface las siguientes propiedades:

$$a \rightarrow (b \wedge c) = (a \rightarrow b) \wedge (a \rightarrow c)$$
$$a \rightarrow (b \vee c) = (a \rightarrow b) \vee (a \rightarrow c)$$
$$\neg(a \rightarrow b) = a \rightarrow \neg b$$

9.3.2 Grados de satisfacción de requisitos SCTL en un estado del sistema

Dado un requisito SCTL R, necesitamos conocer su grado de satisfacción en un estado E_j de un grafo MUS g. Esta relación de satisfacción es consecuencia directa de la semántica de las proposiciones causales formuladas en SCTL: "*un requisito SCTL se satisface en un estado E_j si y sólo si se satisface su premisa y en los estados indicados por su operador temporal, se satisface su consecuencia*".

Definición 9.8. *Se define la* **relación de satisfacción** *como una función de $SCTL \times \{\mathcal{E}_g\}$ en Φ, que obtiene el grado de satisfacción de un requisito R en un estado E_j del grafo MUS g:*

$$\begin{array}{ccc} SCTL \times \{\mathcal{E}_g\} & \longmapsto & \Phi \\ (R, E_j) & \longmapsto & \phi = \vDash (R, E_j) \end{array}$$

donde $\phi \in \{0, \frac{1}{4}, \widehat{\frac{1}{2}}, \frac{1}{2}, \frac{3}{4}, 1\}$.

Definición 9.9. *Dado un requisito R compuesto por una premisa, un operador temporal y una consecuencia, se definen los* **estados de aplicabilidad** *de dicho requisito sobre un estado E_j, denotándose $\perp(R_{E_j})$, como el conjunto de estados en los que la premisa causa la consecuencia. Si el operador temporal es* a la vez *($\Rightarrow$) entonces el estado de aplicabilidad será el mismo estado E_j; si el operador temporal es* después *($\Rightarrow\bigcirc$) entonces el conjunto de los estados de aplicabilidad será el formado por todos aquellos estados hacia los que es posible evolucionar desde E_j a través de una transición posible y cuya acción asociada se encuentra expresada en la premisa de R; si el operador temporal es* antes *($\Rightarrow\bigodot$) entonces el conjunto de los estados de aplicabilidad será el formado por todos aquellos estados desde los que se puede evolucionar a través de una transición posible hasta E_j.*

Dada la sintaxis de un requisito R, expresada en la figura 9.1, se especifica a continuación cómo obtener la relación de satisfacción de dicho requisito en función de su estructura:

1. $R = a_j \in \Lambda$.
   $\vDash (R, E_i) = E_i[a_j]$

2. $R = \neg R_1$.
   $\vDash (R, E_i) \ = \ \neg \vDash (R_1, E_i)$

3. $R = (R_1 \wedge R_2)$.
   $\vDash (R, E_i) \ = \vDash (R_1, E_i) \ \wedge \ \vDash (R_2, E_i)$

4. $R = (R_1 \vee R_2)$.
   $\models (R, E_i) \;=\; \models (R_1, E_i) \;\vee\; \models (R_2, E_i)$

5. $R = (R_1 \Rightarrow R_2)$.
   $\models (R, E_i) \;=\; \models (R_1, E_i) \;\rightarrow\; \models (R_2, E_i)$

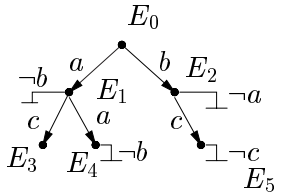
6. $R = (R_1 \Rightarrow\!\bigcirc\ R_2)$.
   $\models (R, E_i) \;=\; \models (R_1, E_i) \;\rightarrow\; \underset{E_i \in \perp(R_{E_i})}{\wedge} \models (R_2, E_i)$

7. $R = (R_1 \Rightarrow\!\odot\ R_2)$.
   $\models (R, E_i) \;=\; \models (R_1, E_i) \;\rightarrow\; \underset{E_i \in \perp(R_{E_i})}{\wedge} \models (R_2, E_i)$

La semántica de los valores Φ, relativos a los diferentes grados de satisfacción de un requisito R en un modelo MUS, es la siguiente:

- $\phi = 0$. El requisito no se satisface.
- $\phi = \frac{1}{4}$. El requisito está parcialmente especificado, pero no puede llegar a satisfacerse.
- $\phi = \widehat{\frac{1}{2}}$. La consecuencia del requisito no puede ser evaluada, ya que su premisa tiene un grado de satisfacción de $0, \frac{1}{4}$ ó $\widehat{\frac{1}{2}}$.
- $\phi = \frac{1}{2}$. El requisito está totalmente subespecificado.
- $\phi = \frac{3}{4}$. El requisito está parcialmente especificado, y no puede llegar a no satisfacerse.
- $\phi = 1$. El requisito se satisface en dicho estado.

EJEMPLO 9.1. En la siguiente figura se pueden apreciar los grados de satisfacción de los requisitos $R_1 \equiv (a \Rightarrow b)$ y $R_2 \equiv (c \Rightarrow a)$ en cada uno de los estados:



$\models (R_1, E_j)$	a	b
$\models (R_1, E_0) = 1$	1	1
$\models (R_1, E_1) = 0$	1	0
$\models (R_1, E_2) = \widehat{\frac{1}{2}}$	0	$\frac{1}{2}$
$\models (R_1, E_3) = \frac{1}{2}$	$\frac{1}{2}$	$\frac{1}{2}$
$\models (R_1, E_4) = \frac{1}{4}$	$\frac{1}{2}$	0
$\models (R_1, E_5) = \frac{1}{2}$	$\frac{1}{2}$	$\frac{1}{2}$

$\models (R_2, E_j)$	c	a
$\models (R_2, E_0) = \frac{3}{4}$	$\frac{1}{2}$	1
$\models (R_2, E_1) = 1$	1	1
$\models (R_2, E_2) = 0$	1	0
$\models (R_2, E_3) = \frac{1}{2}$	1	$\frac{1}{2}$
$\models (R_2, E_4) = \frac{1}{2}$	$\frac{1}{2}$	$\frac{1}{2}$
$\models (R_2, E_5) = \widehat{\frac{1}{2}}$	0	$\frac{1}{2}$

□

PARTE III

Relaciones y distancias entre componentes

CAPÍTULO 10

Relaciones de equivalencia y orden

Dados dos objetos cualesquiera, se pueden establecer tantas relaciones de equivalencia entre ellos como criterios se identifiquen: equivalentes en cuanto a color, a forma, a tamaño, a finalidad... En nuestro caso, perseguimos identificar aquellos criterios que nos permitan definir relaciones de equivalencia funcional entre componentes reutilizables. Además, tras identificar cuatro criterios diferentes, hemos podido organizar dichas relaciones de equivalencia en una estructura de retículo.

10.1 Introducción

Uno de los objetivos de este trabajo es la identificación de criterios que permitan el establecimiento de relaciones de equivalencia funcional y relaciones de contenido-continente funcional entre componentes. Para su consecución, nos hemos concentrado en las trazas de evolución de los grafos MUS. Estas trazas no son más que el abanico de opciones de comportamiento de un componente, así que su estudio nos permite acercarnos de una forma bastante fiable a la funcionalidad expresada en el modelo (van Glabbeek, 1990). Es necesario notar, entonces, que todas las relaciones de equivalencia y contenido definidas a lo largo de este capítulo, aunque expresadas siempre en función de grafos, se hacen automáticamente extensivas a los componentes correspondientes.

Tras el estudio de las trazas hemos llegado a identificar cuatro criterios diferentes que permiten la definición de cuatro relaciones de orden parcial y de equivalencia entre componentes. Además se ha podido establecer una relación de orden parcial entre estas cuatro perspectivas de observación diferentes, con el

consiguiente enriquecimiento de los nexos entre componentes (apartado 10.7). El tener un abanico más amplio de posibilidades a la hora de relacionar componentes reutilizables, permite una mayor flexibilidad a la hora de clasificarlos y recuperarlos de la biblioteca.

Nos interesará identificar relaciones o funciones $\mathcal{O}$ que asocien a cada grafo $g \in \mathbb{G}$ un conjunto $\mathcal{O}(g)$. Este conjunto $\mathcal{O}(g)$ constituirá el comportamiento observable del grafo g bajo determinado criterio. Para cada función o relación $\mathcal{O}$ se define la relación de equivalencia $=_{\mathcal{O}} \in \mathbb{G} \times \mathbb{G}$ dada por $g =_{\mathcal{O}} g' \Leftrightarrow \mathcal{O}(g) = \mathcal{O}(g')$. A partir de esta función que relaciona dos grafos, se define la relación de orden, normalmente parcial, $\sqsubseteq_{\mathcal{O}} \in \mathbb{G} \times \mathbb{G}$, donde $g \sqsubseteq_{\mathcal{O}} g' \Leftrightarrow \mathcal{O}(g) \subseteq \mathcal{O}(g')$. Naturalmente se satisface que $g =_{\mathcal{O}} g' \Leftrightarrow g \sqsubseteq_{\mathcal{O}} g' \wedge g' \sqsubseteq_{\mathcal{O}} g$.

Una relación de equivalencia $=_{\mathcal{O}}$ divide el conjunto de todos los grafos $\mathbb{G}$ en conjuntos disjuntos o clases dentro de los cuales es imposible diferenciarlos, según la observación definida en $\mathcal{O}$, es decir, son grafos $\mathcal{O}$-equivalentes.

10.2 Notación utilizada

Dados dos conjuntos A y B, una relación R entre ellos se caracteriza por un subconjunto del producto cartesiano $A \times B$. Una relación homogénea en un conjunto A se caracterizará por un subconjunto del producto cartesiano $A \times A$. A lo largo de este capítulo trataremos siempre con relaciones homogéneas dentro del conjunto $\mathbb{G}$ de grafos MUS.

Definición 10.1. *Se define una* **vía de evolución finita** *de un grafo MUS $g \in \mathbb{G}$, denotándose π, a una secuencia finita alternada de estados y transiciones, que comienza y termina con un estado, tal que cada transición comienza en el estado anterior y termina en el siguiente:*

$$\pi = E_i(E_i, a_i, E_{i+1})E_{i+1}(E_{i+1}, a_{i+1}, E_{i+2})E_{i+2} \ldots (E_n, a_n, E_{n+1})E_{n+1}$$

$$\Updownarrow$$

$$\pi : E_i \xrightarrow{a_i} E_{i+1} \xrightarrow{a_{i+1}} \ldots \xrightarrow{a_n} E_{n+1}$$

Definición 10.2. *Se define una* **vía de evolución no acotada en longitud** *de un grafo MUS $g \in \mathbb{G}$, denotándose π, a una secuencia alternada de estados y transiciones tal que cada transición comienza en un estado y termina en el siguiente y donde se da la particularidad de que se define un bucle de comportamiento, es decir, la transición que parte de uno de los estados tiene como destino el mismo*

estado de comienzo de la vía:

$$\pi = E_i(E_i, a_i, E_{i+1})E_{i+1}(E_{i+1}, a_{i+1}, E_{i+2})E_{i+2}\ldots(E_n, a_n, E_i)E_i\ldots$$
$$\Updownarrow$$
$$\pi : E_i \xrightarrow{a_i} E_{i+1} \xrightarrow{a_{i+1}} \ldots \xrightarrow{a_n} E_i \xrightarrow{a_i} \ldots$$

Dado que cualquier componente de un grafo MUS puede estar subespecificado, es el caso, por ejemplo, de las acciones asociadas a las transiciones, sería necesario indicar de alguna forma cuál es el grado de especificación de dichos eventos en cada vía de evolución. Sin embargo, asumiremos que en el caso de tener una transición entre dos estados adyacentes, donde todavía no se haya especificado qué evento produce dicha evolución, la acción asociada a dicho estado será la acción reservada a_{sub}. De esta manera, será posible tratar este caso de forma análoga a aquellas vías donde todas las transiciones están totalmente especificadas. De la misma forma si una transición de un estado está especificada como no posible o *falso*, entonces se denotará como una transición a un estado de parada E_p ficticio.

Definición 10.3. *Dado un grafo MUS $g \in \mathbb{G}$ se denota como $V_f(g)$ al conjunto de todas las vías de evolución completas y finitas de g. Entendiendo por vías de evolución completas y finitas a aquellas que, no presentando ningún bucle de comportamiento —vías finitas—, tienen como estado de inicio el estado inicial del grafo E_0, y tienen como estado final un estado totalmente subespecificado o bien un estado ficticio de parada.*

Definición 10.4. *Dado un grafo MUS $g \in \mathbb{G}$ se denota como $V_i(g)$ al conjunto de todas las vías de evolución completas y no acotadas de g. Entendiendo por vías de evolución completas y no acotadas a aquellas que, presentando algún tipo de recursión —vías no acotadas—, tienen como estado de inicio el estado inicial del grafo E_0, y tienen como estado final un estado totalmente subespecificado, un estado ficticio de parada, o bien no es posible identificar un estado de finalización debido a que presentan como sufijo una vía no acotada.*

Definición 10.5. *Dado un grafo MUS $g \in \mathbb{G}$ se denota como $V(g)$ al conjunto de todas las vías de evolución completas del grafo, formado por la unión de $V_f(g)$ y $V_i(g)$.*

EJEMPLO 10.1. Un ejemplo de la obtención del conjunto de vías completas de un grafo $V(g)$ (figura 10.1) es el que se indica a continuación. En este caso, el

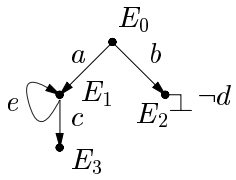


Figura 10.1. Ejemplo de grafo MUS y de obtención del conjunto $V(g)$

conjunto $V_f(g)$ estará formado por las siguientes vías:

$$\begin{aligned}
\pi_1 &: E_0 \xrightarrow{b} E_2 \xrightarrow{\neg d} E_p \\
\pi_2 &: E_0 \xrightarrow{a} E_1 \xrightarrow{c} E_3 \\
\pi_3 &: E_0 \xrightarrow{a} E_1 \xrightarrow{e} E_1 \xrightarrow{c} E_3
\end{aligned}$$

y el conjunto $V_i(g)$ tiene como componentes a las vías siguientes:

$$\begin{aligned}
\pi_4 &: E_0 \xrightarrow{a} E_1 \xrightarrow{e} E_1 \xrightarrow{e} \dots \\
\pi_5 &: E_0 \xrightarrow{a} E_1 \xrightarrow{e} E_1 \xrightarrow{e} \dots \xrightarrow{e} E_1 \xrightarrow{c} E_3
\end{aligned}$$

A partir de lo cual podemos obtener el conjunto $V(g)$ sin más que realizar la unión de los dos conjuntos anteriores. □

Definición 10.6. *Dado un grafo MUS $g \in \mathbb{G}$, se define la relación* **cardinal de** *g, denotándose $\sharp g$, como aquella que obtiene, para cada g su número de vías de evolución completas $V(g)$:*

$$\begin{array}{ccc}
\sharp : \mathbb{G} & \longmapsto & \mathbb{N} \\
g & \longmapsto & n = \sharp g
\end{array}$$

Definición 10.7. *Dados dos grafos $g, g' \in \mathbb{G}$ se dice que están relacionados según $\sim$, denotándose $g \sim g'$, si ambos tienen idéntico cardinal, es decir, $\sharp g = \sharp g'$.*

La relación definida en 10.7 satisface las propiedades transitiva, simétrica y reflexiva:

$$\begin{aligned}
&\forall g \in \mathbb{G} \Rightarrow g \sim g && \text{(Reflexiva)} \\
&g \sim g' \Rightarrow g' \sim g && \text{(Simétrica)} \\
&(g \sim g' \wedge g' \sim g'') \Rightarrow g \sim g'' && \text{(Transitiva)}
\end{aligned}$$

por lo que puede decirse que define una relación de equivalencia en el conjunto $\mathbb{G}$.

Así, utilizando esta relación de equivalencia, es posible definir una partición del conjunto $\mathbb{G}$ en diferentes subconjuntos ($\sim$-equivalentes) de la forma:

$$\mathbb{G}/\sim = \{G_1, G_2, \ldots\}$$

donde cada G_i denota al conjunto de todos los grafos $g \in \mathbb{G}$ satisfaciendo $\sharp g = i$. Debido a las propiedades de una partición, la unión de todos los subconjuntos G_i será igual a $\mathbb{G}$ y la intersección entre dos de ellos cualesquiera tendrá como resultado el conjunto vacío.

Definición 10.8. *Dada una vía de evolución* $\pi \in V_f(g)$

$$\pi : E_0 \xrightarrow{a_1} E_1 \xrightarrow{a_2} \ldots \xrightarrow{a_n} E_n$$

entonces $T(\pi) = a_1 a_2 \ldots a_n$ *es la* **traza finita** *de* π*. En el caso de encontrarnos ante una vía de evolución* $\pi \in V_i(g)$

$$\pi : E_0 \xrightarrow{a_1} \ldots \xrightarrow{a_i} E_i \xrightarrow{a_{i+1}} \ldots \xrightarrow{a_n} E_n \xrightarrow{a_{n+1}} E_i \xrightarrow{a_{i+1}} \ldots$$

se obtiene su **traza finita** *como aquella resultante de considerar las transiciones que provocan la recursión como si fuesen finitas, es decir,* $T(\pi) = a_1 a_2 \ldots a_n a_{n+1}$.

El conjunto de todas las trazas finitas posibles definidas a partir del alfabeto Λ —conjunto de acciones definidas para un grafo— se denotará $\mathbb{T}$, de donde es inmediato que $T(\pi) \in \mathbb{T},\ \forall \pi \in V(g)$.

Definición 10.9. *Dada una vía de evolución* $\pi \in V_i(g)$

$$\pi : E_0 \xrightarrow{a_1} \ldots \xrightarrow{a_i} E_i \xrightarrow{a_{i+1}} \ldots \xrightarrow{a_n} E_n \xrightarrow{a_{n+1}} E_i \xrightarrow{a_{i+1}} \ldots$$

entonces $T^\infty(\pi) = a_1 a_2 \ldots a_i (a_{i+1} \ldots a_{n+1})+$ *es la* **traza no acotada** *de* π[1]*. En el caso de encontrarnos ante una vía de evolución* $\pi \in V_f(g)$

$$\pi : E_0 \xrightarrow{a_1} E_1 \xrightarrow{a_2} \ldots \xrightarrow{a_n} E_n$$

es inmediato que $T^\infty(\pi) = T(\pi) = a_1 a_2 \ldots a_n$*, ya que, en esta situación,* π *no define ningún bucle de comportamiento.*

El conjunto de todas las trazas no acotadas posibles definidas a partir del alfabeto Λ —conjunto de acciones definidas para un grafo— se denotará $\mathbb{T}^\infty$, de donde es inmediato que $T^\infty(\pi) \in \mathbb{T}^\infty,\ \forall \pi \in V(g)$.

[1]Con la notación $(a_{i+1} \ldots a_{n+1})+$ se refleja la posibilidad de que el sistema ejecute un número no determinado de veces la secuencia de eventos $a_{i+1} \ldots a_{n+1}$.

Definición 10.10. *Dada una vía de evolución* $\pi \in V_f(g)$

$$\pi : E_0 \xrightarrow{a_1} E_1 \xrightarrow{a_2} \ldots \xrightarrow{a_n} E_n$$

entonces $N(\pi) = n$ *es el* **número de evoluciones finitas de** π. $N(\pi)$ *denota el número de transiciones que recorre el sistema cuando evoluciona por la vía* π. *En el caso de encontrarnos ante una vía de evolución* $\pi \in V_i(g)$

$$\pi : E_0 \xrightarrow{a_1} \ldots \xrightarrow{a_i} E_i \xrightarrow{a_{i+1}} \ldots \xrightarrow{a_n} E_n \xrightarrow{a_{n+1}} E_i \xrightarrow{a_{i+1}} \ldots$$

$N(\pi)$ *se obtendrá entonces considerando aquellas transiciones que provocan un bucle de comportamiento como si fuesen transiciones finitas, es decir, en este caso* $N(\pi) = n + 1$.

El conjunto de todos los números de evolución finitos posibles definidos sobre un grafo, se denotará $\mathbb{N}$, de donde es inmediato que $N(\pi) \in \mathbb{N},\ \forall \pi \in V(g)$.

Definición 10.11. *Dada una vía de evolución* $\pi \in V_i(g)$

$$\pi : E_0 \xrightarrow{a_1} \ldots \xrightarrow{a_i} E_i \xrightarrow{a_{i+1}} \ldots \xrightarrow{a_n} E_n \xrightarrow{a_{n+1}} E_i \xrightarrow{a_{i+1}} \ldots$$

entonces $N^{\infty}(\pi) = (n+1)+$ *es el* **número de evoluciones no acotadas de** π. $N^{\infty}(\pi)$ *denota el número de transiciones que recorre el sistema cuando evoluciona por la vía no acotada* π [2]. *En el caso de encontrarnos ante una vía de evolución* $\pi \in V_f(g)$

$$\pi : E_0 \xrightarrow{a_1} E_1 \xrightarrow{a_2} \ldots \xrightarrow{a_n} E_n$$

entonces es inmediato que $N^{\infty}(\pi) = N(\pi) = n$, *ya que, en esta situación,* π *no define ningún bucle de comportamiento.*

El conjunto de todos los números de evolución no acotados posibles definidos sobre un grafo, se denotará $\mathbb{N}^{\infty}$, de donde es inmediato que $N^{\infty}(\pi) \in \mathbb{N}^{\infty},\ \forall \pi \in V(g)$.

10.3 Relación de equivalencia de número de evoluciones

Definición 10.12. *Dado un grafo MUS* $g \in \mathbb{G}$ *y, más concretamente, perteneciente a una partición* G_i *—definida por la relación de equivalencia* $\sim$ *(definición 10.7)— se define la relación* **número de evoluciones** *de* g, *denotándose* $NE(g)$, *como aquella que evalúa sobre el grafo el número de evoluciones finitas de cada una de las posibles vías de evolución completas de* g:

[2] Con la notación $(n+1)+$ se refleja la posibilidad de que el sistema repita un número no acotado de veces una secuencia de $n+1$ transiciones.

$$\begin{array}{rcl} NE : G_i & \longmapsto & \mathbb{N}^i \\ g & \longmapsto & NE(g) = (N(\pi_1), N(\pi_2), \ldots, N(\pi_i)) \end{array}$$

donde $\pi_j \in V(g) \quad \forall\, j = 1, \ldots, i.$

De la definición anterior puede deducirse que $NE(g)$ no es más que un vector de números naturales cuya dimensión viene determinada por el número de vías de evolución completas de g, y donde cada componente especifica el número de transiciones de las que consta una de las posibles vías de evolución[3].

Definición 10.13. *Dados los vectores* número de evoluciones *de dos grafos* $NE(g)$ *y* $NE(g')$ *se dice que son idénticos,* $NE(g) = NE(g')$*, si se satisface que* $\sharp g = \sharp g'$ *y, además, se cumple que*

$$\forall\, \pi \in V(g),\ \exists\, \pi' \in V(g') \mid N(\pi) = N(\pi')$$

Esta relación que se establece entre los elementos de $NE(g)$ *y* $NE(g')$ *es biunívoca, es decir, una vez que un elemento* $N(\pi') \in NE(g')$ *ha sido asociado como equivalente a un elemento* $N(\pi) \in NE(g)$*, éste no podrá ser asociado como equivalente a ningún otro elemento* $N(\pi'') \in NE(g)$*. Es decir, será necesario que tengan las componentes de los vectores* número de evoluciones *idénticas dos a dos, aunque no sea relevante su orden.*

Definición 10.14. *Dados dos grafos* $g, g' \in \mathbb{G}$ *se dice que están relacionados según* $=_{\mathrm{NE}}$*, denotándose* $g =_{\mathrm{NE}} g'$ *si su número de evoluciones es idéntico, es decir,* $NE(g) = NE(g')$*.*
Esta relación $=_{\mathrm{NE}}$ *satisface las propiedades transitiva, simétrica y reflexiva, por lo que puede decirse que define una relación de equivalencia en el conjunto* $\mathbb{G}$*.*

La relación NE define un criterio de observación de grafos MUS que nos permite establecer una relación de equivalencia entre ellos, pudiendo fraccionar el conjunto de todos los grafos MUS en subconjuntos NE-equivalentes.

Definición 10.15. *Dados dos valores* $N(\pi)$ *y* $N(\pi')$ *se dice que el primero es menor o igual que el segundo, denotándose* $N(\pi) \leq N(\pi')$*, si el valor* $N(\pi)$ *es menor o igual que valor* $N(\pi')$*, aplicando la relación de orden típica entre números naturales.*

Definición 10.16. *Dado un valor* $N(\pi)$ *y un vector* $NE(g)$ *se define el supremo de* $N(\pi)$ *en el vector* $NE(g)$*, denotándose,* $\sup(N(\pi), NE(g))$*, a aquel valor* $N(\pi') \in NE(g)$ *tal que* $N(\pi) \leq N(\pi')$*, satisfaciendo además que* $\nexists\, N(\pi'') \in NE(g) \mid N(\pi) \leq N(\pi'') \leq N(\pi')$*.*

[3] En el apartado A.5 se detalla el pseudocódigo del algoritmo utilizado para extraer esta información a partir de la aplicación TC (ver apartado 10.5).

Definición 10.17. *Dados los vectores* número de evoluciones *de dos grafos* $NE(g)$ *y* $NE(g')$ *se dice que el primero está incluido en el segundo,* $NE(g) \subseteq NE(g')$*, si se satisface que* $\sharp g \leq \sharp g'$*, se cumple que*

$$\forall\, \pi \in V(g),\ \exists\, \pi' \in V(g') \mid \sup(N(\pi), NE(g')) = N(\pi')$$

y, además, esta relación que se establece entre los elementos de $NE(g)$ *y* $NE(g')$ *es biunívoca, es decir, una vez que un elemento* $N(\pi') \in NE(g')$ *ha sido asociado como supremo a un elemento* $N(\pi) \in NE(g)$*, éste no podrá ser asociado como supremo a ningún otro elemento* $N(\pi'') \in NE(g)$*.*

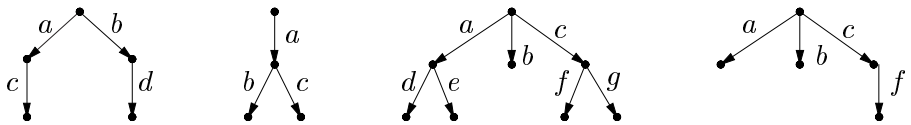
Definición 10.18. *Un grafo MUS* g *está* NE*-incluido en otro grafo* g'*, denotándose* $g \sqsubseteq_{\mathrm{NE}} g'$*, si y sólo si se satisface que* $NE(g) \subseteq NE(g')$*.*

Esta relación satisface las propiedades de una relación de orden:

$$\forall g \in \mathbb{G} \Rightarrow g \sqsubseteq_{\mathrm{NE}} g \qquad \text{(Reflexiva)}$$
$$(g \sqsubseteq_{\mathrm{NE}} g' \wedge g' \sqsubseteq_{\mathrm{NE}} g) \Rightarrow g =_{\mathrm{NE}} g' \qquad \text{(Antisimétrica)}$$
$$(g \sqsubseteq_{\mathrm{NE}} g' \wedge g' \sqsubseteq_{\mathrm{NE}} g'') \Rightarrow g \sqsubseteq_{\mathrm{NE}} g'' \qquad \text{(Transitiva)}$$

y, dado que no todos los grafos de $\mathbb{G}$ pueden ser comparados atendiendo a esta relación, se dice que $(\mathbb{G}, \sqsubseteq_{\mathrm{NE}})$ es un conjunto parcialmente ordenado.

EJEMPLO 10.2. En la figura 10.2 puede verse claramente que tanto el grafo g_1 como el g_2 son NE-equivalentes, ya que el conjunto de sus números de evolución vienen dados, respectivamente, por $NE(g_1) = (2, 2)$ y $NE(g_2) = (2, 2)$. Sin embargo, ninguno de ellos es NE-equivalente con g_3, ya que $NE(g_3) = (2, 2, 2, 2, 1)$. Lo que sí se satisface es que $g_1 \sqsubseteq_{\mathrm{NE}} g_3$ y $g_2 \sqsubseteq_{\mathrm{NE}} g_3$. Por otro lado, sólo g_3 mantiene una NE-relación con g_4 ($g_4 \sqsubseteq_{\mathrm{NE}} g_3$) ya que se satisface que $g_i \not\sqsubseteq_{\mathrm{NE}} g_4 \wedge g_4 \not\sqsubseteq_{\mathrm{NE}} g_i,\ \forall i = 1, 2$. □



(a) g_1 (b) g_2 (c) g_3 (d) g_4

Figura 10.2. Diferentes grafos MUS y sus relaciones de NE-equivalencia

10.4 Relación de equivalencia de número de evoluciones no acotadas

En el apartado 10.3 no se han tenido en cuenta las vías de evolución no acotadas, propias de aquellos grafos que contienen bucles o recursiones. Para este caso, es necesario definir una nueva relación que llamaremos **número de evoluciones no acotadas**:

Definición 10.19. *Dado un grafo MUS $g \in \mathbb{G}$ y, más concretamente, perteneciente a una partición G_i —definida por la relación de equivalencia $\sim$ (definición 10.7)— se define la relación* **número de evoluciones no acotadas** *de g, denotándose $NE^{\infty}(g)$, como aquella que evalúa sobre el grafo el número de evoluciones no acotadas de cada una de las posibles vías de evolución completas de g:*

$$\begin{array}{ccc} NE^{\infty} : G_i & \longmapsto & \mathbb{N}^{\infty\, i} \\ g & \longmapsto & NE^{\infty}(g) = (N^{\infty}(\pi_1), N^{\infty}(\pi_2), \ldots, N^{\infty}(\pi_i)) \end{array}$$

donde $\pi_j \in V(g) \quad \forall\, j = 1, \ldots, i.$

De la definición anterior puede concluirse que $NE^{\infty}(g)$ no es más que un vector de elementos de $\mathbb{N}^{\infty}$, cuya dimensión viene determinada por el número de vías de evolución completas posibles de g, y donde cada componente determina el número de transiciones de cada vía.

Definición 10.20. *Dados los vectores* número de evoluciones no acotadas *de dos grafos $NE^{\infty}(g)$ y $NE^{\infty}(g')$ se dice que son idénticos, $NE^{\infty}(g) = NE^{\infty}(g')$, si se satisface que $\sharp g = \sharp g'$ y, además, se cumple que*

$$\forall\, \pi \in V(g),\ \exists\, \pi' \in V(g') \mid N^{\infty}(\pi) = N^{\infty}(\pi')$$

Esta relación que se establece entre los elementos de $NE^{\infty}(g)$ y $NE^{\infty}(g')$ es biunívoca, es decir, una vez que un elemento $N^{\infty}(\pi') \in NE^{\infty}(g')$ ha sido asociado como equivalente a un elemento $N^{\infty}(\pi) \in NE^{\infty}(g)$, éste no podrá ser asociado como equivalente a ningún otro elemento $N^{\infty}(\pi'') \in NE^{\infty}(g)$. Es decir, será necesario que tengan las componentes de los vectores número de evoluciones no acotadas *idénticas dos a dos, aunque no sea relevante su orden.*

Definición 10.21. *Dados dos grafos $g, g' \in \mathbb{G}$ se dice que están relacionados según $=^{\infty}_{\mathrm{NE}}$, denotándose $g =^{\infty}_{\mathrm{NE}} g'$, si su número de evoluciones no acotadas es idéntico, es decir, $NE^{\infty}(g) = NE^{\infty}(g')$.*
Esta relación $=^{\infty}_{\mathrm{NE}}$ satisface las propiedades transitiva, simétrica y reflexiva, por lo que puede decirse que define una relación de equivalencia en el conjunto $\mathbb{G}$.

Esta relación amplía la definida en 10.14 contemplando también los bucles especificados en el grafo.[4]

Definición 10.22. *Dados dos valores $N^{\infty}(\pi)$ y $N^{\infty}(\pi')$ se dice que el primero es menor o igual que el segundo, denotándose $N^{\infty}(\pi) \leq N^{\infty}(\pi')$ si:*

- *siendo $N^{\infty}(\pi)$ y $N^{\infty}(\pi')$ números naturales, el primero es menor o igual que el segundo, o bien*
- *siendo $N^{\infty}(\pi)$ un número natural y $N^{\infty}(\pi')$ un natural con recursión — notación $()+$— el valor natural del primero es menor o igual que el número natural sin recursión del segundo, o bien*
- *siendo $N^{\infty}(\pi)$ y $N^{\infty}(\pi')$ ambos valores naturales con recursión, el número natural sin recursión del primero es menor o igual que el número natural del segundo, también sin recursión.*

Definición 10.23. *Dado un valor $N^{\infty}(\pi)$ y un vector $NE^{\infty}(g)$ se define el supremo de $N^{\infty}(\pi)$ en $NE^{\infty}(g)$, denotándose $\sup(N^{\infty}(\pi), NE^{\infty}(g))$, a aquel valor $N^{\infty}(\pi') \in NE^{\infty}(g)$ tal que $N^{\infty}(\pi) \leq N^{\infty}(\pi')$, satisfaciendo además que $\nexists\, N^{\infty}(\pi'') \in NE^{\infty}(g) \mid N^{\infty}(\pi) \leq N^{\infty}(\pi'') \leq N^{\infty}(\pi')$.*

Definición 10.24. *Dados los vectores* número de evoluciones no acotadas *de dos grafos $NE^{\infty}(g)$ y $NE^{\infty}(g')$ se dice que el primero está incluido en el segundo, $NE^{\infty}(g) \subseteq NE^{\infty}(g')$, si se satisface que $\sharp g \leq \sharp g'$, se cumple que*

$$\forall\, \pi \in V(g),\ \exists\, \pi' \in V(g') \mid \sup(N^{\infty}(\pi), NE^{\infty}(g)) = N^{\infty}(\pi')$$

y, además, esta relación que se establece entre los elementos de $NE^{\infty}(g)$ y $NE^{\infty}(g')$ es biunívoca, es decir, una vez que se ha asociado un elemento $N^{\infty}(\pi') \in NE^{\infty}(g')$ como supremo a un elemento $N^{\infty}(\pi) \in NE^{\infty}(g)$, éste no podrá ser asociado como supremo a ningún otro elemento $N^{\infty}(\pi'') \in NE^{\infty}(g)$.

Definición 10.25. *Un grafo MUS g está NE^{∞}-incluido en g', denotándose $g \sqsubseteq^{\infty}_{\mathrm{NE}} g'$, si y sólo si $NE^{\infty}(g) \subseteq NE^{\infty}(g')$.*

Al igual que lo hacía la relación de inclusión 10.18, ésta también satisface las propiedades de una relación de orden:

$$\forall g \in \mathbb{G} \Rightarrow g \sqsubseteq^{\infty}_{\mathrm{NE}} g \qquad \text{(Reflexiva)}$$
$$(g \sqsubseteq^{\infty}_{\mathrm{NE}} g' \wedge g' \sqsubseteq^{\infty}_{\mathrm{NE}} g) \Rightarrow g =^{\infty}_{\mathrm{NE}} g' \qquad \text{(Antisimétrica)}$$
$$(g \sqsubseteq^{\infty}_{\mathrm{NE}} g' \wedge g' \sqsubseteq^{\infty}_{\mathrm{NE}} g'') \Rightarrow g \sqsubseteq^{\infty}_{\mathrm{NE}} g'' \qquad \text{(Transitiva)}$$

[4] En el apartado A.3 se detalla el pseudocódigo del algoritmo utilizado para obtener la información dada por NE^{∞} a partir de la información obtenida por la relación TC^{∞} (ver apartado 10.6).

y, dado que no todos los grafos de $\mathbb{G}$ pueden ser comparados según esta relación, se puede decir que $(\mathbb{G}, \sqsubseteq^{\infty}_{\mathrm{NE}})$ es un conjunto parcialmente ordenado.

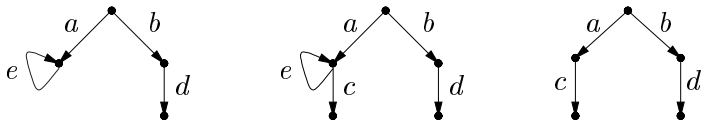
EJEMPLO 10.3. En la figura 10.3 puede verse que las relaciones NE y NE^{∞} para cada grafo son como sigue:

$$\begin{aligned} NE(g_5) &= (2,2) & NE^{\infty}(g_5) &= (2+,2) \\ NE(g_6) &= (2,2,2,3,3) & NE^{\infty}(g_6) &= (2,2,3,2+,3+) \\ NE(g_1) &= (2,2) & NE^{\infty}(g_1) &= (2,2) \end{aligned}$$

De lo que se deduce que las relaciones entre estos grafos son como se indican a continuación:

$$g_5 =_{\mathrm{NE}} g_1 \qquad g_5 \sqsubseteq_{\mathrm{NE}} g_6 \qquad g_1 \sqsubseteq_{\mathrm{NE}} g_6$$
$$g_5 \neq^{\infty}_{\mathrm{NE}} g_6 \neq^{\infty}_{\mathrm{NE}} g_1 \qquad g_1 \sqsubseteq^{\infty}_{\mathrm{NE}} g_5 \sqsubseteq^{\infty}_{\mathrm{NE}} g_6$$

□



(a) g_5 (b) g_6 (c) g_1

Figura 10.3. Diferentes grafos MUS y sus relaciones de NE^{∞}-equivalencia

10.5 Relación de equivalencia de trazas completas

Definición 10.26. *Dado un grafo MUS $g \in \mathbb{G}$ y, más concretamente, perteneciente a una partición G_i —definida por la relación de equivalencia $\sim$ (definición 10.7)— se define la relación* **trazas completas** *de g, denotándose $TC(g)$, como aquella que evalúa sobre el grafo:*

$$\begin{aligned} TC : G_i &\longmapsto \mathbb{T}^i \\ g &\longmapsto TC(g) = (T(\pi_1), T(\pi_2), \ldots, T(\pi_i)) \end{aligned}$$

donde $\pi_j \in V(g) \quad \forall\, j = 1, \ldots, i.$

$TC(g)$ define un vector cuya dimensión viene determinada por el número de vías de evolución completas de g, y donde cada componente es cada una de las trazas finitas de cada vía[5].

Definición 10.27. *Dados los vectores* trazas completas *de dos grafos $TC(g)$ y $TC(g')$ se dice que son idénticos, $TC(g) = TC(g')$, si se satisface que $\sharp g = \sharp g'$ y, además, se cumple que*

$$\forall\, \pi \in V(g),\ \exists\, \pi' \in V(g') \mid T(\pi) = T(\pi')$$

Esta relación que se establece entre los elementos de $TC(g)$ y $TC(g')$ es biunívoca, es decir, una vez que un elemento $T(\pi') \in TC(g')$ ha sido asociado como equivalente a un elemento $T(\pi) \in TC(g)$, éste no podrá ser asociado como equivalente a ningún otro elemento $T(\pi'') \in TC(g)$. Es decir, será necesario que tengan las componentes de los vectores trazas completas *idénticas dos a dos, aunque no sea relevante su orden.*

Definición 10.28. *Dados dos grafos $g, g' \in \mathbb{G}$ se dice que están relacionados según $=_{\mathrm{TC}}$, denotándose $g =_{\mathrm{TC}} g'$, si sus trazas completas son idénticas, es decir, $TC(g) = TC(g')$.*
Esta relación $=_{\mathrm{TC}}$ satisface las propiedades transitiva, simétrica y reflexiva, por lo que puede decirse que define una relación de equivalencia en el conjunto $\mathbb{G}$.

Basándonos en esta misma relación, es posible definir una relación de inclusión o contenido que permitirá una ordenación parcial del conjunto de grafos $\mathbb{G}$ atendiendo al criterio de observación definido por TC.

Definición 10.29. *Dadas dos trazas completas, $T(\pi)$ y $T(\pi')$, se dice que la primera está incluida en la segunda, denotándose $T(\pi) \subseteq T(\pi')$, si se satisface que $T(\pi)$ es una traza prefijo de $T(\pi')$.*

Definición 10.30. *Dado un valor $T(\pi)$ y un vector $TC(g)$ se define el supremo de $T(\pi)$ en $TC(g)$, denotándose $\sup(T(\pi), TC(g))$, a aquel valor $T(\pi') \in TC(g)$ tal que $T(\pi) \subseteq T(\pi')$, satisfaciendo además que $\nexists\, T(\pi'') \in TC(g) \mid T(\pi) \subseteq T(\pi'') \subseteq T(\pi')$.*

Definición 10.31. *Dados los vectores* trazas completas *de dos grafos $TC(g)$ y $TC(g')$ se dice que el primero está incluido en el segundo, $TC(g) \subseteq TC(g')$, si se satisface que $\sharp g \leq \sharp g'$, se cumple que*

$$\forall\, \pi \in V(g),\ \exists\, \pi' \in V(g') \mid\ \sup(T(\pi), TC(g)) = T(\pi')$$

[5] En el apartado A.4 se detalla el pseudocódigo del algoritmo utilizado para extraer la información suministrada por TC a partir de la información obtenida de la relación TC^{∞} (ver apartado 10.6).

y, además, esta relación que se establece entre los elementos de $TC(g)$ *y* $TC(g')$ *es biunívoca, es decir, una vez que se ha asociado un elemento* $T(\pi') \in TC(g')$ *como supremo a un elemento* $T(\pi) \in TC(g)$*, éste no podrá ser asociado como supremo a ningún otro elemento* $T(\pi'') \in TC(g)$*.*

Definición 10.32. *Un grafo MUS* g *es* TC*-menor que otro grafo* g'*, denotándose* $g \sqsubseteq_{\mathrm{TC}} g'$ *si y sólo si* $TC(g) \subseteq TC(g')$*.*

Esta relación satisface las propiedades de una relación de orden:

$$\forall g \in \mathbb{G} \Rightarrow g \sqsubseteq_{\mathrm{TC}} g \qquad \text{(Reflexiva)}$$
$$(g \sqsubseteq_{\mathrm{TC}} g' \wedge g' \sqsubseteq_{\mathrm{TC}} g) \Rightarrow g =_{\mathrm{TC}} g' \qquad \text{(Antisimétrica)}$$
$$(g \sqsubseteq_{\mathrm{TC}} g' \wedge g' \sqsubseteq_{\mathrm{TC}} g'') \Rightarrow g \sqsubseteq_{\mathrm{TC}} g'' \qquad \text{(Transitiva)}$$

y, dado que no todos los grafos de $\mathbb{G}$ pueden ser comparados atendiendo a esta relación, se dice que $(\mathbb{G}, \sqsubseteq_{\mathrm{TC}})$ es un conjunto parcialmente ordenado.

EJEMPLO 10.4. En la figura 10.2 puede verse fácilmente que ninguno de los grafos es TC-equivalente, ya que $TC(g_1) = (ac, bd)$, $TC(g_2) = (ab, ac)$, $TC(g_3) = (ad, ae, b, cf, cg)$ y $TC(g_4) = (a, b, cf)$. Lo que sí se puede asegurar es que sólo g_3 y g_4 mantienen una relación de orden parcial según TC, es decir, $g_4 \sqsubseteq_{\mathrm{TC}} g_3$.

□

10.6 Relación de equivalencia de trazas completas no acotadas

En el apartado 10.5 no se han tenido en cuenta aquellos grafos que contienen bucles de evolución, para este caso es necesario definir una nueva relación que llamaremos **trazas completas no acotadas**:

Definición 10.33. *Dado un grafo MUS* $g \in \mathbb{G}$ *y, más concretamente, perteneciente a una partición* G_i *—definida por la relación de equivalencia* $\sim$ *(definición 10.7)— se define la relación* **trazas completas no acotadas** *de* g*, denotándose* $TC^{\infty}(g)$*, como aquella que evalúa sobre el grafo:*

$$\begin{array}{rcl} TC^{\infty} : G_i & \longmapsto & \mathbb{T}^{\infty\, i} \\ g & \longmapsto & TC^{\infty}(g) = (T^{\infty}(\pi_1), T^{\infty}(\pi_2), \ldots, T^{\infty}(\pi_i)) \end{array}$$

donde $\pi_j \in V(g) \quad \forall\, j = 1, \ldots, i.$

$TC^{\infty}(g)$ representa un vector cuya dimensión viene determinada por el número de vías de evolución completas de g, y donde cada componente es la traza no acotada de cada una de dichas vías[6]

Definición 10.34. *Dados los vectores $TC^{\infty}(g)$ y $TC^{\infty}(g')$ de dos grafos se dice que son idénticos, $TC^{\infty}(g) = TC^{\infty}(g')$, si se satisface que $\sharp g = \sharp g'$ y, además, se cumple que*

$$\forall\, \pi \in V(g),\ \exists\, \pi' \in V(g') \mid T^{\infty}(\pi) = T^{\infty}(\pi')$$

Esta relación que se establece entre los elementos de $TC^{\infty}(g)$ y $TC^{\infty}(g')$ es biunívoca, es decir, una vez que un elemento $T^{\infty}(\pi') \in TC^{\infty}(g')$ ha sido asociado como equivalente a un elemento $T^{\infty}(\pi) \in TC^{\infty}(g)$, éste no podrá ser asociado como equivalente a ningún otro elemento $T^{\infty}(\pi'') \in TC^{\infty}(g)$. Es decir, será necesario que tengan las componentes de los vectores trazas completas no acotadas *idénticas dos a dos, aunque no sea relevante su orden.*

Definición 10.35. *Dados dos grafos $g, g' \in \mathbb{G}$ se dice que están relacionados según $=^{\infty}_{\mathrm{TC}}$, denotándose $g =^{\infty}_{\mathrm{TC}} g'$, si sus trazas completas no acotadas son idénticas, es decir, $TC^{\infty}(g) = TC^{\infty}(g')$.*
Esta relación $=^{\infty}_{\mathrm{TC}}$ satisface las propiedades transitiva, simétrica y reflexiva, por lo que puede decirse que define una relación de equivalencia en el conjunto $\mathbb{G}$.

La definición de una relación de orden atendiendo a la relación definida en 10.33 es inmediata.

Definición 10.36. *Dados dos valores $T^{\infty}(\pi)$ y $T^{\infty}(\pi')$ se dice que el primero está contenido en el segundo, denotándose $T^{\infty}(\pi) \subseteq T^{\infty}(\pi')$, si se satisface que:*

- *siendo ambos valores trazas finitas, $T^{\infty}(\pi)$ es prefijo de $T^{\infty}(\pi')$, o bien*
- *siendo $T^{\infty}(\pi)$ una traza finita y $T^{\infty}(\pi')$ una traza no acotada, $T^{\infty}(\pi)$ es prefijo de $T^{\infty}(\pi')$.*

Definición 10.37. *Dado un valor $T^{\infty}(\pi)$ y un vector $TC^{\infty}(g)$ se define el supremo de $T^{\infty}(\pi)$ en el vector $TC^{\infty}(g)$, denotándose $\sup(T^{\infty}(\pi), TC^{\infty}(g))$, a aquel valor $T^{\infty}(\pi') \in TC^{\infty}(g)$ tal que $T^{\infty}(\pi) \subseteq T^{\infty}(\pi')$, satisfaciendo además que $\nexists\, T^{\infty}(\pi'') \in TC(g) \mid T^{\infty}(\pi) \subseteq T^{\infty}(\pi'') \subseteq T^{\infty}(\pi')$.*

Definición 10.38. *Dados los vectores $TC^{\infty}(g)$ y $TC^{\infty}(g')$ de dos grafos se dice que el primero está incluido en el segundo, $TC^{\infty}(g) \subseteq TC^{\infty}(g')$, si se satisface*

[6] En el apartado A.2 se detalla el pseudocódigo del algoritmo recursivo que permite extraer la información suministrada por la relación TC^{∞} de un grafo MUS.

que $\sharp g \leq \sharp g'$*, se cumple que*

$$\forall\, \pi \in V(g),\ \exists\, \pi' \in V(g') \mid \sup(T^{\infty}(\pi), TC(g)) = T^{\infty}(\pi')$$

y, además, esta relación que se establece entre los elementos de $TC^{\infty}(g)$ *y* $TC^{\infty}(g')$ *es biunívoca, es decir, una vez que se haya asociado un elemento* $T^{\infty}(\pi') \in TC^{\infty}(g')$ *como supremo a un elemento* $T^{\infty}(\pi) \in TC^{\infty}(g)$*, éste no podrá ser asociado como supremo a ningún otro* $T^{\infty}(\pi'') \in TC(g)$*.*

Definición 10.39. *Un grafo MUS* g *es* TC^{∞}*-menor que otro grafo* g'*, denotándose* $g \sqsubseteq^{\infty}_{\mathrm{TC}} g'$*, si y sólo si se satisface que* $TC^{\infty}(g) \subseteq TC^{\infty}(g')$*.*

Al igual que el resto de las relaciones de inclusión definidas, ésta también satisface las propiedades de una relación de orden:

$$\begin{aligned}
&\forall g \in \mathbb{G} \Rightarrow g \sqsubseteq^{\infty}_{\mathrm{TC}} g && \text{(Reflexiva)}\\
&(g \sqsubseteq^{\infty}_{\mathrm{TC}} g' \wedge g' \sqsubseteq^{\infty}_{\mathrm{TC}} g) \Rightarrow g =^{\infty}_{\mathrm{TC}} g' && \text{(Antisimétrica)}\\
&(g \sqsubseteq^{\infty}_{\mathrm{TC}} g' \wedge g' \sqsubseteq^{\infty}_{\mathrm{TC}} g'') \Rightarrow g \sqsubseteq^{\infty}_{\mathrm{TC}} g'' && \text{(Transitiva)}
\end{aligned}$$

y, dado que no todos los grafos de $\mathbb{G}$ pueden ser comparados atendiendo a esta relación, se dice que $(\mathbb{G}, \sqsubseteq^{\infty}_{\mathrm{TC}})$ es un conjunto parcialmente ordenado.

EJEMPLO 10.5. En el ejemplo de la figura 10.3 se tiene que los conjuntos de trazas completas, tanto finitas como no acotadas, son los siguientes:

$$\begin{aligned}
&TC(g_5) = (ae, bd) && TC^{\infty}(g_5) = (ae+, bd)\\
&TC(g_6) = (ac, ae, bd, aec, aec) && TC^{\infty}(g_6) = (ac, ae + c, ae+, aec, bd)\\
&TC(g_1) = (ac, bd) && TC^{\infty}(g_1) = (ac, bd)
\end{aligned}$$

De lo que se deduce que las relaciones, en cuanto a trazas completas finitas y no acotadas, entre los grafos de la figura serán las siguientes:

$$\begin{aligned}
&g_5 \sqsubseteq_{\mathrm{TC}} g_6 && g_1 \sqsubseteq_{\mathrm{TC}} g_6\\
&g_5 \sqsubseteq^{\infty}_{\mathrm{TC}} g_6 && g_1 \sqsubseteq^{\infty}_{\mathrm{TC}} g_6
\end{aligned}$$

□

10.7 Relaciones de orden parcial entre las relaciones de equivalencia

Una vez definidas cuatro relaciones entre grafos MUS —relación de número de evoluciones, relación de número de evoluciones no acotadas, relación de trazas

completas y relación de trazas completas no acotadas— es inmediato extrapolar estas relaciones mantenidas por grafos MUS hacia los componentes que los incluyen, es decir, dos componentes C y C' mantienen una relación de equivalencia, $=_{\mathcal{O}}$, o bien de inclusión, $\sqsubseteq_{\mathcal{O}}$, denotándose $C =_{\mathcal{O}} C'$ y $C \sqsubseteq_{\mathcal{O}} C'$ respectivamente, si y sólo si sus correspondientes grafos MUS mantienen dicha relación, es decir, si $g_C =_{\mathcal{O}} g_{C'}$ o bien $g_C \sqsubseteq_{\mathcal{O}} g_{C'}$.

Dadas dos relaciones de equivalencia $\mathcal{O}$ y $\mathcal{N}$ se dice que la relación $\mathcal{N}$ contiene a la relación $\mathcal{O}$, denotándose $\mathcal{O} \preceq^*_{\mathbb{G}} \mathcal{N}$, si se satisface que $g \sqsubseteq_{\mathcal{N}} g' \Rightarrow g \sqsubseteq_{\mathcal{O}} g'$, $\forall g, g' \in \mathbb{G}$.

Sea Γ el conjunto de las relaciones de equivalencia definidas sobre los elementos de $\mathbb{G}$, es decir, $\Gamma = \{NE, NE^\infty, TC, TC^\infty\}$, la relación $\preceq^*_{\mathbb{G}}$ define un orden parcial en Γ, es decir, $(\Gamma, \preceq^*_{\mathbb{G}})$ define un conjunto parcialmente ordenado, es más, se puede decir que $(\Gamma, \preceq^*_{\mathbb{G}})$ define un **retículo** ya que $\forall \gamma, \gamma' \in \Gamma,\ \exists\ \sup\{\gamma, \gamma'\}\ e\ \inf\{\gamma, \gamma'\}$. El retículo puede verse gráficamente en la figura 10.4.

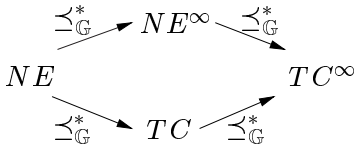


Figura 10.4. Retículo formado por las diferentes relaciones de equivalencia identificadas

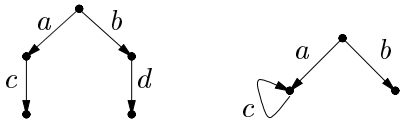
Es inmediato, definiciones 10.12 y 10.19, demostrar que $NE \preceq^*_{\mathbb{G}} NE^\infty$ y que $TC \preceq^*_{\mathbb{G}} TC^\infty$ (ver definiciones 10.26 y 10.33). Las relaciones existentes entre los criterios NE^∞ y TC^∞ —$NE^\infty \preceq^*_{\mathbb{G}} TC^\infty$— y entre los criterios NE y TC —$NE \preceq^*_{\mathbb{G}} TC$— son también directas. Por último, puede demostrarse que NE^∞ y TC no tienen ningún tipo de relación de contenido entre ellas: $TC \not\preceq^*_{\mathbb{G}} NE^\infty$ por definición y $NE^\infty \not\preceq^*_{\mathbb{G}} TC$ es fácil de ver con el contraejemplo de las figuras 10.5(a) y 10.5(b). En ellas, los grafos almacenan la siguiente información:

$$NE^\infty(g_1) = (2, 2) \qquad TC(g_1) = (ac, bd)$$
$$NE^\infty(g_2) = (2+, 1) \qquad TC(g_2) = (ac, b)$$

Así que $g_2 \sqsubseteq_{\mathrm{TC}} g_1$ y, sin embargo, $g_2 \not\sqsubseteq^\infty_{\mathrm{NE}} g_1$, por lo que se deduce que $NE^\infty \not\preceq^*_{\mathbb{G}} TC$.

10.8 Cadenas de componentes

En este apartado se pretende definir una serie de conceptos que son comunes para todas y cada una de las relaciones de orden parcial definidas a lo largo del



(a) g_1 (b) g_2

Figura 10.5. Contraejemplo para demostrar que no hay relación entre NE^{∞} y TC

presente capítulo. Para que las definiciones no resulten tediosas y repetitivas se van a realizar partiendo de una relación de orden genérica $\mathcal{O}$ y ésta puede ser substituida en cualquier caso por las relaciones identificadas en apartados anteriores.

Definición 10.40. *Dada una relación de orden parcial $\mathcal{O}$, se dice que dos componentes C y C' son* **comparables según** *$\mathcal{O}$ sii es posible identificar una relación de $\mathcal{O}$-precedencia entre ellos, es decir, $C \sqsubseteq_{\mathcal{O}} C'$ o bien $C' \sqsubseteq_{\mathcal{O}} C$.*

Definición 10.41. *Recibe el nombre de* **cadena de componentes***, $\mathcal{C}\mathrm{h}$, cualquier conjunto de componentes comparables según una relación de orden $\mathcal{O}$, denotándose $(\mathcal{C}\mathrm{h}, \sqsubseteq_{\mathcal{O}})$:*

$$\mathcal{C}\mathrm{h} = \{C_0 \sqsubseteq_{\mathcal{O}} C_1 \sqsubseteq_{\mathcal{O}} \ldots \sqsubseteq_{\mathcal{O}} C_k\}$$

Realizando la analogía típica con una estantería llena de libros, se podría decir que cada cadena es un estante y cada libro un componente, de forma que todos los *libros* relacionados se pueden localizar en el mismo *estante*. En el supuesto caso de tener un *libro* no relacionado con ninguno de los otros, se destinará a un *estante* donde permanecerá aislado hasta que lleguen otros *libros* que versen sobre temas relacionados.

En la figura 10.6 puede verse cómo se organizan en cadenas los componentes (o grafos, en este contexto es indiferente) según las relaciones TC y TC^{∞}; en la figura 10.7(a) según la relación NE; y en 10.7(b) según NE^{∞}.

Estos puntos de vista de la biblioteca, aunque distintos, se complementan debido a la estructura de retículo que mantienen las relaciones definidas (ver figura 10.4). Es decir, el más refinado o más restrictivo viene determinado por la relación TC^{∞}, siendo el de mayor granularidad el definido por NE.

10.9 Unión e intersección de componentes

Definición 10.42. *Dados dos componentes C y C' se define el componente $\mathcal{O}$-unión de ambos, si existe, a aquel componente que es la $\mathcal{O}$-*menor cota superior

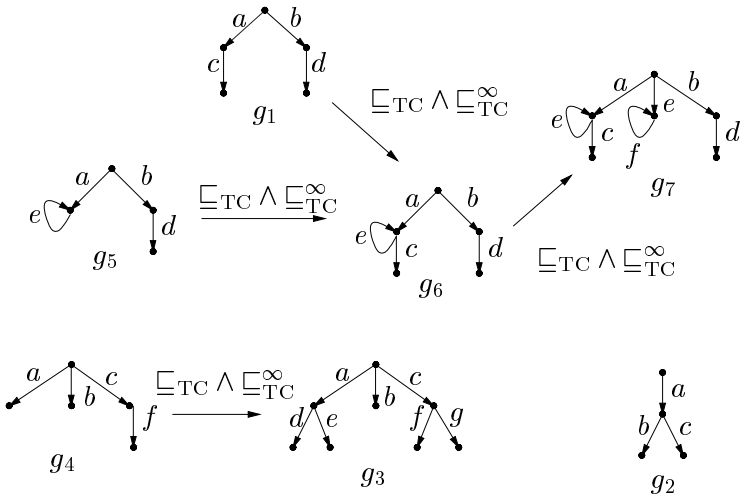


Figura 10.6. Ordenación atendiendo a las trazas de evolución

(supremo) de C y C', denotándose $C \cup_{\mathcal{O}} C'$,

El significado de un componente $\mathcal{O}$-unión de otros dos es sencillo, simplemente que, con el criterio definido por la relación $\mathcal{O}$, es el menor componente que puede reunir la funcionalidad de ambos.

EJEMPLO 10.6. Observando las diferentes figuras de almacenamiento según los diferentes criterios de ordenación definidos (figuras 10.6, 10.7(a), y 10.7(b)) puede verse que:

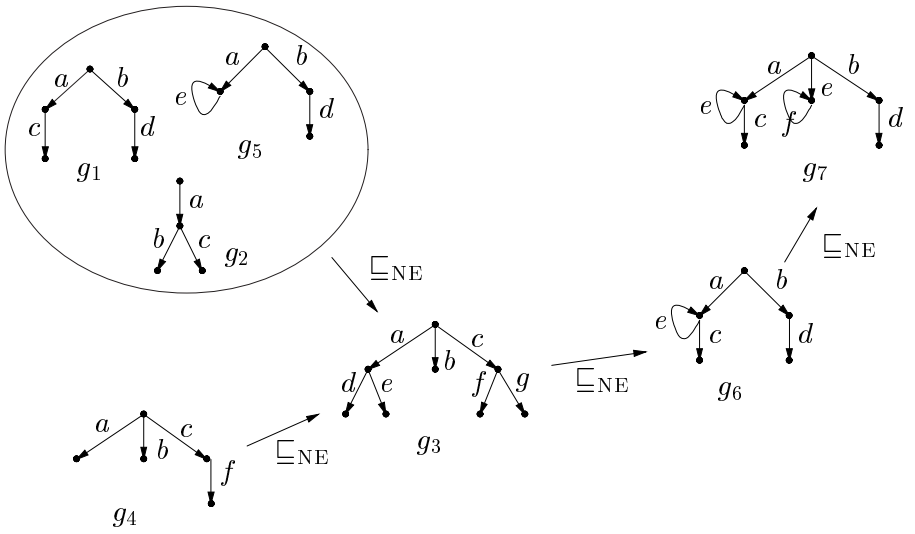
$$(g_4 \cup_{\mathrm{NE}} g_1) \sqsubseteq_{\mathrm{NE}} g_3$$
$$(g_3 \cup^{\infty}_{\mathrm{NE}} g_5) \sqsubseteq^{\infty}_{\mathrm{NE}} g_6$$
$$(g_5 \cup_{\mathrm{TC}} g_1) \sqsubseteq_{\mathrm{TC}} g_6$$

□

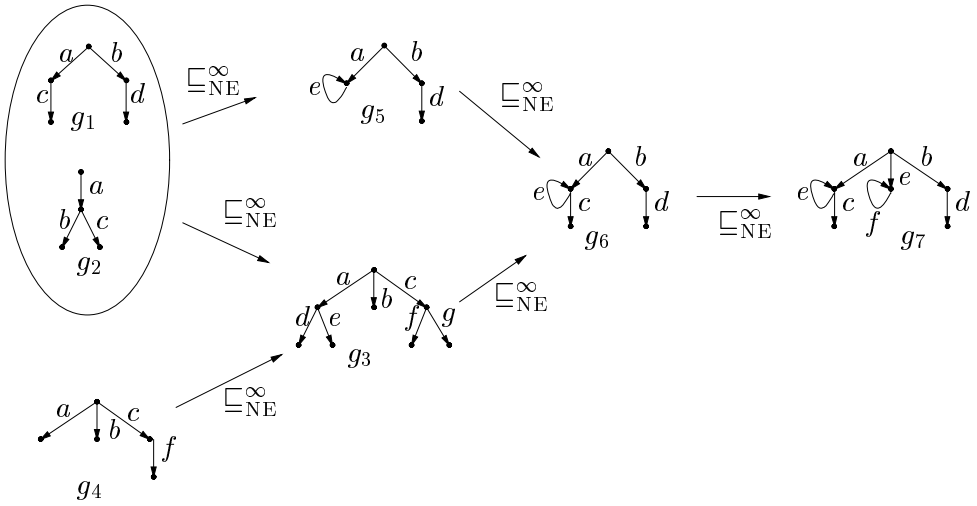
Definición 10.43. *Dados dos componentes C y C' se define el componente $\mathcal{O}$-intersección de ambos, si existe, a aquel componente que es la $\mathcal{O}$-*mayor cota inferior *(ínfimo) de C y C', denotándose $C \cap_{\mathcal{O}} C'$.*

El significado de un componente $\mathcal{O}$-intersección de otros dos es sencillo, simplemente que, con el criterio definido por la relación $\mathcal{O}$, este componente almacena la mayor funcionalidad común a ambos.

EJEMPLO 10.7. Observando las diferentes figuras de almacenamiento según los



(a) Ordenación según NE-orden



(b) Ordenación según NE^{∞}-orden

Figura 10.7. Ordenaciones atendiendo al número de evoluciones posibles

diferentes criterios de ordenación definidos (figuras 10.6, 10.7(a), y 10.7(b)) puede verse que:

$$g_1 \sqsubseteq^{\infty}_{\mathrm{NE}} (g_5 \cap^{\infty}_{\mathrm{NE}} g_3)$$

□

CAPÍTULO 11

Distancias y diferencias funcionales entre componentes

El propósito de este capítulo es definir, a partir de las relaciones identificadas en el capítulo 10, distancias que permitan cuantificar las diferencias de funcionalidad entre componentes reutilizables. El establecimiento de estos criterios de medida permitirá evaluar la proximidad funcional y estructural entre componentes, y la conveniencia de reutilizar un componente o, por el contrario, desecharlo ya que podría ser más costosa su adaptación que su construcción desde el principio.

11.1 Introducción

En el capítulo 10 se detallan las diferentes relaciones de equivalencia identificadas entre grafos MUS, que pueden extrapolarse directamente a los componentes que los contienen. Tal y como se introdujo, cada una de ellas, $\mathcal{O} \in \Gamma$, permite definir $(\mathbb{G}, \sqsubseteq_{\mathcal{O}})$ como un conjunto parcialmente ordenado, con lo que podríamos tener diferentes visiones de la biblioteca de componentes, atendiendo al criterio de observación.

Estas relaciones de precedencia expresan relaciones de tipo *contenido-continente* que, aunque interesantes, no son suficientes para graduar las diferencias entre componentes relacionados por inclusión, es decir, cuantificar *cuánto le falta al componente contenido para igualar al componente continente*. Para poder evaluar esta característica se hace imprescindible la definición de **distancias**.

Hemos diferenciado entre tres tipos o categorías de distancias o medidas de distancias que, ordenadas de menor a mayor precisión, evalúan las diferencias funcionales entre componentes almacenados en una biblioteca:

- **Distancia de clasificación**: cuantifica la distancia entre dos componentes según su ordenación en una cadena. Es directamente proporcional a la cantidad de información almacenada en la biblioteca y depende más de esto último que del verdadero parecido entre componentes, aunque es bastante útil para una primera aproximación. Se detalla en el apartado 11.2.

- **Distancias estructurales o sintácticas**: cuantifican la distancia entre dos componentes según el grado de parecido en su estructura, es decir, tendrán una menor distancia estructural cuanto más parecida sea su representación o estructura de su grafo. Se detalla en el apartado 11.3.

- **Diferencias funcionales o semánticas**: cuantifican las diferencias entre dos componentes según el parecido de sus especificaciones, es decir, estarán más próximos funcionalmente cuanto menor sea su diferencia de comportamiento, teniendo en cuenta en este caso las secuencias de eventos posibles, no posibles y subespecificadas que se tengan almacenadas en el grafo MUS. En el apartado 11.4 se detallan las tres aplicaciones básicas que permiten cotejar diferencias funcionales.

A la hora de hablar de componentes de código podría mantenerse esta clasificación de distancias pero, en este caso, la distancia estructural y la distancia funcional entre dos componentes podría ser muy diferente, incluso podría hablarse de ortogonalidad. Este fenómeno puede ilustrarse con un ejemplo sencillo: tenemos dos componentes de código o procedimientos, uno que se encarga de sumar las cantidades que recibe como parámetro y otro que se encarga de restarlas. Su parecido estructural es muy elevado ya que sólo difieren en una cadena de una instrucción, sin embargo su parecido funcional o semántico es totalmente opuesto (Jilani, 1997).

En nuestro caso, sin embargo, tratamos con componentes de elevado nivel de abstracción, modelos MUS de sistemas, donde la estructura del componente, grafo, sigue siendo lo suficientemente abstracta como para que mantenga la coherencia entre funcionalidad y representación; además la generación de modelos MUS se hace de forma semiautomática, con lo que no es posible tener dos representaciones *muy diferentes* con los mismos requisitos funcionales. Es por esto que podemos obviar este problema y tratar con distancias estructurales o de representación como una buena aproximación a distancias funcionales.

11.2 Distancia de clasificación

Definición 11.1. *Dada una cadena de componentes* $(\mathcal{C}\mathrm{h}, \sqsubseteq_{\mathcal{O}})$*, ordenada según una relación* $\mathcal{O} \in \Gamma$ *(ver apartado 10.7), se define la distancia de clasificación entre dos componentes* $C, C' \in \mathcal{C}\mathrm{h}$*, denotándose* $d_c(C, C')$*, como el mínimo número de transiciones que es necesario realizar sobre la cadena para evolucionar de uno a*

otro. Se define la distancia de clasificación entre dos componentes no comparables C *y* C' *como* $d_c(C, C') = \infty$.

Esta distancia se basa en la idea de que cuando más cercanos se encuentren dos componentes almacenados en la biblioteca, mayor será su parecido funcional. d_c satisface las propiedades de una distancia, ya que:

$$\begin{aligned} &d_c(C, C') = d_c(C', C) && \forall\, C, C' \\ &d_c(C, C') \geq 0 && \forall\, C, C' \\ &d_c(C, C') = 0 \;\Leftrightarrow\; C =_{\mathcal{O}} C' && \forall\, C, C' \\ &d_c(C, C') \leq d_c(C, C'') + d_c(C'', C') && \forall\, C, C', C'' \end{aligned}$$

Esta distancia es claramente dependiente del contexto de almacenamiento, ya que, cuanto mayor sea la cantidad de componentes almacenados, más refinada será la evaluación de esta distancia.

EJEMPLO 11.1. En la figura 11.1 puede verse como, en la cadena de componentes ordenada según la relación NE, la distancia de clasificación entre g_1 y g_3, $d_c(g_4, g_3) = 1$, es idéntica a la distancia entre g_5 y g_3, $d_c(g_5, g_3) = 1$. Sin embargo, es fácilmente observable que las estructuras de g_5 y g_4 son diferentes. □

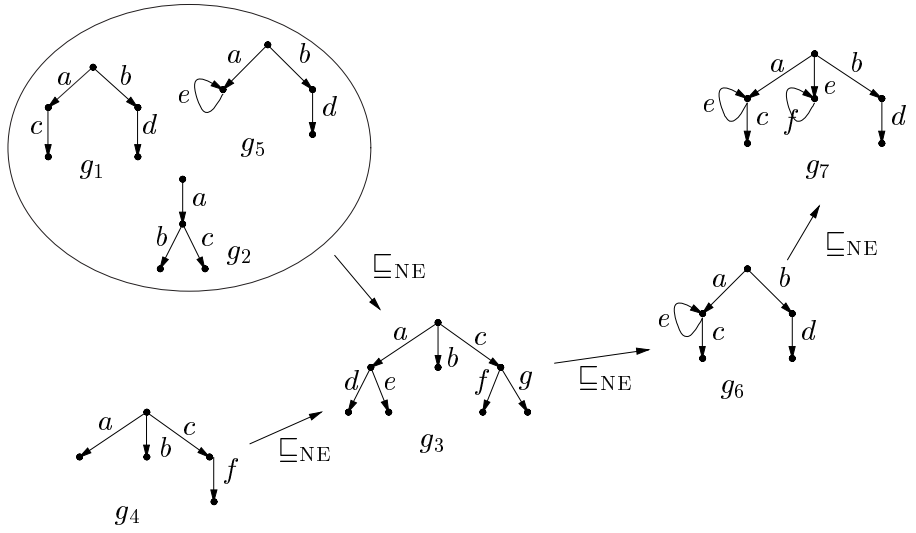


Figura 11.1. Cadena de componentes ordenada según la relación NE

Definición 11.2. *Dada una cadena de componentes* $(\mathcal{C}\mathrm{h}, \sqsubseteq_{\mathcal{O}})$*, ordenada según una relación* $\mathcal{O} \in \Gamma$*, se define el* **conjunto de antecesores** *de un componente* C*, denotándose* $\mathcal{A}(C)$*, como aquel que satisface:*

$$\mathcal{A}(C) = \{C_i \in \mathcal{C}\mathrm{h} \mid C_i \sqsubseteq_{\mathcal{O}} C \wedge d_c(C_i, C) = 1\}$$

Definición 11.3. *Dada una cadena de componentes* $(\mathcal{C}\mathrm{h}, \sqsubseteq_{\mathcal{O}})$*, ordenada según una relación* $\mathcal{O} \in \Gamma$*, se define el* **conjunto de sucesores** *de un componente* C*, denotándose* $\mathcal{S}(C)$*, como aquel que satisface:*

$$\mathcal{S}(C) = \{C_i \in \mathcal{C}\mathrm{h} \mid C \sqsubseteq_{\mathcal{O}} C_i \wedge d_c(C_i, C) = 1\}$$

Es necesario notar que esta distancia sólo es evaluable entre elementos comparables, con lo que no resulta posible calcular, utilizando d_c, la distancia existente entre cadenas de componentes. Para esto será mejor solución la aportada en el apartado 11.3.1.

11.3 Distancias estructurales

El grafo MUS de un componente almacena dos tipos de información: por un lado lo que es su estructura —el grafo desprovisto de acciones—; y por otro lo que es la funcionalidad —las acciones asociadas a las transiciones del grafo— es decir, la semántica del componente. Desde luego, ambas informaciones se complementan, pero en este caso puede estudiarse la estructura de los grafos de forma aislada al conjunto de los eventos que los ocasionan como aproximación a lo que sería el conjunto total del grafo.

Las dos distancias que se definen en este apartado son relativas a la estructura del componente, es decir, al grafo desprovisto de acciones, y pueden verse como una consecuencia de las relaciones de equivalencia NE y NE^{∞} definidas en los apartados 10.3 y 10.4.

11.3.1 Distancia del número de evoluciones totales

La distancia que se va a definir en este apartado es consecuencia directa de la relación NE definida entre grafos (apartado 10.3). Dicha relación permitía definir un orden parcial entre grafos y obtener cuándo un grafo estaba NE-contenido en otro, pero no permitía estimar la NE-diferencia entre ellos.

Definición 11.4. *Dados dos grafos* g *y* g' *se define la distancia del número de evoluciones totales, denotándose* $d_{NE}(g, g')$*, como el resultado de calcular la distancia euclídea de aquellos vectores que, contenidos en* $NE(g)$ *y* $NE(g')$ *respectivamente, no tengan ningún elemento común entre ellos, tengan el mismo número de elementos (en otro caso se completaría con ceros el de menor dimensión), y sus elementos estén ordenados de menor a mayor (el orden en este caso no altera la información contenida).*

EJEMPLO 11.2. En la figura 11.1 vemos que el conjunto de antecesores y suceso-

res de g_6 es el siguiente:

$$\mathcal{A}(g_6) \cup \mathcal{S}(g_6) = \{g_3, g_7\}$$

con lo que, por definición, su distancia de clasificación a g_6 es la unidad. Para refinar un poco más las diferencias de funcionalidad, vamos a recurrir a la distancia d_{NE} que nos permitirá saber cuál de ellos está más próximo, estructuralmente hablando:

$$NE(g_3) = (2,2,1,2,2) \qquad NE(g_7) = (2,2,2,2,3,3)$$
$$NE(g_6) = (2,2,2,3,3)$$

Se obtiene la distancia estructural de cada uno de los componentes a g_6:

$$d_{NE}(g_3, g_6) = d_{NE}((2,2,1,2,2),(2,2,2,3,3)) = \parallel (1,2) - (3,3) \parallel = \sqrt{5}$$
$$d_{NE}(g_7, g_6) = d_{NE}((2,2,2,2,3,3),(2,2,2,3,3)) = \parallel (0) - (2) \parallel = 2$$

De donde se deduce que g_6 mantiene una distancia de número de evoluciones totales con g_7 menor que con g_3. □

Aunque en la definición 11.4 hemos hablado de grafos, ésta puede hacerse directamente extensible a los componentes que los contienen, así que diremos que la $d_{NE}(C, C') = d_{NE}(g, g')$, donde g es el grafo MUS de C y g' el de C'. d_{NE} satisface las propiedades de una distancia ya que cumple:

$$d_{NE}(C, C') = d_{NE}(C', C) \qquad \forall\, C, C'$$
$$d_{NE}(C, C') \geq 0 \qquad \forall\, C, C'$$
$$d_{NE}(C, C') = 0 \;\Leftrightarrow\; C =_{\mathrm{NE}} C' \qquad \forall\, C, C'$$
$$d_{NE}(C, C') \leq d_{NE}(C, C'') + d_{NE}(C'', C') \qquad \forall\, C, C', C''$$

Definición 11.5. *Dado un grupo de componentes formando una cadena* $(\mathcal{C}\mathrm{h}, \sqsubseteq_{\mathrm{NE}})$*, se define la distancia entre un componente* $C \notin \mathcal{C}\mathrm{h}$ *a la cadena como:*

$$d_{NE}(C, \mathcal{C}\mathrm{h}) = \min\{d_{NE}(C, C_i)\}_{i=1}^{n}, \qquad C_i \in \mathcal{C}\mathrm{h},\ \forall i = 1, \ldots, n$$

Definición 11.6. *Dado dos grupos de componentes* $\mathcal{C}\mathrm{h}_1$ *y* $\mathcal{C}\mathrm{h}_2$*, se define la distancia entre ambos grupos como:*

$$d_{NE}(\mathcal{C}\mathrm{h}_1, \mathcal{C}\mathrm{h}_2) = \min\{d_{NE}(C_i, \mathcal{C}\mathrm{h}_2)\}_{i=1}^{m}, \qquad C_i \in \mathcal{C}\mathrm{h}_1,\ \forall i = 1, \ldots, m$$

De esta forma puede evaluarse el parecido estructural entre cadenas de componentes disjuntas.

11.3.2 Distancia del número de evoluciones totales no acotadas

En la distancia definida en el apartado 11.3.1 no se tenía en cuenta la posibilidad de que los componentes comparados tuviesen algún bucle o recursión de comportamiento. Para que esta característica influya a la hora de evaluar la distancia estructural de dos elementos, será necesario recurrir a la relación NE^{∞} definida en el apartado 10.4.

Dado el resultado de aplicar la relación NE^{∞} a un grafo g, $NE^{\infty}(g)$, éste puede ser interpretado como la composición de los dos vectores siguientes:

$$NE^{\infty}(g) = (NE_f^{\infty}(g), NE_i^{\infty}(g))$$

donde $NE_f^{\infty}(g)$ representará el resultado de aplicar la relación NE a todas las vías del conjunto $V_f(g)$, y donde $NE_i^{\infty}(g)$ representará el resultado de aplicar la relación NE^{∞} a todas las vías del conjunto $V_i(g)$.

Definición 11.7. *Dados dos grafos g y g', y sus correspondientes $NE^{\infty}(g)$ y $NE^{\infty}(g')$ se define el vector diferencia estructural $d\vec{f}_e(g, g')$ como aquel compuesto por:*

- *$d\vec{f}_{e1}$, obtenido por la unión de las componentes de $NE_f^{\infty}(g)$ y de $NE_f^{\infty}(g')$, tras haber eliminado las que eran comunes a ambas y haberlas ordenado de menor a mayor según la relación de orden propia de los números naturales; y*

- *$d\vec{f}_{e2}$, obtenido por la unión de las componentes de $NE_i^{\infty}(g)$ y de $NE_i^{\infty}(g')$, tras haber eliminado las que eran comunes a ambas y haberlas ordenado de menor a mayor según la relación de orden definida en 10.22.*

donde $d\vec{f}_e(g, g') = (d\vec{f}_{e1}, d\vec{f}_{e2})$.

La definición 11.4 no es más que una versión simplificada de esta última definición ya que, en aquel caso, los vectores d_{NE} constaban sólo de valores naturales, sin recursión.

Existen situaciones donde es posible evaluar fácilmente cuál, de entre varios componentes, se aproxima más a uno dado según el vector diferencia estructural definido en 11.7, pues se puede establecer una relación de NE^{∞}-orden entre los vectores de diferencia estructural correspondientes. Sin embargo, esta relación de orden no es siempre posible ya que la relación NE^{∞} define un orden parcial (ver apartado 10.4). Así que, para poder cuantificar en todos los casos la distancia estructural entre dos componentes, teniendo en cuenta sus comportamientos recursivos, se define la distancia del número de evoluciones totales no acotadas.

Definición 11.8. *Dados dos grafos g y g' se define la distancia del número de evoluciones totales no acotadas, denotándose $d^{\infty}_{\mathrm{NE}}(g, g')$, al resultado de calcular la norma del vector $\vec{df}_{e\infty} = (\vec{df}_{e1}, \vec{df}_{e2}{}')$, donde $\vec{df}_{e1}$ se calcula según la definición 11.7; y las componentes de $\vec{df}_{e2}{}'$ se obtienen a partir de las de $\vec{df}_{e2}$, eliminando de cada término la recursividad, $()+$, y restando una unidad al resultado.*

EJEMPLO 11.3. Dada la cadena de la figura 11.2, ordenada según la relación NE^{∞}, vamos a calcular cuál de los sucesores y antecesores del grafo g_6 está a una distancia menor según la distancia del número de evoluciones totales no acotadas definida en 11.8.

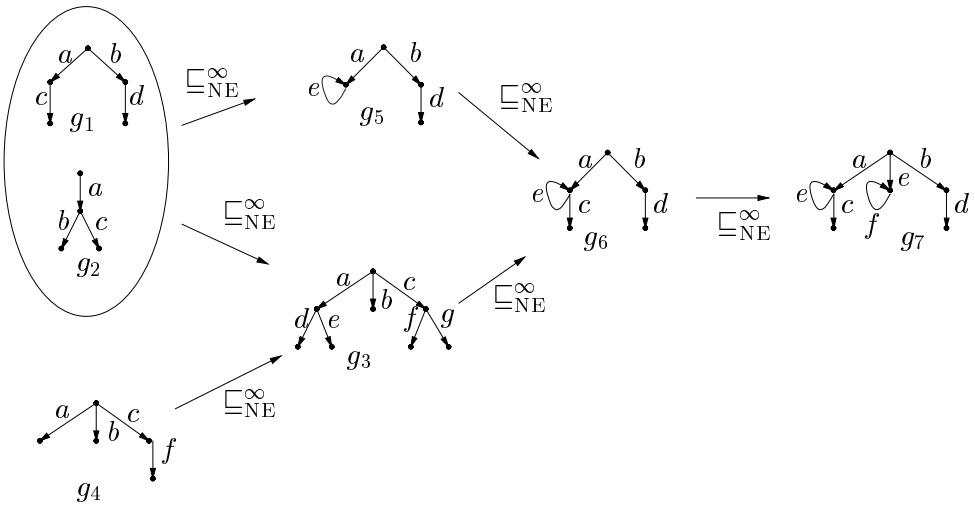


Figura 11.2. Cadena de componentes ordenada según la relación NE^{∞}

El conjunto de antecesores y sucesores de g_6 en la cadena dada es el siguiente:

$$\mathcal{A}(g_6) \cup \mathcal{S}(g_6) = \{g_5, g_3, g_7\}$$

Lo primero será obtener los vectores de diferencia estructural entre cada uno de ellos y g_6.

$$\begin{aligned} NE^{\infty}(g_6) &= (2, 2, 3, 2+, 3+) \\ NE^{\infty}(g_3) &= (2, 2, 1, 2, 2) \\ NE^{\infty}(g_5) &= (2, 2+) \\ NE^{\infty}(g_7) &= (2, 2, 3, 2+, 2+, 3+) \end{aligned}$$

de donde

$$\vec{df}_e(g_3, g_6) = (1, 2, 2, 3, 2+, 3+)$$
$$\vec{df}_e(g_5, g_6) = (2, 3, 3+)$$
$$\vec{df}_e(g_7, g_6) = (2+)$$

Puede verse con claridad que $\vec{df}_e(g_7, g_6) \sqsubseteq^{\infty}_{\mathrm{NE}} \vec{df}_e(g_5, g_6) \sqsubseteq^{\infty}_{\mathrm{NE}} \vec{df}_e(g_3, g_6)$, de donde podría concluirse que el componente g_7 es el más próximo estructuralmente, de todas formas calcularemos la distancia del número de evoluciones no acotadas:

$$\vec{df}_{e\infty}(g_3, g_6) = (1, 2, 2, 3, 1, 2)$$
$$\vec{df}_{e\infty}(g_5, g_6) = (2, 3, 2)$$
$$\vec{df}_{e\infty}(g_7, g_6) = (1)$$

De donde se obtiene que las distancias del número de evoluciones totales no acotadas son las siguientes:

$$\mathrm{d}^{\infty}_{\mathrm{NE}}(g_3, g_6) = \sqrt{23}$$
$$\mathrm{d}^{\infty}_{\mathrm{NE}}(g_5, g_6) = \sqrt{15}$$
$$\mathrm{d}^{\infty}_{\mathrm{NE}}(g_7, g_6) = 1$$

Concluyendo que el componente que tiene como grafo MUS a g_7 es el que está más próximo, según la relación NE^{∞}, a g_6. □

11.4 Diferencias semánticas

Para cuantificar la proximidad funcional entre dos componentes, las métricas definidas en 11.2 y 11.3 son una buena aproximación, pero tienen una carencia fundamental ya que no tienen en cuenta los eventos de las transiciones. La inclusión de los eventos o acciones que provocan las evoluciones de comportamiento del componente, proporciona la semántica del componente, es decir, su significado.

En este apartado se definen tres funciones: intersección funcional, diferencia funcional e inconsistencia funcional; que permiten obtener valores útiles para la cuantificación de las diferencias y parecidos en la semántica de los componentes reutilizables.

11.4.1 Vector intersección funcional

La posibilidad de obtener la intersección funcional entre dos componentes o grafos, nos permitiría calcular la funcionalidad común entre ellos y, con esta información, obtener aquella funcionalidad que sobra a uno y falta al otro para que sean funcionalmente idénticos.

Partiendo de la información obtenida aplicando las funciones TC^{∞} y TC a cada componente, tenemos una descripción de todas y cada una de las vías de evolución completas de un componente y, para cada vía, la secuencia de eventos observables que se producen. Va a ser partiendo de esta información como obtendremos nuestro vector de intersección funcional.

Definición 11.9. *Dadas dos trazas finitas $T(\pi)$ y $T(\pi')$ se define su traza intersección, denotándose $T(\pi) \cap T(\pi')$, de la siguiente forma:*

- *si $T(\pi) = T(\pi')$, la intersección de ambas será cualquiera de ellas,*
- *si $T(\pi) \subseteq T(\pi')$, la intersección de ambas será $T(\pi)$,*
- *si $T(\pi') \subseteq T(\pi)$, la intersección de ambas será $T(\pi')$,*
- *si $T(\pi)$ y $T(\pi')$ tienen una traza prefijo común a ambas, la intersección será dicha traza prefijo,*
- *en otro caso la intersección de ambas será nula.*

Definición 11.10. *Dada una traza finita $T(\pi)$ y un vector $TC(g)$, se define la intersección de la traza con el vector, denotándose $T(\pi) \sqcap TC(g)$, como*

$$T(\pi) \sqcap TC(g) = T(\pi) \cap T(\pi') \mid T(\pi') \in TC(g)$$

y donde el valor $T(\pi')$ satisface la siguiente condición:

$$\nexists\, T(\pi'') \in TC(g) \mid (T(\pi) \cap T(\pi')) \subseteq (T(\pi) \cap T(\pi''))$$

Definición 11.11. *Dados dos vectores $TC(g)$ y $TC(g')$, donde $\sharp g \leq \sharp g'$, se define su vector intersección funcional, denotándose $TC(g) \sqcap TC(g')$, como el formado por las intersecciones no nulas de cada uno de los elementos $T(\pi) \in TC(g)$ con el vector $TC(g')$.*
La intersección entre los elementos de $TC(g)$ y el vector $TC(g')$ se realiza de forma biunívoca, es decir, si existe una traza $T(\pi) \in TC(g)$ tal que ya se ha calculado su intersección con una traza $T(\pi') \in TC(g')$, entonces $T(\pi')$ no podrá ser utilizada para calcular la intersección de ninguna otra traza $T(\pi'') \in TC(g)$.

Definición 11.12. *Dadas dos trazas $T^{\infty}(\pi)$ y $T^{\infty}(\pi')$ se define su traza intersección, denotándose $T^{\infty}(\pi) \cap T^{\infty}(\pi')$, de la siguiente forma:*

- *si ambas son trazas finitas, la definición de la traza intersección se corresponde con la proporcionada en la definición 11.9,*
- *si una de ellas es una traza finita y la otra es una traza no acotada, existirá intersección entre ambas siempre que se pueda encontrar una traza finita $T(\pi'')$ que sea traza prefijo de ambas, siendo esta última la traza intersección,*
- *si ambas son trazas no acotadas, existirá intersección entre ambas siempre que se pueda encontrar una traza finita $T(\pi'')$ o una traza no acotada $T^{\infty}(\pi'')$, que sea prefijo de ambas, siendo esta última la traza intersección. En el caso particular de que $T^{\infty}(\pi) = T^{\infty}(\pi')$, la traza intersección resultante es cualquiera de ellas,*
- *en otro caso no existe una traza intersección.*

Definición 11.13. *Dada una traza no acotada $T^{\infty}(\pi)$ y un vector $TC^{\infty}(g)$, se define la intersección de la traza con el vector, denotándose $T^{\infty}(\pi) \sqcap TC^{\infty}(g)$, como*

$$T^{\infty}(\pi) \sqcap TC^{\infty}(g) = T^{\infty}(\pi) \cap T^{\infty}(\pi') \mid T^{\infty}(\pi') \in TC^{\infty}(g)$$

y donde el valor $T^{\infty}(\pi')$ satisface la siguiente condición:

$$\nexists\, T^{\infty}(\pi'') \in TC^{\infty}(g) \mid (T^{\infty}(\pi) \cap T^{\infty}(\pi')) \subseteq (T^{\infty}(\pi) \cap T^{\infty}(\pi''))$$

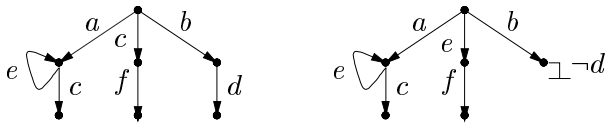
Definición 11.14. *Dados dos vectores $TC^{\infty}(g)$ y $TC^{\infty}(g')$, donde $\sharp g \leq \sharp g'$ se define su vector intersección funcional, denotándose $TC^{\infty}(g) \sqcap TC^{\infty}(g')$, como el formado por las intersecciones no nulas de cada uno de los elementos $T(\pi) \in TC^{\infty}(g)$ con el vector $TC^{\infty}(g')$.*
La intersección entre los elementos de $TC^{\infty}(g)$ y el vector $TC^{\infty}(g')$ se realiza de forma biunívoca, es decir, si existe una traza $T^{\infty}(\pi) \in TC^{\infty}(g)$ tal que ya se ha calculado su intersección con una traza $T^{\infty}(\pi') \in TC^{\infty}(g')$, entonces $T^{\infty}(\pi')$ no podrá ser utilizada para calcular la intersección de ninguna otra traza $T^{\infty}(\pi'') \in TC^{\infty}(g)$.

Definición 11.15. *Se define el vector de intersección funcional entre dos grafos g y g', denotándose $g \sqcap g'$ como aquel resultante de calcular la intersección entre los vectores $TC^{\infty}(g)$ y $TC^{\infty}(g')$, o bien entre los vectores $TC(g)$ y $TC(g')$. La decisión de adoptar un tipo de vector u otro radica en la existencia de recursiones en g y g', si alguno de los grafos contiene algún tipo de recursión entonces $g \sqcap g' = TC^{\infty}(g) \sqcap TC^{\infty}(g')$, en otro caso $g \sqcap g' = TC(g) \sqcap TC(g')$.*

EJEMPLO 11.4. Siguiendo con el ejemplo de la figura 11.1 vamos a obtener la intersección funcional entre dos grupos de componentes

$$g_3 \sqcap g_7 = (ad, ae, b, cf, cg) \sqcap (ae+, ae + c, aec, ac, ef+, bd) = (ae, a, b)$$

$$\begin{aligned} g_6 \sqcap g_7 &= (ae+, ae + c, aec, ac, bd) \sqcap (ae+, ae + c, aec, ac, ef+, bd) \\ &= (ae+, ae + c, aec, ac, bd) = g_6 \end{aligned}$$



(a) g (b) g'

Figura 11.3. Intersección y diferencia funcionales

En el caso de los grafos de las figuras 11.3(a) y 11.3(b), tenemos que ni g ni g' mantienen una relación de orden definida por TC^{∞}. En este caso

$$\begin{aligned} TC^{\infty}(g) &= (ae+, ae + c, aec, ac, bd, cf) \\ TC^{\infty}(g') &= (ae+, ae + c, aec, ac, b\neg d, ef), \end{aligned}$$

con lo que

$$g \sqcap g' = (ae+, ae + c, aec, ac, b)$$

□

11.4.2 Vector diferencia funcional

Además de la funcionalidad común entre dos componentes, dada por su intersección funcional, nos interesa conocer la funcionalidad que le falta a uno para completar la especificada en el otro.

Definición 11.16. *Dados dos grafos MUS g y g', tal que alguno de ellos presenta recursiones de funcionalidad, y satisfaciendo que $g \sqsubseteq^{\infty}_{\mathrm{TC}} g'$, se define el vector de diferencia funcional entre g' y g, denotándose $g' \ominus g$, como el vector compuesto por los componentes de $TC^{\infty}(g')$ que no se encuentran en $TC^{\infty}(g)$.*

Dados dos grafos MUS g y g', tal que ninguno de ellos presenta recursiones de funcionalidad, y satisfaciendo que $g \sqsubseteq_{\mathrm{TC}} g'$, se define el vector de diferencia funcional entre g' y g, denotándose $g' \ominus g$, como el vector compuesto por los componentes de $TC(g')$ que no se encuentran en $TC(g)$.

EJEMPLO 11.5. En el ejemplo 11.4 veíamos como $g_6 \sqsubseteq^{\infty}_{\mathrm{TC}} g_7$, y calculábamos su vector de intersección funcional. Aquí vamos a obtener el vector de diferencia funcional, es decir, la funcionalidad que habría que especificar en g_6 para igualar la funcionalidad de g_7:

$$TC^{\infty}(g_6) = (ae+, ae + c, aec, ac, bd)$$
$$TC^{\infty}(g_7) = (ae+, ae + c, aec, ac, ef+, bd)$$
$$g_7 \ominus g_6 = (ef+)$$

□

11.4.3 Vector de inconsistencia funcional

Además de la funcionalidad que tienen en común dos componentes —expresada por el vector de intersección funcional— y de poder conocer la funcionalidad que le falta a un componente para igualar a la de su componente continente —expresada por el vector diferencia funcional—, también nos será útil conocer si existe algún tipo de contradicción entre los comportamientos posibles de dos componentes. El vector de inconsistencia funcional expresa precisamente esto último, es decir, aquellas vías de evolución que, siendo posibles en un componente, son malogradas en otro.

Definición 11.17. *Dados dos grafos MUS g y g', se define el vector de inconsistencia funcional, denotándose $g \oslash g'$, como aquel formado por los componentes de $TC^{\infty}(g)$ y $TC^{\infty}(g')$ que son contradictorios, en caso de que alguno de los grafos presente recursión, o bien como aquel formado por los componentes de $TC(g)$ y $TC(g')$ que son contradictorios, en caso contrario.*

EJEMPLO 11.6. En el ejemplo 11.4 teníamos que

$$TC^{\infty}(g) = (ae+, ae + c, aec, ac, bd, cf)$$
$$TC^{\infty}(g') = (ae+, ae + c, aec, ac, b\neg d, ef),$$

de donde es fácil concluir que el vector de inconsistencia de ambos grafos vendrá dado por:

$$g \oslash g' = (bd, b\neg d)$$

□

El vector de inconsistencia mide aquella funcionalidad que es contradictoria, es decir, aquella que impide que, simplemente añadiendo vías de evolución a uno u otro componente la funcionalidad de éstos se iguale.

PARTE IV

Reutilización de componentes: clasificación, búsqueda y adaptación

CAPÍTULO 12

Clasificación de componentes

En capítulos anteriores se han cimentado las bases teóricas que permiten la ordenación parcial de componentes reutilizables atendiendo a cuatro criterios diferentes, y se han definido distancias y aplicaciones de cotejo de funcionalidad como consecuencia de dichos criterios de ordenación. En este capítulo se verá cómo aplicar estas bases teóricas para permitir la clasificación de componentes reutilizables en una biblioteca, de forma que su posterior recuperación pueda ser realizada de forma efectiva y eficiente.

12.1 Estructuras reticulares de componentes

Dado un conjunto A, se dice que una relación binaria $\leq$ es una relación de orden para A si dicha relación satisface las propiedades reflexiva, antisimétrica y transitiva. Además, $\leq$ será de orden parcial si no todos los elementos de A pueden relacionarse según $\leq$.

Las relaciones de orden que se pueden establecer entre los elementos de A según $\leq$ definen una estructura en forma de red, o estructura reticular, que puede ser representada gráficamente según un diagrama como los incluidos en el capítulo 11.

Según las relaciones de orden definidas en el capítulo 10 podemos decir entonces que dado cualquier conjunto $G \in \mathbb{G}$ se cumple que $(G, \sqsubseteq_{\mathrm{NE}})$, $(G, \sqsubseteq^{\infty}_{\mathrm{NE}})$, $(G, \sqsubseteq_{\mathrm{TC}})$, y $(G, \sqsubseteq^{\infty}_{\mathrm{TC}})$ son conjuntos parcialmente ordenados, manteniendo una estructura reticular.

La posibilidad de mantener estas estructuras para cada uno de los cuatro criterios de ordenación entre grafos definidos en el capítulo 10 —número de evoluciones, número de evoluciones no acotadas, trazas completas y trazas completas no acotadas— permite establecer cuatro formas de almacenar los componentes

reutilizables, según las relaciones de orden (*contenido-continente*) definidas entre sus correspondientes grafos MUS.

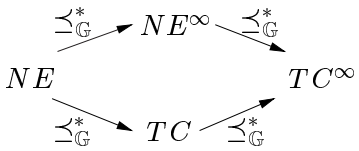
Estas redes de componentes permiten el almacenamiento ordenado de los mismos atendiendo a sus parecidos de funcionalidad según dos grandes criterios:

– similitudes estructurales, y

– similitudes semánticas.

Tal y como se introdujo en el apartado 11.1, las similitudes estructurales se basarán fundamentalmente en la estructura de su grafo MUS —definidas según las relaciones de orden $\sqsubseteq_{\mathrm{NE}}$ y $\sqsubseteq^{\infty}_{\mathrm{NE}}$—, y las similitudes semánticas tienen en cuenta las secuencias de eventos posibles, no posibles y subespecificados en dicho grafo —definidas según las relaciones de orden $\sqsubseteq_{\mathrm{TC}}$ y $\sqsubseteq^{\infty}_{\mathrm{TC}}$.

Teniendo en cuenta, además, que estas cuatro relaciones de orden mantienen, a su vez, una relación de orden parcial (ver apartado 10.7) tal y como puede verse en la siguiente figura:



entonces puede concluirse que aunque expresando diferentes relaciones de tipo *contenido-continente*, éstas no son totalmente independientes, es más, puede decirse que son complementarias.

Como tanto la información de relaciones estructurales, como la información de relaciones semánticas, será de vital importancia a la hora de recuperar los componentes (ver capítulo 13), y a la hora de adaptarlos a los requisitos solicitados en la consulta (ver capítulo 14), hemos optado por mantener la estructura de las cuatro retículas definidas. Así, nuestra biblioteca de componentes almacenará el conjunto de componentes reutilizables del que se disponga y, además, mantendrá información sobre las relaciones de *contenido-continente* consecuencia de los cuatro criterios de orden definidos.

12.2 Componente reutilizable

Hasta ahora se ha utilizado frecuentemente el término de *componente reutilizable* sin que, realmente, se haya definido. Tal y como se introdujo en la revisión bibliográfica, existen multitud de definiciones de componentes reutilizables, casi tantas como entornos de reutilización definidos. Sin embargo, todas

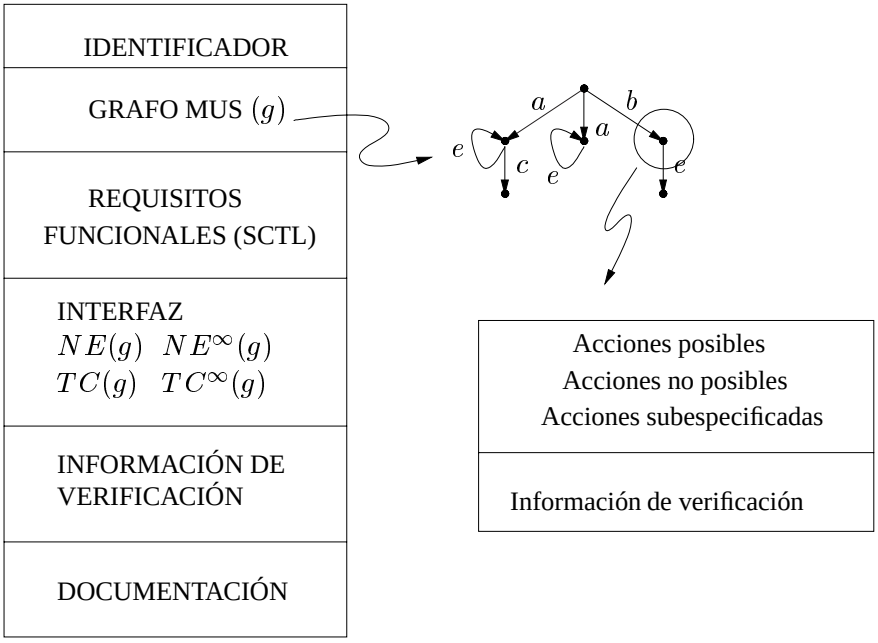


Figura 12.1. Definición de componente reutilizable

ellas coinciden en que los componentes deben ser autocontenidos, describir una funcionalidad clara y documentada, y proporcionar una interfaz apropiada. Estas características facilitarán las tareas de clasificación, recuperación y adaptación necesarias en cada entorno. En el contexto en el que se enmarca este trabajo, se ha particularizado la definición de componente reutilizable, manteniendo también las características anteriormente citadas.

Nuestro componente reutilizable constará de la información indicada en la figura 12.1:

- **Identificador de componente**: cada uno de los componentes de la biblioteca constará de un identificador que lo caracteriza unívocamente.

- **Grafo MUS**: el grafo MUS de un componente almacena toda la información de comportamiento del mismo, los diferentes estados de que consta y las evoluciones posibles entre ellos. Cada estado, además, almacenará la información siguiente:
  - conjunto de eventos posibles, no posibles y subespecificados; e
  - información de verificación de las propiedades que se han comprobado sobre el grafo.

- **Requisitos funcionales (SCTL)**: el grafo MUS satisface todos los requisi-

INTERFAZ	
$NE(g)$	ANTECESORES
	SUCESORES
	EQUIVALENTES
$NE^{\infty}(g)$	ANTECESORES
	SUCESORES
	EQUIVALENTES
$TC(g)$	ANTECESORES
	SUCESORES
	EQUIVALENTES
$TC^{\infty}(g)$	ANTECESORES
	SUCESORES
	EQUIVALENTES

Figura 12.2. Información de clasificación de un componente reutilizable

tos funcionales almacenados en este campo, ya que fue sintetizado a partir de ellos.

➢ **Interfaz:** la interfaz de un componente es vital para su clasificación y posterior recuperación de la biblioteca de componentes. Consta de los resultados de aplicar las funciones NE, TC, NE^{∞} y TC^{∞} sobre el grafo MUS.

➢ **Información de verificación**: cada componente almacenará también información relativa a los resultados de verificar diferentes propiedades sobre su grafo MUS (ver capítulo 15).

➢ **Documentación:** es preciso almacenar también toda la documentación interesante sobre dicho componente: datos sobre su creación, información sobre su reutilización, etc.

Algoritmo 12.1 Almacenar en la biblioteca un componente

si (la biblioteca está vacía) **entonces**
– Insertar el componente en la biblioteca, sin crear ningún enlace.
– Fin del algoritmo.
fin de la condición
– Buscar en la biblioteca todos los componentes origen de una cadena y almacenar el resultado en la variable lista.
para todo (componente_aux en lista) **entonces**
en caso de que (la comparación entre componente_aux y componente)
sea (no están relacionados) **entonces**
– Agregar componente, que no está relacionado con componente_aux, en la subcadena de la que componente_aux es origen (algoritmo 12.3).

sea (son equivalentes) **entonces**
– Insertar el componente en la biblioteca, si no ha sido almacenado antes.
– Crear el enlace de equivalencia correspondiente.

sea (componente_aux está contenido en componente) **entonces**
– Agregar componente, que es mayor que componente_aux, en la subcadena de la que componente_aux es origen (algoritmo 12.2).

sea (componente está contenido en componente_aux) **entonces**
– Insertar el componente en la biblioteca, si no ha sido almacenado antes.
– Crear el enlace desde componente hasta componente_aux;

fin (en caso de que)
fin de la iteración

si (no se ha almacenado el componente) **entonces**
– Insertar el componente en la biblioteca. (No se crea ningún enlace)
en otro caso
– Ajustar los enlaces de componente para evitar enlaces redundantes.
fin de la condición

12.3 Clasificación y almacenamiento de un componente reutilizable

El proceso de clasificación y almacenamiento de un componente reutilizable pasa necesariamente por la creación de dicho componente, según las características enumeradas en la sección 12.2, en la obtención de las relaciones de *contenido-continente* según los cuatro criterios de orden previamente definidos (información de clasificación), y en su inclusión en la base de datos o biblioteca de componentes.

Con el objetivo de almacenar esta información de clasificación, cada componente consta de un campo *INTERFAZ* con la estructura esbozada en la figura 12.2. Para cada una de las cuatro relaciones de orden definidas, se obtendrán los componentes antecesores, sucesores y equivalentes, es decir, su posición en la red correspondiente.

El pseudocódigo del algoritmo de clasificación y almacenamiento de un componente en la biblioteca se esboza en el algoritmo 12.1. Este algoritmo almacenará el componente que recibe como parámetro teniendo en cuenta una de las relaciones de orden parcial definidas. Es fácil observar que el algoritmo es idéntico para todas ellas, la única diferencia radica en el criterio de comparación entre componentes.

El algoritmo de clasificación y almacenamiento de un componente en la biblioteca se encarga de detectar las cadenas existentes (el comienzo de una cadena será un componente que no tenga ningún otro menor que él) y llamar a otros algoritmos recursivos que las recorran y busquen la posición adecuada para el nuevo componente. El pseudocódigo de estos algoritmos se esboza en los algoritmos 12.2 y 12.3 respectivamente.

Un resultado típico del almacenamiento de componentes, según una estructura de red definida por uno de los criterios de orden, puede verse en la figura 12.3. Aunque las cadenas obtenidas suelen estar entrelazadas por componentes comunes, el algoritmo de clasificación y almacenamiento evita la redundancia en los enlaces.

12.4 Aspectos prácticos

En los pseudocódigos de los algoritmos 12.1, 12.2, y 12.3 se ha expresado escuetamente la forma en la que se realiza la inclusión de un componente en un retículo de la biblioteca, pero sin entrar en demasiados detalles de implementación en aras de una mayor claridad en la descripción de la funcionalidad. Sin embargo, en este apartado, vamos a notar dos variaciones en la implementación que permiten obtener una mayor eficiencia en la ejecución de los algoritmos.

Algoritmo 12.2 Almacenar un componente_nuevo en la subcadena generada por componente_anterior, sabiendo que este último está contenido en componente_nuevo

– Buscar todos los componentes sucesores de componente_anterior y almacenarlos en una variable de nombre lista
si (la lista está vacía) **entonces**
– Insertar el componente_nuevo en la biblioteca, si no ha sido almacenado antes.
– Crear un enlace desde componente_anterior a componente_nuevo.
– Fin de la iteración.
fin de la condición
para todo (componente_aux en lista) **entonces**
en caso de que (la comparación entre componente_aux y componente_nuevo)
sea (no están relacionados) **entonces**
– Incrementar el contador de no relaciones (no_relacion).

sea (son equivalentes) **entonces**
– Almacenar el componente_nuevo en la biblioteca, siempre que no haya sido insertado previamente.
– Crear un enlace de equivalencia entre componente_nuevo y componente_aux.

sea (componente_aux está contenido en componente_nuevo) **entonces**
– Llamada recursiva a esta misma función con el objetivo de almacenar componente_nuevo, que es mayor que componente_aux, en la subcadena generada por este último.

sea (componente_nuevo está contenido en componente_aux) **entonces**
– Insertar componente_nuevo en la biblioteca, siempre que no haya sido almacenado previamente.
– Crear el enlace desde componente_nuevo hasta componente_aux y el enlace desde componente_anterior hasta componente_nuevo, eliminando el existente desde componente_anterior hasta componente_aux.

fin (en caso de que)
fin de la iteración

si (el número de no relaciones (no_relacion) es igual al número de elementos en lista) **entonces**
– Insertar el componente en la biblioteca, siempre que no haya sido almacenado previamente.
– Crear un enlace desde componente_anterior hasta componente_nuevo.
fin de la condición

Algoritmo 12.3 Almacenar un componente_nuevo en la subcadena generada por componente_norelacionado, sabiendo que no guardan entre sí ninguna relación de orden

– Buscar todos los componentes sucesores de componente_norelacionado y almacenarlos en una variable de nombre lista
si (la lista está vacía) **entonces**
– Fin de la iteración.
fin de la condición
para todo (componente_aux en lista) **entonces**
en caso de que (la comparación entre componente_aux y componente_nuevo)
sea (no están relacionados) **entonces**
– Llamada recursiva a este mismo algoritmo con el objetivo de insertar el elemento componente_nuevo en la subcadena generada por componente_aux, sabiendo que no existe ninguna relación entre ambos.

sea (componente_nuevo está contenido en componente_aux) **entonces**
– Insertar el componente_nuevo en la biblioteca, siempre que no haya sido almacenado previamente.
– Crear un enlace desde componente_nuevo hasta componente_aux.

fin (en caso de que)
fin de la iteración

➢ **Evitar comparaciones superfluas**: Dado que es muy frecuente la compartición de componentes por parte de las cadenas, esto podría provocar multiplicidad en las comparaciones entre el componente a insertar y los componentes almacenados, con la consiguiente ralentización en la ejecución. El algoritmo implementado tiene en cuenta esta situación y evita totalmente las comparaciones innecesarias.

➢ **Comienzo aleatorio de las comparaciones**: Para la localización de la posición apropiada de un componente en la biblioteca, el algoritmo de clasificación comienza la búsqueda desde los elementos menores, es decir, aquellos que son cabeza de una cadena —no existe ningún otro componente menor. Una forma de mejorar la eficiencia del algoritmo es variar el punto de la cadena a partir del que se comienza la búsqueda de forma que éste sea un componente escogido al azar, de esta forma se disminuyen, en media, las comparaciones. El pseudocódigo del algoritmo de clasificación mejorado es muy semejante al reflejado en el algoritmo 12.1, y se detalla en el algoritmo 12.4.

Aunque una mejora en la eficiencia de los algoritmos de clasificación repercute positivamente en el entorno de reutilización, es necesario notar que la eficiencia es mucho más vital a la hora de recuperar componentes de la base de datos. Las tareas de clasificación y almacenamiento de componentes pueden ser realizadas

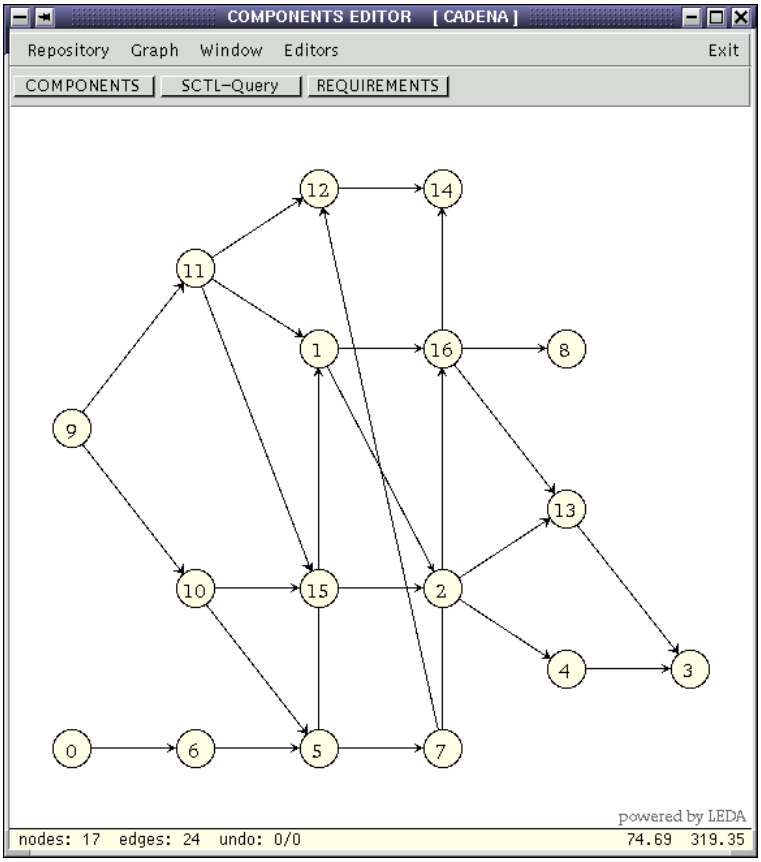


Figura 12.3. Ejemplo de red de componentes almacenada en la biblioteca según uno de los criterios de orden parcial definidos

Algoritmo 12.4 Almacenar en la biblioteca un componente (versión mejorada)

si (la biblioteca está vacía) **entonces**
– Insertar el componente en la biblioteca, sin crear ningún enlace.
– Fin del algoritmo.
fin de la condición
– Obtener, de forma aleatoria, un componente de la biblioteca y almacenarlo en la variable componente_aux.
en caso de que (la comparación entre componente_aux y componente)
sea (no están relacionados) **entonces**
– Agregar componente, que no está relacionado con componente_aux, en la subcadena de la que componente_aux es origen (algoritmo 12.3).

sea (son equivalentes) **entonces**
– Insertar el componente en la biblioteca, si no ha sido almacenado antes.
– Crear el enlace de equivalencia correspondiente.

sea (componente_aux está contenido en componente) **entonces**
– Agregar componente, que es mayor que componente_aux, en la subcadena de la que componente_aux es origen (algoritmo 12.2).

sea (componente está contenido en componente_aux) **entonces**
– Agregar componente, que es menor que componente_aux, en la subcadena prefijo de la que componente_aux es origen (algoritmo 12.5).

fin (en caso de que)

si (no se ha almacenado el componente) **entonces**
– Insertar el componente en la biblioteca. (No se crea ningún enlace)
en otro caso
– Ajustar los enlaces de componente para evitar enlaces redundantes.
fin de la condición

Algoritmo 12.5 Almacenar un componente_nuevo en las subcadenas prefijos de componente_posterior, sabiendo que el primero está contenido en este último

– Buscar todos los componentes antecesores de componente_posterior y almacenarlos en una variable de nombre lista
si (la lista está vacía) **entonces**
– Insertar el componente_nuevo en la biblioteca, si no ha sido almacenado antes.
– Crear un enlace desde componente_nuevo a componente_posterior.
– Fin de la iteración.
fin de la condición
para todo (componente_aux en lista) **entonces**
en caso de que (la comparación entre componente_aux y componente_nuevo)
sea (no están relacionados) **entonces**
– Incrementar el contador de no relaciones (no_relacion).

sea (son equivalentes) **entonces**
– Almacenar el componente_nuevo en la biblioteca, siempre que no haya sido insertado previamente.
– Crear un enlace de equivalencia entre componente_nuevo y componente_aux.

sea (componente_nuevo está contenido en componente_aux) **entonces**
– Llamada recursiva a esta misma función con el objetivo de almacenar componente_nuevo, que es menor que componente_aux, en la subcadena prefijo de este último.

sea (componente_aux está contenido en componente_nuevo) **entonces**
– Insertar componente_nuevo en la biblioteca, siempre que no haya sido almacenado previamente.
– Crear el enlace desde componente_aux hasta componente_nuevo, crear un enlace desde componente_nuevo hasta componente_posterior, y eliminar el que había entre componente_aux y componente_posterior

fin (en caso de que)
fin de la iteración

si (el número de no relaciones (no_relacion) es igual al número de elementos en lista) **entonces**
– Insertar el componente en la biblioteca, siempre que no haya sido almacenado previamente.
– Crear un enlace desde componente_nuevo hasta componente_posterior.
fin de la condición

off-line, así que en este caso los tiempos no son tan críticos. Sin embargo, la localización de componentes adecuados en la base de datos sí debe ser motivo de toda la optimización posible, ya que ésta se produce bajo demanda, durante el propio proceso de desarrollo.

CAPÍTULO 13

Recuperación de componentes

En los capítulos 10 y 11 se sentaron las bases teóricas que permitirán, en este capítulo, describir cómo se localizan aquellos componentes más adecuados en la biblioteca de componentes reutilizables. La recuperación de estos componentes se realizará atendiendo a criterios de similitud funcional con el patrón de búsqueda o consulta. Dicha consulta puede realizarse de dos formas diferentes: partiendo de la especificación funcional expresada en SCTL, o bien partiendo de un grafo MUS.

13.1 Introducción

¿Bajo qué circunstancias puede interesarle a un desarrollador de software reutilizar un modelo de sistema almacenado en nuestra biblioteca de componentes? Nosotros hemos identificado dos situaciones diferentes:

- al comienzo del desarrollo del sistema cuando, dado un conjunto de especificaciones funcionales expresadas según SCTL, espera encontrar algún modelo *funcionalmente parecido*, es decir, que satisfaga en la mayor medida posible la funcionalidad de partida; o bien,

- durante el desarrollo de un sistema cuando, dado un modelo incompleto sobre el que debe verificar el grado de satisfacción de una propiedad funcional, espera encontrar algún otro modelo *funcionalmente parecido* al original sobre el que ya se hayan realizado labores de verificación de dicha propiedad.

En ambas circunstancias realizará una consulta sobre la biblioteca de componentes reutilizables: en el primer caso el patrón de búsqueda vendrá determinado

por un requisito funcional expresado en SCTL; y en el segundo caso el patrón de búsqueda vendrá determinado por un modelo incompleto MUS.

En este capítulo se abordará la recuperación de componentes con el objeto de reutilizar sus modelos de comportamiento MUS, en el capítulo 16 se tratará la recuperación de componentes con el objetivo de reutilizar su información de verificación. En esta situación se intentará localizar a aquel o aquellos componentes que sean funcionalmente parecidos a lo solicitado. Dado que es poco probable que se encuentre en la base de datos un componente que se comporte exactamente como especificó el usuario y que pueda ser reutilizado sin realizar ninguna modificación —lo que se conoce como reutilización de *cajas negras*—, será preciso realizar alguna modificación sobre aquellos componentes recuperados para satisfacer plenamente la funcionalidad requerida, conocido como reutilización de *cajas blancas*. En la figura 13.1 se esquematizan los pasos que deberá seguir este proceso de reutilización de componentes.

- **Obtención del perfil o patrón de búsqueda**
  Normalmente en esta primera etapa la funcionalidad requerida por el usuario se expresa según la lógica SCTL. Para poder buscar en la biblioteca o base de datos de componentes aquellos funcionalmente próximos a estos requisitos será necesario expresar esta funcionalidad en términos de NE, NE^{∞}, TC y TC^{∞}, ya que en estos términos se han ordenado los componentes en la base de datos (capítulo 12).
  En el apéndice A se detalla como extraer estos perfiles dado un grafo MUS. Para la obtención de estos mismos datos partiendo de requisitos SCTL, se remite al lector al apéndice B.

- **Localización de componentes funcionalmente próximos a la consulta**
  Una vez que se tienen los patrones que permiten la búsqueda, se procede a localizar dentro de la base de datos aquellos componentes más parecidos a lo requerido. A lo largo de este capítulo se detallará cómo realizar esta búsqueda de componentes. Aquí sólo vamos a adelantar que se realiza en dos fases o etapas y según dos criterios diferentes: se busca a aquel o aquellos componentes más próximos estructuralmente a la consulta y más próximos, semánticamente hablando, a la misma.

- **Selección y adaptación funcional a la consulta**
  De la fase de localización se obtienen uno o más componentes cuyo esfuerzo de adaptación a la consulta es muy semejante. Llegados a este punto será el usuario quien decida cuál de ellos será el seleccionado finalmente. Una vez que se ha escogido un componente reutilizable apropiado para ser adaptado a los requisitos funcionales pedidos, se procederá a las tareas de adaptación propiamente dichas. En el capítulo 14 se explica detalladamente este proceso.

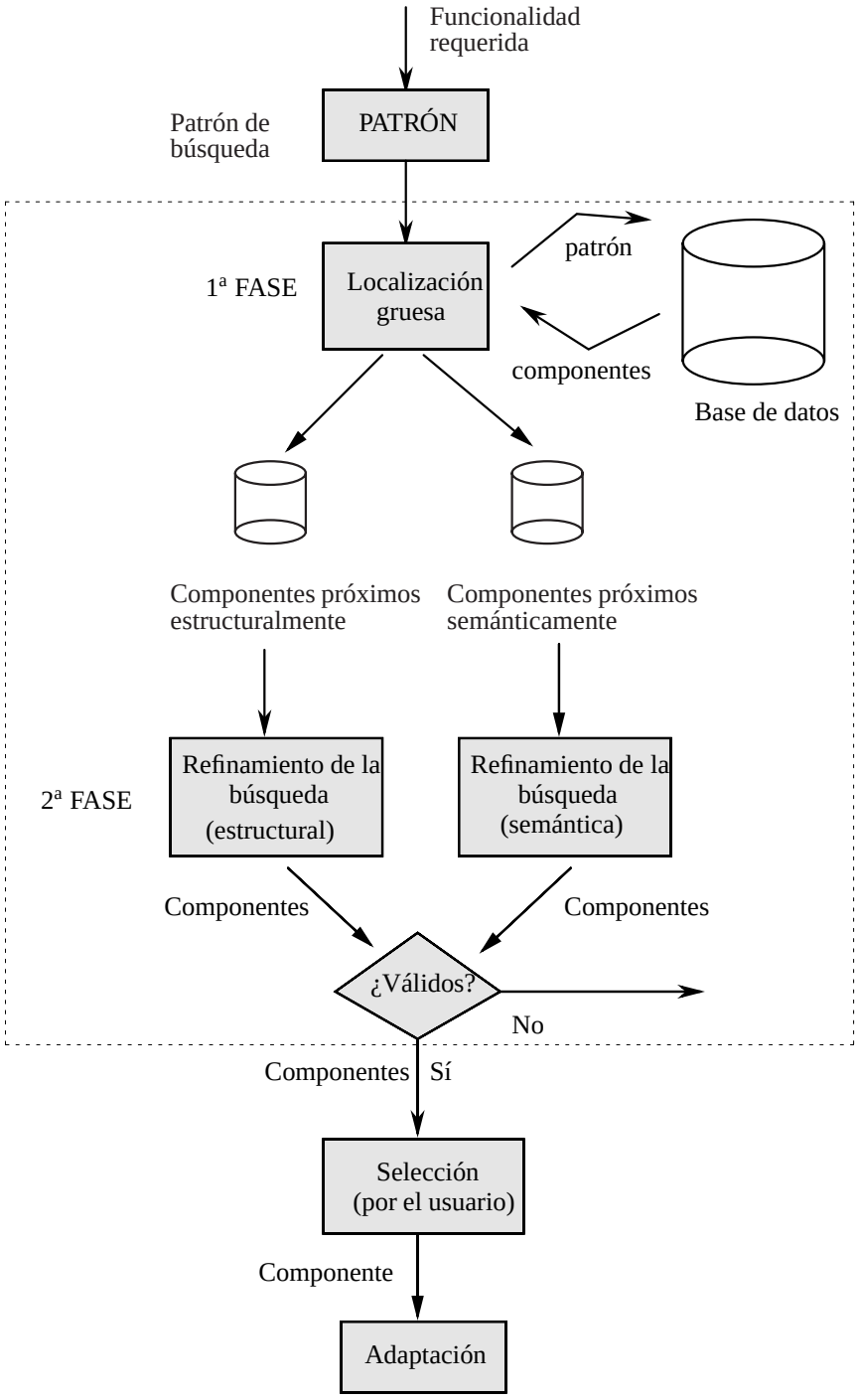


Figura 13.1. Esquema de la reutilización de componentes

13.2 Estructura de la búsqueda

Es obvio que deseamos encontrar de, entre los componentes almacenados $\mathcal{C} = \{C_i\}_{i=1}^{m}$, aquel o aquellos que sean más cercanos en funcionalidad a los requisitos expresados en la consulta, Q. Es inmediato, también, que aquellos componentes que estén próximos a la consulta según la relación TC^{∞}, también lo estarán según las otras relaciones (consecuencia de la ordenación parcial entre criterios definida en el apartado 10.7). Sin embargo, también es posible encontrar algún componente muy próximo o incluso equivalente según la relación NE al patrón de la consulta y que no esté en absoluto próximo según la relación TC^{∞}. Estas dos situaciones no son más que la consecuencia de primar, en el caso de la relación TC^{∞}, la semejanza semántica entre componentes, y de primar la semejanza estructural en el caso de la relación NE.

La búsqueda de componentes *funcionalmente semejantes* a una consulta dada se ha organizado atendiendo a dos amplios criterios:

– recuperación de componentes estructuralmente semejantes a la consulta, y

– recuperación de componentes semánticamente semejantes a la consulta.

Como resultado de ambas tareas se obtendrá un subconjunto de componentes próximos, estructuralmente hablando, a la consulta y otro subconjunto de componentes próximos, semánticamente hablando, a la misma consulta. Esto constituirá el primer paso o fase de una **búsqueda escalonada** (apartado 13.3). La decisión de dividir las labores de localización en dos fases, una primera aproximada y una segunda más refinada, no es en absoluto novedosa (consultar el apartado 7.3) y atiende simplemente a necesidades de eficiencia, vital en este caso. Una vez que se tienen estos dos subconjuntos, se realizará una ordenación de sus elementos en función de la consulta, es decir, una ordenación totalmente adaptada a la misma.

Con este escalonamiento en la búsqueda de componentes hemos conseguido la convergencia entre las dos tendencias más destacables en la organización de componentes: organización estática y dinámica. Las ordenaciones estáticas tienen como principal inconveniente su rigidez y la gran cantidad de información que suele ser necesario adjuntar a cada elemento para conseguir una eficacia aceptable. Las ordenaciones dinámicas, por otro lado, provocan la reorganización total de la base de datos o biblioteca de componentes en función de la consulta realizada, así que, aunque su precisión es mucho mayor, implican también una elevada ralentización en las tareas de búsqueda.

Al fraccionar la búsqueda en dos grandes pasos conseguimos la rapidez en la primera etapa, ventaja de los almacenamientos estáticos; y la elevada precisión en la segunda, ventaja de los almacenamientos dinámicos, ya que en esta segunda etapa sí se realiza una organización *ad-hoc* en función de la consulta.

13.3 Primera fase: búsqueda aproximada

En esta primera fase de localización, se aprovecha la ordenación estática de la base de datos para recuperar, de una forma rápida, un conjunto de componentes que son funcionalmente próximos a la consulta. Esta proximidad funcional se ha evaluado en función de su parecido estructural y semántico.

13.3.1 Recuperación estructural

Los criterios de ordenación entre componentes que priman la similitud estructural son NE y NE^{∞} que, básicamente, se diferencian en que mientras que el primero no contempla la existencia de bucles de evolución, el segundo sí lo hace. Como consecuencia de esto último, será interesante aplicar un criterio u otro dependiendo de si la consulta contempla, a su vez, bucles de evolución. Así, en caso de que la consulta presente bucles, la recuperación estructural se realizará atendiendo al criterio NE^{∞} y, en otro caso, atendiendo al criterio NE.

Esta decisión de optar por uno u otro criterio dependiendo de las características de la consulta viene motivada por dos razones:

- Si la consulta presenta algún bucle de comportamiento, el criterio NE^{∞} ofrece una ordenación que permitirá recuperar componentes más próximos estructuralmente al dado. La relación NE, en este caso, no ofrece resultados más precisos y, así, ahorrando búsquedas, se logra mejorar la eficiencia de la localización sin penalizar los resultados.

- Si la consulta no presenta ningún bucle de comportamiento, el criterio NE^{∞} no ofrece mejor información que el NE y, dado que la comparación de componentes según NE^{∞} es más compleja, estamos ahorrando también recursos sin que repercuta en los resultados.

Para recuperar aquellos componentes más próximos funcionalmente, se recurrirá a localizar la posible posición de la consulta en la base de datos y devolver aquellos componentes sucesores, antecesores y equivalentes de la consulta según el criterio elegido —NE ó NE^{∞}.

13.3.2 Recuperación semántica

TC y TC^{∞} son las dos relaciones de orden definidas que priman la similitud semántica entre componentes, la diferencia entre ellas es que, mientras la primera no contempla la existencia de bucles de funcionalidad, la segunda sí lo hace. Así que, al igual que en el caso de la recuperación estructural, se aplicará un criterio de localización u otro dependiendo de si la consulta contempla bucles de funcionalidad o no. Las justificaciones para esta elección son directamente extrapolables

de las dadas para la recuperación estructural (ver apartado 13.3.1). Así, en caso de que la consulta Q presente bucles en alguna de sus evoluciones posibles, recurriremos a una recuperación basada en el criterio TC^{∞}, y en caso contrario, la localización se hará atendiendo al criterio TC. En ambas circunstancias el algoritmo de recuperación devolverá aquellos componentes almacenados que sean antecesores, sucesores o equivalentes a la consulta.

13.4 Segunda fase: refinamiento en la búsqueda

Para esta segunda fase se parte de dos conjuntos diferentes de componentes: aquellos que son estructuralmente próximos a la consulta, según los criterios NE ó NE^{∞}, y aquellos que son semánticamente próximos, según los criterios TC ó TC^{∞}. Estos dos conjuntos son resultantes de una valoración aproximada de las similitudes funcionales y, en esta segunda fase, se refinará la búsqueda para obtener los más próximos según ambos criterios. Cada subconjunto tendrá un tratamiento de selección diferente, aunque guiado por una idea común: la minimización del esfuerzo de adaptación a la consulta.

13.4.1 Recuperación estructural

La recuperación estructural busca localizar aquellos componentes cuyo *esqueleto* de comportamiento sea lo más parecido posible al solicitado. A los componentes recuperados en la primera fase se les presupone una mayor proximidad estructural respecto a la consulta que al resto de los componentes almacenados en la base de datos, sin embargo el criterio utilizado para recuperarlos de la biblioteca sólo permite obtener relaciones de *contenido-continente*. Esto permite que, aún manteniendo la misma relación de contenido con la consulta, se hayan recuperado componentes con notables diferencias estructurales con la misma.

Es objetivo de esta segunda fase de localización refinar la búsqueda y dilucidar cuán diferente es cada componente recuperado de la consulta y, en consecuencia, decidir cuál o cuáles de ellos serán los componentes resultantes de todo el proceso de búsqueda.

En el capítulo 11 se habían propuesto dos distancias que cuantificaban estas diferencias estructurales entre componentes —*distancia del número de evoluciones totales* y *distancia del número de evoluciones totales no acotadas*. La decisión de adoptar una u otra se basa en la existencia o no de recursión en la consulta, en caso de que la consulta presente recursión se optará por cuantificar las diferencias estructurales utilizando la distancia del número de evoluciones no acotadas y, en caso contrario, se utilizará la distancia del número de evoluciones totales. Las justificaciones para obrar de una u otra forma son idénticas a las que se aportaron para seleccionar el criterio de observación en la primera fase (NE ó NE^{∞}) y pueden consultarse en el apartado 13.3.1.

Obviamente serán seleccionados aquellos componentes que presenten la menor distancia estructural con la consulta, ya que de esta manera estaremos minimizando los esfuerzos de adaptación.

13.4.2 Recuperación semántica

En esta segunda fase de la localización se parte del subconjunto de elementos próximos semánticamente a la consulta, resultado de la primera etapa. Ahora se variará su ordenación —antes en función de TC o bien de TC^{∞}— de forma que ésta esté totalmente adaptada a la consulta en curso, logrando así una mayor precisión.

El refinamiento en la búsqueda se hará atendiendo a diversos criterios, todos ellos basados en las cuantificaciones de diferencias semánticas detalladas en el apartado 11.4 —intersección, diferencia e inconsistencia funcionales. Partiendo de esta base se han definido otros criterios que permiten la cuantificación de la funcionalidad común con la consulta, el exceso y defecto de funcionalidad entre componentes y consulta, etc.

Consenso funcional

Dados dos componentes, C y C', se define la relación

$$\rho(C, C') = g \sqcap g',$$

donde g es el grafo MUS correspondiente a C, g' el grafo MUS de C' y donde $\sqcap$ sigue la definición 11.15. $\rho(C, C')$ expresa el conjunto de vías de evolución completas que tienen en común C y C'.

Según la situación que se nos plantea a nosotros, nos interesará buscar aquel componente C_i que maximice $\rho(Q, C_i)$, ya que éste será el que presenta mayor similitud funcional con la consulta propuesta.

Dados dos componentes C_1 y C_2 se considera que el primero está más próximo en *consenso funcional* a una consulta Q que el segundo, denotándose de la forma $\rho(Q, C_2) \subseteq \rho(Q, C_1)$, si se satisface la condición siguiente:

$$\rho(Q, C_2) \sqsubseteq_{\mathcal{O}} {}^{1}\rho(Q, C_1)$$

Déficit funcional

Dados dos componentes C y C', se define la relación

$$\delta(C, C') = g \ominus \rho(C, C') = g \ominus (g \sqcap g')$$

[1]La relación $\mathcal{O}$ será instanciada por la relación TC o bien TC^{∞} en función de si la consulta presenta recursiones o no.

donde g es el grafo MUS de C, g' el grafo MUS de C' y $\ominus$ sigue la definición 11.16. $\delta(C, C')$ expresa la funcionalidad que, estando contenida en C, falta por especificar en C'.

En nuestra situación $\delta(Q, C_i)$ expresará la funcionalidad requerida en Q y que falta en C_i, es decir, la carencia de funcionalidad de cada C_i respecto a lo solicitado.

Dados dos componentes C_1 y C_2, se considera que el primero está más próximo en *déficit funcional* a una consulta Q que el segundo, denotándose $\delta(Q, C_1) \subseteq \delta(Q, C_2)$, si se satisface la condición siguiente:

$$\delta(Q, C_1) \sqsubseteq_{\mathcal{O}} \delta(Q, C_2)$$

Exceso funcional

Dados dos componentes C y C', se define la relación

$$\varepsilon(C, C') = g' \ominus \rho(C, C') = g' \ominus (g \sqcap g')$$

donde g es el grafo MUS de C, g' el grafo MUS de C' y $\ominus$ sigue la definición 11.16. $\varepsilon(C, C')$ expresa la funcionalidad que, estando contenida en C' no lo está en C.

En nuestra situación $\varepsilon(Q, C_i)$ expresará aquella funcionalidad que, estando especificada en C_i, no se requiere en Q, es decir, aquella funcionalidad que *sobra*.

Dados dos componentes C_1 y C_2, se considera que el primero está más próximo en *exceso funcional* a una consulta Q que el segundo, denotándose $\varepsilon(Q, C_1) \subseteq \varepsilon(Q, C_2)$, si se satisface la condición siguiente:

$$\varepsilon(Q, C_1) \sqsubseteq_{\mathcal{O}} \varepsilon(Q, C_2)$$

Adaptación funcional

Dados dos componentes C y C', se define la relación

$$\Delta(C, C') = \delta(C, C') \cup \varepsilon(C, C')$$

$\Delta(C, C')$ expresa la funcionalidad que estando incluida en uno de los componentes no lo está en el otro, y viceversa.

En nuestro caso $\Delta(Q, C)$ cuantifica la funcionalidad a *eliminar* para que componente y consulta incrementen su parecido semántico.

Dados dos componentes C_1 y C_2, se considera que el primero está más próximo en adaptación funcional a una consulta Q que el segundo, denotándose $\Delta(Q, C_1) \subseteq \Delta(Q, C_2)$, si se satisface la condición siguiente:

$$\Delta(Q, C_1) \sqsubseteq_{\mathcal{O}} \Delta(Q, C_2)$$

Vector de ajuste funcional

Para que un componente C se adapte a una consulta Q no basta con que maximice el *consenso funcional* entre ambas, también es preciso que minimice el *vector de diferencia funcional*. El vector de ajuste funcional evalúa estos dos parámetros:

$$\Theta(Q, C) = (\rho(Q, C),\ \Delta(Q, C))$$

Dados dos componentes C_1 y C_2, se considera que el primero necesita menos ajustes para adaptarse a una consulta Q que el segundo, denotándose $\Theta(Q, C_1) \subseteq \Theta(Q, C_2)$, si se satisface la condición siguiente:

$$\begin{aligned}&(\Delta(Q, C_1) \subseteq \Delta(Q, C_2))\\ &\vee ((\Delta(Q, C_1) =_{\mathcal{O}} \Delta(Q, C_2)) \wedge (\rho(Q, C_2) \subseteq \rho(Q, C_1)))\end{aligned}$$

Representación gráfica de los resultados

Cualquiera de los criterios de ordenación propuestos —consenso funcional, déficit funcional, exceso funcional, adaptación funcional y vector de ajuste— permite una reorganización de un conjunto de componentes $\mathcal{C} = \{C_i\}_{i=1}^{m}$ en función de una consulta Q. Para cada criterio es posible obtener un grafo representativo del orden parcial de los elementos de $\mathcal{C}$.

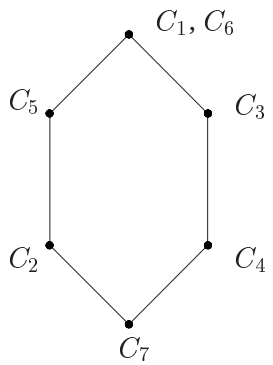


Figura 13.2. Diagrama Hasse de una relación de refinamiento ξ

Dada una relación de refinamiento cualquiera, ξ, y un conjunto de componentes $\mathcal{C}$, ésta se reduce a una relación TC ó TC^{∞} entre ellos, pudiendo realizarse una representación gráfica semejante a la realizada en el capítulo 10, y dado que, además, el conjunto de elementos con el que trabajamos es reducido podemos simplificarla aún más. Estas relaciones mantienen la propiedad reflexiva, con lo que podemos obviar la flechas de autorrelación; como además mantienen la propiedad transitiva, podemos omitir la flechas de relación entre componentes que ya

están relacionados por secuencias de flechas; además, manteniendo un protocolo de representación como el que sigue:

$(\xi(Q,C) \subseteq \xi(Q,C') \wedge (\xi(Q,C') \subseteq \xi(Q,C)) \Rightarrow C$ a la misma altura que C'
$(\xi(Q,C) \subseteq \xi(Q,C') \wedge (\xi(Q,C') \not\subseteq \xi(Q,C)) \Rightarrow C'$ se representa sobre C
$(\xi(Q,C) \not\subseteq \xi(Q,C') \wedge (\xi(Q,C') \subseteq \xi(Q,C)) \Rightarrow C$ se representa sobre C'
$(\xi(Q,C) \not\subseteq \xi(Q,C') \wedge (\xi(Q,C') \not\subseteq \xi(Q,C)) \Rightarrow C$ no tiene relación con C'

podemos obviar las flechas, ya que las relaciones de inclusión se expresarán en vertical, es decir, aquellos componentes contenidos en otros se presentarán en la parte inferior de la gráfica (ver figura 13.2). Esta representación gráfica se conoce con el nombre de *diagrama Hasse* de la relación (Liu, 1985).

Selección refinada de componentes

Llegados a este punto podemos recapitular y, en la tabla 13.1, enumerar brevemente el conjunto de las relaciones de refinamiento propuestas. Según esto, podemos concluir que la relación más exacta a la hora de cuantificar el esfuerzo de adaptación de un componente, teniendo en cuenta también la funcionalidad que ya satisface, es el *vector de ajuste funcional*.

Nombre	Notación	Fórmula
Consenso funcional	$\rho(C,C')$	$g \sqcap g'$
Déficit funcional	$\delta(C,C')$	$g \ominus (g \sqcap g')$
Exceso funcional	$\varepsilon(C,C')$	$g' \ominus (g \sqcap g')$
Adaptación funcional	$\Delta(C,C')$	$\delta(C,C') \cup \varepsilon(C,C')$
Vector de ajuste funcional	$\Theta(C,C')$	$(\rho(C,C'),\ \Delta(C,C'))$

Tabla 13.1. Criterios para la cuantificación de la funcionalidad común entre dos componentes C y C'

Así que se obtendrá el vector de ajuste funcional con la consulta Q para todos los componentes recuperados en la primera fase, y se ordenarán atendiendo a este criterio según un diagrama Hasse, seleccionando a todos aquellos que se encuentren en la parte inferior del grafo resultante. La existencia de más de un grafo en la parte inferior es consecuencia de trabajar con relaciones de refinamiento que definen un orden parcial, en este caso puede ocurrir que haya que tratar con componentes que no están relacionados según el vector de ajuste funcional. En este último caso y, para poder terminar la fase de selección o refinamiento de

la búsqueda, se definirá un último criterio que permita ordenar totalmente estos componentes:

Definición 13.1. *Dada una relación de refinamiento ξ de las recopiladas en la tabla 13.1 (salvo que ésta sea el* consenso funcional*), y dados dos componentes C_1 y C_2 tal que $\xi(Q, C_1)$ y $\xi(Q, C_2)$ no son comparables según la relación TC ó TC^{∞} (dependiendo de la recursividad en Q), se dice que el primero necesita un esfuerzo menor de adaptación a la consulta que el segundo si $\sharp(\xi(Q, C_1)) \leq \sharp(\xi(Q, C_2))$. Donde $\sharp$ es la función cardinal definida en 10.6, que calcula el número de vías de evolución completas. En el caso de la relación de refinamiento ρ, se dirá que el primero necesita un esfuerzo menor de adaptación a la consulta que el segundo si $\sharp(\rho(Q, C_2)) \leq \sharp(\rho(Q, C_1))$.*

13.5 Ejemplo de recuperación de componentes

Este apartado pretende clarificar el procedimiento de recuperación de componentes de una biblioteca utilizando para ello un ejemplo sencillo. Partiremos de una biblioteca de componentes conformada por los elementos de la figura 13.3 y de la consulta reflejada en la figura 13.4.

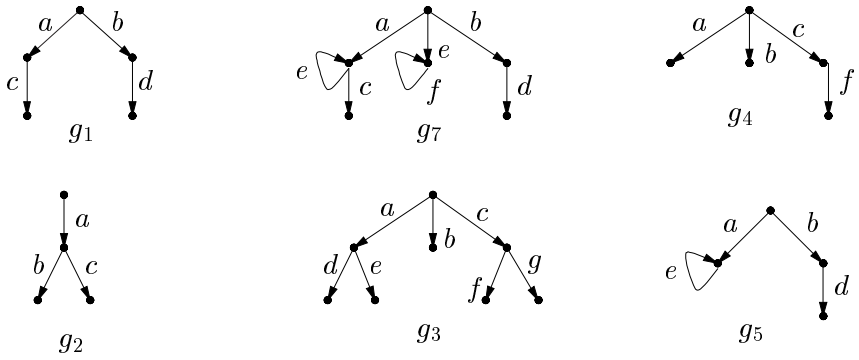


Figura 13.3. Conjunto de grafos $\mathcal{C}$ sobre los que realizar una búsqueda

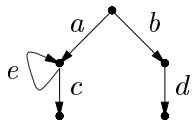


Figura 13.4. Consulta Q

Primera fase

Dado que la consulta Q presenta recursión, los criterios de localización que seguiremos serán NE^{∞} para la localización estructural y TC^{∞} para la localización semántica. Como los componentes de la biblioteca se mantienen ordenados según todos los criterios —NE, NE^{∞}, TC, y TC^{∞}—, sólo será necesario insertar la consulta Q en las estructuras reticulares formadas por los criterios escogidos (NE^{∞} y TC^{∞}) para efectuar la primera fase de la recuperación de componentes.

Así, como puede verse en la figura 13.5, los componentes antecesores, sucesores y equivalentes según NE^{∞} son los enmarcados en un cuadrado punteado (g_5, g_3 y g_7).

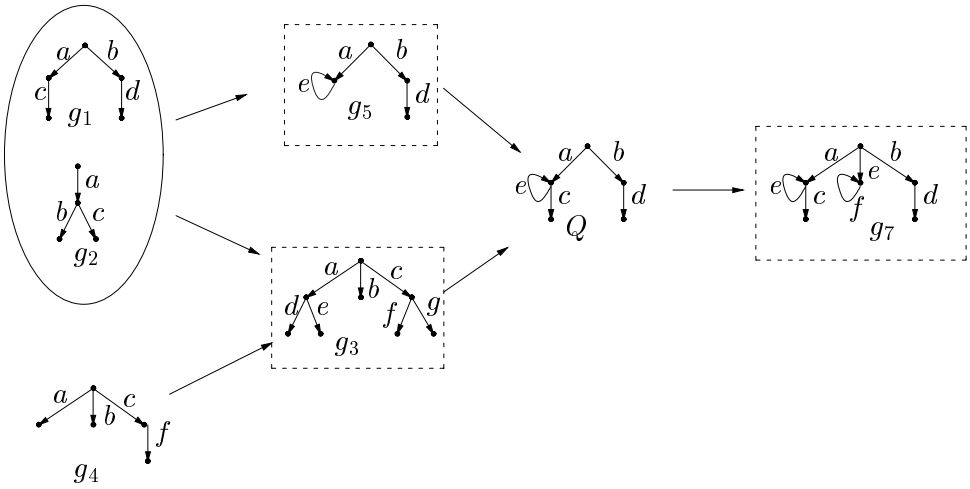


Figura 13.5. Primera fase de la localización estructural

En la figura 13.6, los componentes antecesores, sucesores y equivalentes según TC^{∞} son los enmarcados en un cuadrado punteado (g_1, g_5 y g_7).

Segunda fase

El refinamiento en la **búsqueda estructural** comienza a partir del conjunto de componentes resultante de la primera fase (g_5, g_3 y g_7). Como la consulta Q presenta recursiones en su comportamiento, se recurrirá a la *distancia del número de evoluciones totales no acotadas* para cuantificar cuál o cuáles son los componentes que necesitarán menores esfuerzos de adaptación a la consulta. En este

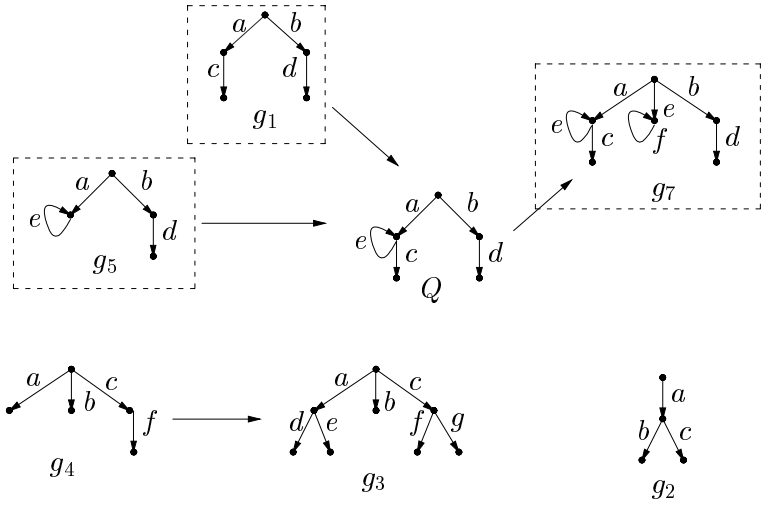


Figura 13.6. Primera fase de la localización semántica

caso se obtendrá entonces:

$$NE^{\infty}(Q) = (2, 2, 3, 2+, 3+)$$
$$NE^{\infty}(g_3) = (2, 2, 1, 2, 2)$$
$$NE^{\infty}(g_5) = (2, 2+)$$
$$NE^{\infty}(g_7) = (2, 2, 3, 2+, 2+, 3+)$$

de donde:

$$\vec{df}_{e\infty}(g_3, Q) = (1, 2, 2, 3, 1, 2)$$
$$\vec{df}_{e\infty}(g_5, Q) = (2, 3, 2)$$
$$\vec{df}_{e\infty}(g_7, Q) = (1)$$

Así que las distancias del número de evoluciones totales no acotadas serán las siguientes:

$$\mathrm{d}^{\infty}_{\mathrm{NE}}(g_3, Q) = \sqrt{23}$$
$$\mathrm{d}^{\infty}_{\mathrm{NE}}(g_5, Q) = \sqrt{15}$$
$$\mathrm{d}^{\infty}_{\mathrm{NE}}(g_7, Q) = 1$$

concluyendo que el componente C_7, que tiene como grafo MUS a g_7, es el componente resultante de esta localización estructural.

El refinamiento en la **búsqueda semántica** parte de la obtención de un conjunto de componentes TC^{∞}-próximos a la consulta: g_1, g_5 y g_7. Para calcular los vectores de ajuste funcional correspondientes a cada componente es necesario, primero obtener las funciones TC^{∞} aplicadas a cada uno de los grafos y a la consulta:

$\mathbf{g_i}$	$\mathbf{TC^{\infty}(g_i)}$
g_1	(ac, bd)
g_7	$(ac, aec, ae+, ae+c, ef+, bd)$
g_5	$(ae+, bd)$

$TC^{\infty}(Q) = (ae+, ac, aec, ae+c, bd)$

De forma que ahora ya es posible obtener los valores de $\rho(Q, C_i)$, de $\delta(Q, C_i)$, de $\varepsilon(Q, C_i)$, de $\Delta(Q, C_i)$ y de $\Theta(Q, C_i)$:

$\mathbf{g_i}$	$\rho(\mathbf{Q}, \mathbf{g_i})$
g_1	(ac, bd)
g_7	$(ac, aec, ae+, ae+c, bd)$
g_5	$(ae+, bd)$

$\mathbf{g_i}$	$\delta(\mathbf{Q}, \mathbf{g_i})$
g_1	$(ae+, aec, ae+c)$
g_7	$\emptyset$
g_5	$(ac, aec, ae+c)$

$\mathbf{g_i}$	$\varepsilon(\mathbf{Q}, \mathbf{g_i})$
g_1	$\emptyset$
g_7	$(ef+)$
g_5	$\emptyset$

$\mathbf{g_i}$	$\mathbf{\Delta(Q, g_i)}$
g_1	$(ae+, aec, ae+c)$
g_7	$(ef+)$
g_5	$(ac, aec, ae+c)$

$\mathbf{g_i}$	$\mathbf{\Theta(Q, g_i)}$
g_1	$((ac, bd), (ae+, aec, ae+c))$
g_7	$((ac, aec, ae+, ae+c, bd), (ef+))$
g_5	$((ae+, bd), (ac, aec, ae+c))$

Una vez obtenidos estos resultados sólo resta organizarlos en un grafo, siguiendo el criterio definido en 13.4.2 según su vector de ajuste funcional. En

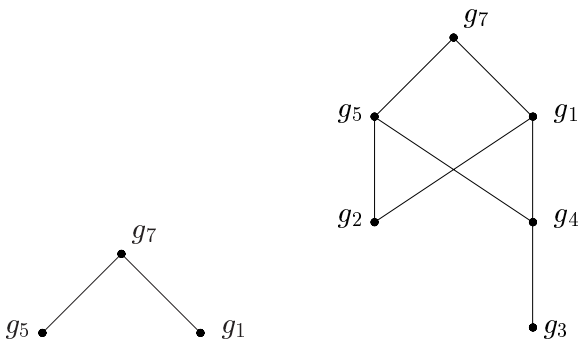
la figura 13.7(a) puede verse que ninguno de los tres componentes mantiene una relación de orden entre sus vectores de ajuste (definida según el criterio TC^{∞}), así que, teniendo en cuenta los cardinales de cada vector, en la figura 13.7(b) se representa la relación de orden final, de donde se deduce que el componente más apropiado es el que tiene como grafo a g_7.

(a) Orden parcial (b) Orden total

Figura 13.7. Diagramas de orden definidos por el vector de ajuste funcional

Comparación con una recuperación en una única fase

En la situación propuesta en este ejemplo, podemos ver como la ordenación dinámica totalmente adaptada a la consulta se hace sólo sobre un subconjunto de tres elementos de la biblioteca de componentes. De no haber realizado la primera fase de localización aproximada, nos habríamos encontrado con una labor de ordenación mucho más ardua, involucrando a todos los componentes de la base de datos, en este caso seis elementos.



(a) Orden según ρ (b) Orden según ρ

Figura 13.8. Grafos resultantes de la ordenación de componentes según ρ

Para el caso del criterio definido por el mayor consenso funcional con la consulta, nos encontraríamos ante la situación descrita en la figura 13.8(b) si no se hubiera realizado la primera fase de búsqueda. Sin embargo, tras realizar el primer filtrado, sólo restaría ordenar tres componentes, en lugar de los seis iniciales,

con el resultado expresado en la figura 13.8(a). Así, con un ejemplo muy reducido, podemos ver la gran ventaja que supone la división de la recuperación en dos etapas, ya que hemos agilizado las tareas de búsqueda sin que ello suponga ningún detrimento en la calidad de los resultados.

CAPÍTULO 14

Adaptación de componentes reutilizables

A la hora de reutilizar componentes software existen dos tendencias claramente diferenciadas: la reutilización exacta del componente sin que sufra ningún tipo de modificación, que se suele conocer como reutilización de cajas negras*; o bien la reutilización de componentes susceptibles de ser modificados para que satisfagan los requisitos pedidos, que se suele conocer como reutilización de* cajas blancas*. En nuestro entorno de reutilización se van a proporcionar las herramientas precisas para poder realizar modificaciones sobre los componentes recuperados de la base de datos, es decir su adaptación a los requisitos solicitados.*

14.1 Selección del componente a reutilizar

Tras recibir el usuario aquellos componentes más próximos, estructuralmente y semánticamente, a la funcionalidad requerida será necesario seleccionar cuál de ellos será adaptado para poder reutilizarse en nuestro entorno. Esta selección se ha dejado en manos del usuario quien, observando las características diferenciadoras de cada uno con la consulta, escogerá el que le parezca más apropiado.

La decisión de confiar en el usuario para seleccionar entre el componente más próximo estructuralmente y aquel más cercano semánticamente se ha tomado tras sopesar el coste computacional de adaptación de funcionalidad de un componente próximo estructuralmente. Los valores de distancia estructural definidos sólo cuantifican cuán costoso puede ser conseguir una estructura equivalente a la deseada, pero no tienen en cuenta que, además, será preciso adaptar semánticamente dicha estructura a la consulta. Realmente esta adaptación estructural sólo compensa a la semántica si el componente recuperado es estructuralmente equivalente al

pedido y, además, puede realizarse una substitución automática entre eventos. Esta decisión, que puede llegar a ser computacionalmente costosa, sería realizada por el usuario de forma cuasiautomática sin más que cotejar ambos componentes.

En este capítulo sólo se tratarán las peculiaridades de la adaptación semántica (apartado 14.2) ya que la adaptación estructural puede verse como una combinación de labores de substitución de eventos y adaptación semántica posterior.

14.2 Adaptación semántica

Tras seleccionar el componente C que se desea adaptar para que satisfaga la funcionalidad solicitada en la consulta Q podrá procederse a dicho proceso de adaptación. Antes de detallar este proceso, se realiza una recapitulación de las principales funciones de cuantificación de diferencias funcionales en la tabla 14.1.

Nombre	Notación	Fórmula
Consenso funcional	$\rho(Q,C)$	$g_Q \sqcap g_C$
Déficit funcional	$\delta(Q,C)$	$g_Q \ominus (g_Q \sqcap g_C)$
Exceso funcional	$\varepsilon(Q,C)$	$g_C \ominus (g_Q \sqcap g_C)$
Adaptación funcional	$\Delta(Q,C)$	$\delta(Q,C) \cup \varepsilon(Q,C)$
Vector de ajuste funcional	$\Theta(Q,C)$	$(\rho(Q,C),\ \Delta(Q,C))$

Tabla 14.1. Criterios para la cuantificación de la funcionalidad común entre una consulta Q y un componente C

Así pues, el proceso de adaptación consta básicamente de dos fases: una primera donde se especifica nueva funcionalidad, aquella dada por el valor de $\delta(Q,C)$, es decir, las trazas de evolución que se desea que tenga Q y que C no tiene especificadas; y una segunda donde se elimina la funcionalidad especificada en C y que no interesa que permanezca en Q, $\varepsilon(Q,C)$. El orden en este proceso es importante ya que de esta forma se evita la supresión de estados y transiciones pertenecientes a las trazas a eliminar que podrían ser requeridas para la inclusión de nuevas vías de evolución.

Definición 14.1. *Un grafo $g \in \mathbb{G}$ es determinista si se satisface que dada una secuencia de eventos posibles $a_0 a_1 \ldots a_n \ldots \in \mathbb{T}^\infty$ especificada en dicho grafo, existe una única vía de evolución $\pi \in V(g)$ tal que*

$$\pi : E_0 \xrightarrow{a_0} E_1 \xrightarrow{a_1} \ldots \xrightarrow{a_n} E_{n+1} \ldots$$

Trabajar con grafos deterministas (definición 14.1) facilita estas labores de adaptación ya que al localizar una traza de evolución susceptible de ser eliminada sólo es posible la existencia de un camino que la contenga, y, de la misma forma, cuando se desea especificar una nueva traza es sencillo localizar la vía de evolución donde se incrementará la especificación.

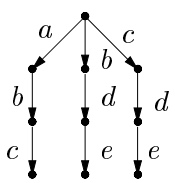
14.2.1 Ampliación de funcionalidad

El pseudocódigo del algoritmo de ampliación de funcionalidad se ha detallado en el algoritmo 14.1. Este algoritmo recursivo recibe como parámetros el nodo sobre el que se añade una nueva transición, en la primera ejecución será el nodo inicial, y la secuencia de eventos que conforma la traza a añadir. El mecanismo es sencillo ya que simplemente es necesario detectar, para cada nodo, si ya existe la transición especificada en la nueva traza: en este caso sólo será preciso seguir avanzando en el grafo a través de dicha subtraza; en caso contrario, será necesario añadir una nueva transición hacia un nuevo estado, totalmente subespecificado sobre el que se trabajará en la iteración posterior.

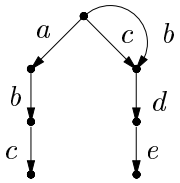
Algoritmo 14.1 Añadir una traza a partir de un nodo: añade_traza(traza, nodo_actual)

– Eliminar el primer evento de la traza, y almacenarlo en la variable evento_actual.
– Crear una lista con todas las transiciones posibles desde el nodo_actual y almacenarla en la variable transiciones_salientes.
si (evento_actual esta incluido en la lista transiciones_salientes) **entonces**
  – Llamada recursiva a esta misma función pasándole como parámetros el nodo destino de la transición etiquetada con evento_actual y con la traza modificada: añade_traza(traza, nodo_siguiente).
en otro caso
  – Crear un nuevo nodo, nuevo_nodo, y una nueva transición etiquetada con el evento_actual desde el nodo_actual hasta ese nuevo nodo.
  – Llamada recursiva a esta misma función pasándole como parámetros el nuevo nodo y la traza modificada: añade_traza(traza, nuevo_nodo).
fin de la condición

Es necesario notar que si la traza a añadir tiene una subtraza común con alguna de las trazas del grafo, que no sea una subtraza prefijo, no se reutilizan los estados ya existentes del grafo para añadir las nuevas transiciones (figura 14.1(b)), sino que se crean nuevos estados que permitan la secuencia de eventos deseada (figura 14.1(a)). Se ha optado por esta solución porque, si bien el grafo modificado tendrá un tamaño mayor, también es cierto que satisfará la funcionalidad deseada con un menor coste computacional en el proceso de adaptación y, consecuentemente, con una mayor eficiencia temporal en el mismo. Realmente la eficiencia del modelo obtenido puede ser optimizada en la fase siguiente del ciclo de vida, fase de refinamientos sucesivos (figura 6.1) donde sí se persigue una optimización de la implementación, que no es vital en la fase de requisitos iniciales.



(a) Sin optimización de estados



(b) Con optimización de estados

Figura 14.1. Comparación entre un grafo optimizado y uno, funcionalmente equivalente al anterior, pero no optimizado

El caso de las trazas prefijo comunes es totalmente diferente, como los modelos con los que trabajamos son deterministas, no sería correcto tener en un mismo estado dos transiciones posibles provocadas por el mismo evento y que lleven a dos estados del modelo.

14.2.2 Reducción de funcionalidad

En el proceso de reducción de funcionalidad, eliminando trazas, ha de ser más cuidadoso que el proceso de incremento de la misma. Esto es debido a que cualquier traza de evolución puede tener transiciones y nodos comunes con otras trazas del grafo que no se desea que se vean afectadas. En el algoritmo 14.2 se muestra el pseudocódigo del algoritmo básico de eliminación de una traza de evolución de un grafo MUS.

14.3 Resultados de verificación

Una pregunta interesante tras el proceso de adaptación de funcionalidad es ¿qué información de verificación puedo reutilizar o mantener tras él? Teniendo en cuenta que es posible modificar todas las trazas, variando los estados de especificación de los eventos desde *falso* a *subespecificado* o desde *verdadero* a *subespecificado* y viceversa, y que es posible añadir o eliminar estados de evolución del sistema, no es posible determinar un patrón común que permita alguna conclusión a este respecto, con lo que la reutilización de información de verificación en esta situación no puede plantearse como una labor sistemática y documentada, sino como una labor puramente anecdótica propia de la pericia de cada usuario. Básicamente sería necesario analizar cada uno de los estados modificados de algún modo durante el proceso de adaptación, y actualizar los grados de satisfacción de cada una de las propiedades verificadas sobre el mismo para disponer de los resultados de verificación correctos.

Algoritmo 14.2 Eliminar una traza a partir de un nodo: eliminar_traza(traza, nodo_actual)

– Obtener una lista con el conjunto de todas las transiciones salientes del nodo_actual y almacenarla en una variable transiciones_salientes.
– Obtener una lista con el conjunto de todas las transiciones entrantes al nodo_actual y almacenarla en una variable transiciones_entrantes.
si (número de transiciones anteriores es mayor que uno) **entonces**
  – Fin de la iteración.
fin de la condición
si (número de transiciones posteriores es mayor que uno) **entonces**
  – Todas las eliminaciones que se hayan señalizado en iteraciones anteriores a la actual no son válidas. Se activa una variable estática para indicarlo y se almacena en otra variable estática el número de iteración actual.
fin de la condición
si (traza está vacía) **entonces**
  – Eliminar el nodo actual y todas las transiciones salientes desde él.
  – Fin de la iteración.
fin de la condición
si (el número de transiciones salientes es idéntico al de entrantes e igual a uno) **entonces**
  – Señalizar la eliminación de este nodo.
fin de la condición
– Extraer de la traza actual el primer evento.
– Seleccionar el nodo al que se accede a través de dicho evento.
– Llamada recursiva a esta misma función con este nuevo nodo y la traza modificada como parámetros: eliminar_traza(traza, nodo_siguiente).
si (no es posible realizar ninguna eliminación) **entonces**
  – Fin de la iteración.
fin de la condición
si (está señalizada la eliminación de este nodo) **entonces**
  – Eliminar el nodo actual y todas las transiciones salientes del mismo.
  – Fin de la iteración.
fin de la condición
– Eliminar la transición provocada por el evento actual.
– Fin de la iteración.

Este es el principal inconveniente que presentan los entornos de reutilización de *cajas blancas* frente a la reutilización sin modificaciones. En este último caso se mantiene la integridad del componente y, consecuentemente, sus resultados de verificación (consultar apartado 1.3.4).

14.4 Almacenamiento de modelos adaptados en la base de datos

Tras la adaptación de un componente, ¿es éste almacenado en la base de datos? Aunque de la figura 8.2 se extrae esta conclusión, podría llegarse a una sobrecarga de la base de datos con componentes muy semejantes que llevaría a una ralentización excesiva del proceso de localización de componentes, sin que la fiabilidad de las recuperaciones se vea compensada.

Lo ideal sería establecer un umbral de diferencia de funcionalidad que definiese cuándo se considera que la distancia funcional entre un componente y el obtenido tras la adaptación de éste, son lo suficientemente distintos como para almacenar este último en la biblioteca. Este parámetro no debería ser un valor estático, ya que debería depender del número de componentes almacenados en la base de datos en cada momento.

PARTE V

Reutilización de información de verificación

CAPÍTULO 15

Grados de satisfacción de requisitos SCTL sobre grafos MUS

En este capítulo se definen los posibles grados de satisfacción de una propiedad o requisito SCTL en un componente. Se extienden los grados de satisfacción de un requisito SCTL sobre un estado con el fin de obtener resultados de verificación a lo largo de todas las posibles vías de evolución del sistema. Para ello se identifican cuatro metapropiedades cuyo resultado de verificación sobre un grafo MUS será reutilizado cuando convenga. Para una correcta lectura de este capítulo es imprescindible conocer los resultados de verificación de un requisito sobre un estado cualquiera de un sistema, con este fin en el capítulo 9 se ha realizado un breve resumen del trabajo anteriormente realizado por (García-Duque, 2000).

15.1 Introducción y notación

A la hora de reutilizar la información de verificación de una propiedad o requisito SCTL en un grafo MUS, nos interesa la síntesis de toda la información de verificación obtenida en todos y cada uno de los estados del grafo, para que ésta pueda ser gestionada de una forma eficiente. Con este objetivo se han definido cuatro metapropiedades que definen el comportamiento global de un grafo o modelo ante un requisito o propiedad determinado. Una vez sintetizada toda la información de verificación interesante, almacenada en estas cuatro metapropiedades, resta aplicarla en aquellos casos donde sea conveniente.

Definición 15.1. *Dada una vía de evolución* $\pi \in V(g)$ *se define su* **traza de**

estados, *denotándose $E(\pi)$, al conjunto formado por todos aquellos estados $E(\pi) = E_0E_1 \dots E_n$ por los que transcurre el sistema cuando evoluciona según la traza π.*

Definición 15.2. *Dado un grafo MUS $g \in \mathbb{G}$, se define el conjunto de sus trazas de estados, $E(g)$, como:*

$$E(g) = \{E(\pi) \mid \pi \in V(g)\}$$

donde $V(g)$ es el conjunto de vías de evolución completas del grafo g (definición 10.5).

15.2 Definición de metapropiedades sobre un grafo MUS

Cuando se estudia el comportamiento de un sistema ante una propiedad funcional, suele ser interesante conocer si estamos ante una propiedad de viveza, o de seguridad. Siguiendo este criterio, es comúnmente aceptada la clasificación de propiedades en:

- **Finalidades**: expresan una condición de viveza, es decir, condición que satisface el sistema en, al menos, algún momento de su ejecución. Dentro de este grupo se subdividen en:
  - **Finalidades universales**: propiedades que el sistema satisface al menos una vez, independientemente de la secuencia de evolución que siga.
  - **Finalidades existenciales**: propiedades que el sistema satisface al menos una vez, en alguna de sus secuencias de evolución.

- **Invarianzas**: expresan una condición de seguridad, es decir, condición que satisface el sistema durante su ejecución. Dentro de estas propiedades podemos diferenciar entre:
  - **Invarianzas universales:** propiedades que el sistema satisface en todos y cada uno de sus estados, independientemente de la secuencia de evolución que siga.
  - **Invarianzas existenciales:** propiedades que el sistema satisface en todos y cada uno de los estados en, al menos, alguna secuencia de evolución posible.

Siendo coherentes con esta clasificación, hemos definido cuatro metapropiedades que, una vez instanciadas y conocidos sus grados de satisfacción en el grafo

MUS, nos permitirán una gestión eficiente de la información de verificación de la propiedad que es instanciada con vistas a su posterior reutilización. Es decir, además de almacenar información interesante, como el tipo de propiedad que es y el tipo de propiedad que podría llegar a ser, permite manejar su información de verificación sobre el grafo.

Definición 15.3. *Se define el metarrequisito* $\exists \Diamond\_\mathcal{R}$*, que será instanciado en cada caso por una propiedad o requisito* R_i*, como aquel que expresa que "*existe alguna secuencia de evolución del sistema donde finalmente —al menos en un estado— se satisface $\_\mathcal{R}$*".*

Definición 15.4. *Se define el metarrequisito* $\exists \Box\_\mathcal{R}$*, que será instanciado en cada caso por una propiedad o requisito* R_i*, como aquel que expresa que "*existe alguna secuencia de evolución del sistema donde invariantemente —en todos sus estados— se satisface $\_\mathcal{R}$*".*

Definición 15.5. *Se define el metarrequisito* $\forall \Diamond\_\mathcal{R}$*, que será instanciado en cada caso por una propiedad o requisito* R_i*, como aquel que expresa que "*en todas las secuencias de evolución del sistema, finalmente —al menos en un estado— se satisface $\_\mathcal{R}$*".*

Definición 15.6. *Se define el metarrequisito* $\forall \Box\_\mathcal{R}$*, que será instanciado en cada caso por una propiedad o requisito* R_i*, como aquel que expresa que "*en todas las secuencias de evolución del sistema, invariantemente —en todos sus estados— se satisface $\_\mathcal{R}$*".*

De esta forma, por cada propiedad verificada en el grafo, tendremos cuatro propiedades derivadas de las que necesitamos conocer sus correspondientes grados de satisfacción. Estos grados de satisfacción pertenecen al conjunto definido en 9.2, $\phi \in \Phi = \{0, \frac{1}{4}, \widehat{\frac{1}{2}}, \frac{1}{2}, \frac{3}{4}, 1\}$, con las operaciones descritas en las definiciones 9.3, 9.4 y 9.5. La semántica de estos grados de satisfacción es, además, idéntica a la expresada en el caso de los grados de satisfacción de una propiedad en un estado.

15.3 Obtención de los grados de satisfacción de una propiedad SCTL en una traza de estados

Definición 15.7. *Se define el grado de satisfacción del metarrequisito* $\Box\_\mathcal{R}$ *en una traza de estados del grafo* g*,* $E(\pi_i) \in E(g)$*, y se denota* $\vDash (\Box\_\mathcal{R}, E(\pi_i))$*, como el resultado de realizar la operación siguiente:*

$$\vDash (\Box\_\mathcal{R}, E(\pi_i)) \;=\; \vDash (\_\mathcal{R}, E_i^1) \wedge \vDash (\_\mathcal{R}, E_i^2) \wedge \ldots \wedge \vDash (\_\mathcal{R}, E_i^n)$$

siendo $E_i^j \in E(\pi_i),\ \forall j = 1, \ldots, n$ *y* $\wedge$ *la operación definida en 9.4.*

El grado de satisfacción de este metarrequisito indicará si la propiedad $\_\mathcal{R}$ es una invarianza en una de las vías de evolución del sistema, es decir, si se satisface en todos y cada uno de los estados de la misma.

Definición 15.8. *Se define el grado de satisfacción del metarrequisito* $\Diamond\_\mathcal{R}$ *en una traza del grafo* g*,* $E(\pi_i) \in E(g)$*, y se denota* $\vDash (\Diamond\_\mathcal{R}, E(\pi_i))$*, como el resultado de realizar la operación siguiente:*

$$\vDash (\Diamond\_\mathcal{R}, E(\pi_i)) \ = \ \vDash (\_\mathcal{R}, E_i^1) \vee \vDash (\_\mathcal{R}, E_i^2) \vee \ldots \vee \vDash (\_\mathcal{R}, E_i^n)$$

siendo $E_i^j \in E(\pi_i),\ \forall j = 1, \ldots, n$ *y* $\vee$ *la operación definida en 9.3.*

El grado de satisfacción de este metarrequisito indicará si la propiedad $\_\mathcal{R}$ es una finalidad en una de las vías de evolución del sistema, es decir, si se satisface al menos en alguno de los estados de la misma.

Definición 15.9. *El conjunto* $\Phi = \{0, \frac{1}{4}, \widehat{\frac{1}{2}}, \frac{1}{2}, \frac{3}{4}, 1\}$ *de los posibles grados de satisfacción es un conjunto parcialmente ordenado según la relación de orden "menor o igual" (*$\leq$*) definida en la figura 15.1.*

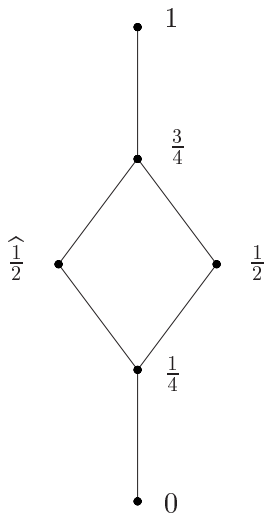


Figura 15.1. Diagrama Hasse de la relación del conjunto parcialmente ordenado $(\Phi, \leq)$

Debido a la definición anterior, que establece una relación de orden parcial entre grados de satisfacción de propiedades, y, debido a las definiciones de la operación lógica *or* ($\vee$) (definición 9.3) y *and* ($\wedge$) (definición 9.4), puede concluirse

que:

$$\vDash (\Diamond\_\mathcal{R}, E(\pi_i)) = \sup\{\vDash (\_\mathcal{R}, E_i^j)\} \qquad \forall E_i^j \in E(\pi_i)$$
$$\vDash (\Box\_\mathcal{R}, E(\pi_i)) = \inf\{\vDash (\_\mathcal{R}, E_i^j)\} \qquad \forall E_i^j \in E(\pi_i)$$

15.3.1 Propiedades de los grados de satisfacción sobre una traza

Propiedad 15.1. *Dada una traza de estados $E(\pi_i)$ y una propiedad R, se satisface que:*

$$\vDash (\Box R, E(\pi_i)) \;\leq\; \vDash (\Diamond R, E(\pi_i))$$

Demostración 15.1. *Dado que, por la definición 15.9 puede expresarse*

$$\vDash (\Diamond R, E(\pi_i)) = \sup\{\vDash (R, E_i^j)\} \qquad \forall E_i^j \in E(\pi_i)$$

y

$$\vDash (\Box R, E(\pi_i)) = \inf\{\vDash (R, E_i^j)\} \qquad \forall E_i^j \in E(\pi_i)$$

Entonces es trivial que

$$\sup\{\vDash (R, E_i^j)\} \geq \inf\{\vDash (R, E_i^j)\} \qquad \forall E_i^j \in E(\pi_i)$$
$$\Downarrow$$
$$\vDash (\Box R, E(\pi_i)) \;\leq\; \vDash (\Diamond R, E(\pi_i))$$

□

Propiedad 15.2. *Dada una propiedad R y una traza de estados $E(\pi_i)$, si se cumple que*

$$\vDash (\Box R, E(\pi_i)) \;=\; \vDash (\Diamond R, E(\pi_i)) \;=\; \phi \in \Phi$$

entonces se satisface que

$$\vDash (R, E_i^j) = \phi \qquad \forall E_i^j \in E(\pi_i)$$

Demostración 15.2.

$$\sup\{\vDash (R, E_i^j)\} = \inf\{\vDash (R, E_i^j)\} = \phi \in \Phi \qquad \forall E_i^j \in E(\pi_i)$$
$$\Downarrow$$
$$\vDash (R, E_i^j) = \phi \qquad \forall E_i^j \in E(\pi_i)$$

□

15.4 Obtención de los grados de satisfacción de una propiedad SCTL en un grafo MUS

Sea un grafo MUS $g \in \mathbb{G}$, de conjunto de trazas de estados $E(g) = \{E(\pi_i)\}_{i=1}^{m}$.

Definición 15.10. *Se define el grado de satisfacción del metarrequisito* $\exists \Diamond\_\mathcal{R}$ *en un grafo* g*, y se denota* $\vDash (\exists \Diamond\_\mathcal{R}, g)$*, como el resultado de realizar la operación siguiente:*

$$\vDash (\exists \Diamond\_\mathcal{R}, g) \;=\; \vDash (\Diamond\_\mathcal{R}, E(\pi_1)) \vee \vDash (\Diamond\_\mathcal{R}, E(\pi_2)) \vee \ldots \vee \vDash (\Diamond\_\mathcal{R}, E(\pi_m))$$

siendo $E(\pi_i) \in E(g),\ \forall i = 1, \ldots, m$ *y* $\vee$ *la operación definida en 9.3.*

El grado de satisfacción de este metarrequisito indicará si la propiedad $\_\mathcal{R}$ es una finalidad existencial en el grafo MUS.

Definición 15.11. *Se define el grado de satisfacción del metarrequisito* $\forall \Diamond\_\mathcal{R}$ *en un grafo* g*, y se denota* $\vDash (\forall \Diamond\_\mathcal{R}, g)$*, como el resultado de realizar la operación siguiente:*

$$\vDash (\forall \Diamond\_\mathcal{R}, g) \;=\; \vDash (\Diamond\_\mathcal{R}, E(\pi_1)) \wedge \vDash (\Diamond\_\mathcal{R}, E(\pi_2)) \wedge \ldots \wedge \vDash (\Diamond\_\mathcal{R}, E(\pi_m))$$

siendo $E(\pi_i) \in E(g),\ \forall i = 1, \ldots, m$ *y* $\wedge$ *la operación definida en 9.4.*

El grado de satisfacción de este metarrequisito indicará si la propiedad $\_\mathcal{R}$ es una finalidad universal en el grafo MUS.

Definición 15.12. *Se define el grado de satisfacción del metarrequisito* $\exists \Box\_\mathcal{R}$ *en un grafo* g*, y se denota* $\vDash (\exists \Box\_\mathcal{R}, g)$*, como el resultado de realizar la operación siguiente:*

$$\vDash (\exists \Box\_\mathcal{R}, g) \;=\; \vDash (\Box\_\mathcal{R}, E(\pi_1)) \vee \vDash (\Box\_\mathcal{R}, E(\pi_2)) \vee \ldots \vee \vDash (\Box\_\mathcal{R}, E(\pi_m))$$

siendo $E(\pi_i) \in E(g),\ \forall i = 1, \ldots, m$ *y* $\vee$ *la operación definida en 9.3.*

El grado de satisfacción de este metarrequisito indicará si la propiedad $\_\mathcal{R}$ es una invarianza existencial en el grafo MUS.

Definición 15.13. *Se define el grado de satisfacción del metarrequisito* $\forall \Box\_\mathcal{R}$ *en un grafo* g*, y se denota* $\vDash (\forall \Box\_\mathcal{R}, g)$*, como el resultado de realizar la operación siguiente:*

$$\vDash (\forall \Box\_\mathcal{R}, g) \;=\; \vDash (\Box\_\mathcal{R}, E(\pi_1)) \wedge \vDash (\Box\_\mathcal{R}, E(\pi_2)) \wedge \ldots \wedge \vDash (\Box\_\mathcal{R}, E(\pi_m))$$

siendo $E(\pi_i) \in E(g),\ \forall i = 1, \ldots, m$ *y* $\wedge$ *la operación definida en 9.4.*

El grado de satisfacción de este metarrequisito indicará si la propiedad $\_\mathcal{R}$ es una invarianza universal en el grafo MUS.

Definición 15.14. *Se define el* **grado de satisfacción de un requisito** R **en un grafo** g *y se denota* $\vDash (R, g)$ *como la cuádrupla* $(\phi_1, \phi_2, \phi_3, \phi_4)$*, donde* $\phi_i \in \Phi \quad \forall i = 1, \ldots, 4$*. Los valores de la cuádrupla se corresponden con:*

- $\phi_1 = \vDash (\exists \Diamond R, g)$ *se corresponde con el grado de satisfacción del metarrequisito* finalidad existencial *definido en 15.10*
- $\phi_2 = \vDash (\forall \Diamond R, g)$ *se corresponde con el grado de satisfacción del metarrequisito* finalidad universal *definido en 15.11*
- $\phi_3 = \vDash (\exists \Box R, g)$ *se corresponde con el grado de satisfacción del metarrequisito* invarianza existencial *definido en 15.12*
- $\phi_4 = \vDash (\forall \Box R, g)$ *se corresponde con el grado de satisfacción del metarrequisito* invarianza universal *definido en 15.13*

15.4.1 Propiedades de los grados de satisfacción de una propiedad sobre un grafo MUS

Propiedad 15.3. *Dado un grafo MUS* g *y una propiedad* R*, se satisface que*

$$\vDash (\forall \Diamond R, g) \leq \vDash (\exists \Diamond R, g)$$

Demostración 15.3. *La expresión* $\vDash (\exists \Diamond R, g)$ *puede calcularse como*

$$\vDash (\exists \Diamond R, g) = \sup\{\vDash (\Diamond R, E(\pi_i))\} \qquad \forall\, E(\pi_i) \in E(g)$$

y, análogamente, la expresión $\vDash (\forall \Diamond R, g)$ *como*

$$\vDash (\forall \Diamond R, g) = \inf\{\vDash (\Diamond R, E(\pi_i))\} \qquad \forall\, E(\pi_i) \in E(g)$$

De donde

$$\begin{gathered} \inf\{\vDash (\Diamond R, E(\pi_i))\} \leq \sup\{\vDash (\Diamond R, E(\pi_i))\} \\ \Downarrow \\ \vDash (\forall \Diamond R, g) \leq \vDash (\exists \Diamond R, g) \end{gathered}$$

□

Propiedad 15.4. *Dado un grafo MUS g y una propiedad R, si se satisface que*

$$\vDash (\forall \Diamond R, g) \;\; = \;\; \vDash (\exists \Diamond R, g) = \phi \in \Phi$$

entonces se cumple que

$$\vDash (\Diamond R, E(\pi_i)) = \phi \qquad \forall E(\pi_i) \in E(g)$$

Demostración 15.4. *Dado que podemos expresar* $\vDash (\forall \Diamond R, g)$ *como:*

$$\vDash (\forall \Diamond R, g) = \inf\{\vDash (\Diamond R, E(\pi_i))\} \qquad \forall\, E(\pi_i) \in E(g)$$

y siendo la expresión $\vDash (\exists \Diamond R, g)$ *equivalente a:*

$$\vDash (\exists \Diamond R, g) = \sup\{\vDash (\Diamond R, E(\pi_i))\} \qquad \forall\, E(\pi_i) \in E(g)$$

Entonces es inmediato que

$$\inf\{\vDash (\Diamond R, E(\pi))\} = \sup\{\vDash (\Diamond R, E(\pi))\} = \phi \in \Phi$$

satisfaciéndose claramente que

$$\vDash (\Diamond R, E(\pi_i)) = \phi \qquad \forall E(\pi_i) \in E(g)$$

□

Propiedad 15.5. *Dado un grafo MUS g y una propiedad R, se cumple que*

$$\vDash (\forall \Box R, g) \;\; \leq \;\; \vDash (\exists \Box R, g)$$

Demostración 15.5. $\vDash (\exists \Box R, g)$ *puede calcularse como*

$$\vDash (\exists \Box R, g) = \sup\{\vDash (\Box R, E(\pi_i))\} \qquad \forall\, E(\pi_i) \in E(g)$$

y la expresión $\vDash (\forall \Box R, g)$ *como*

$$\vDash (\forall \Box R, g) = \inf\{\vDash (\Box R, E(\pi_i))\} \qquad \forall\, E(\pi_i) \in E(g)$$

De donde

$$\begin{array}{c} \inf\{\vDash (\Box R, E(\pi_i))\} \leq \sup\{\vDash (\Box R, E(\pi_i))\} \\ \Downarrow \\ \vDash (\forall \Box R, g) \;\; \leq \;\; \vDash (\exists \Box R, g) \end{array}$$

□

Propiedad 15.6. *Dado un grafo MUS g y una propiedad R, si se cumple que*

$$\vDash (\exists\, \Box R, g) \;\; = \;\; \vDash (\forall\, \Box R, g) = \phi \in \Phi$$

entonces, también se satisface que

$$\vDash (\Box R, E(\pi_i)) = \phi \qquad \forall E(\pi_i) \in E(g)$$

Demostración 15.6. $\vDash (\exists\, \Box R, g)$ *puede expresarse como:*

$$\vDash (\exists\, \Box R, g) = \sup\{\vDash (\Box R, E(\pi_i))\} \qquad \forall\, E(\pi_i) \in E(g)$$

y la expresión $\vDash (\forall\, \Box R, g)$ *como*

$$\vDash (\forall\, \Box R, g) = \inf\{\vDash (\Box R, E(\pi_i))\} \qquad \forall\, E(\pi_i) \in E(g)$$

De donde

$$\inf\{\vDash (\Box R, E(\pi_i))\} = \sup\{\vDash (\Box R, E(\pi_i))\}\} = \phi \in \Phi$$

entonces claramente se satisface que

$$\vDash (\Box R, E(\pi_i)) = \phi \qquad \forall E(\pi_i) \in E(g)$$

□

Propiedad 15.7. *Dado un grafo MUS g y una propiedad R, siempre se satisface la condición siguiente:*

$$\vDash (\forall\, \Box R, g) \;\; \leq \;\; \vDash (\forall\, \Diamond R, g)$$

Demostración 15.7. $\vDash (\forall\, \Diamond R, g)$ *puede expresarse como:*

$$\vDash (\forall\, \Diamond R, g) = \inf\{\vDash (\Diamond R, E(\pi_i))\} = \inf\{\sup\{\vDash (R, E_i^j)\}\}$$
$$\forall\, E(\pi_i) \in E(g),\ \forall\, E_i^j \in E(\pi_i)$$

análogamente, $\vDash (\forall\, \Box R, g)$ *puede obtenerse como:*

$$\vDash (\forall\, \Box R, g) = \inf\{\vDash (\Box R, E(\pi_i))\} = \inf\{\inf\{\vDash (R, E_i^j)\}\}$$
$$\forall\, E(\pi_i) \in E(g),\ \forall\, E_i^j \in E(\pi_i)$$

De donde es inmediato ver que:

$$\inf\{\inf\{\vDash (R, E_i^j)\}\} \leq \inf\{\sup\{\vDash (R, E_i^j)\}\}$$
$$\Downarrow$$
$$\vDash (\forall\, \Box R, g) \;\; \leq \;\; \vDash (\forall\, \Diamond R, g)$$

□

Propiedad 15.8. *Dado un grafo MUS g y una propiedad R, siempre se satisface la condición siguiente:*

$$\vDash (\exists\, \Box R, g) \;\; \leq \;\; \vDash (\exists\, \Diamond R, g)$$

Demostración 15.8. $\vDash (\exists\, \Box R, g)$ *puede expresarse como:*

$$\vDash (\exists\, \Box R, g) = \sup\{\vDash (\Diamond R, E(\pi_i))\} = \sup\{\inf\{\vDash (R, E_i^j)\}\}$$
$$\forall\, E(\pi_i) \in E(g),\ \forall\, E_i^j \in E(\pi_i)$$

análogamente, $\vDash (\exists\, \Diamond R, g)$ *puede obtenerse como:*

$$\vDash (\exists\, \Diamond R, g) = \sup\{\vDash (\Diamond R, E(\pi_i))\} = \sup\{\sup\{\vDash (R, E_i^j)\}\}$$
$$\forall\, E(\pi_i) \in E(g),\ \forall\, E_i^j \in E(\pi_i)$$

De donde es inmediato ver que:

$$\sup\{\inf\{\vDash (R, E_i^j)\}\} \leq \sup\{\sup\{\vDash (R, E_i^j)\}\}$$
$$\Downarrow$$
$$\vDash (\exists\, \Box R, g) \;\; \leq \;\; \vDash (\exists\, \Diamond R, g)$$

□

Propiedad 15.9. *Dado un grafo MUS g y una propiedad R, las metapropiedades* $\vDash (\forall\, \Diamond R, g)$ *y* $\vDash (\exists\, \Box R, g)$ *no son comparables según la relación de orden definida en 15.9. Es decir,* $\vDash (\forall\, \Diamond R, g) \;\; \not\leq \;\; \vDash (\exists\, \Box R, g)$ *y* $\vDash (\exists\, \Box R, g) \;\; \not\leq \;\; \vDash (\forall\, \Diamond R, g)$

Demostración 15.9. $\vDash (\forall\, \Diamond R, g)$ *puede expresarse como*

$$\vDash (\forall\, \Diamond R, g) = \inf\{\vDash (\Diamond R, E(\pi_i))\} = \inf\{\sup\{\vDash (R, E_i^j)\}\}$$
$$\forall\, E(\pi_i) \in E(g),\ \forall\, E_i^j \in E(\pi_i)$$

y, análogamente, la expresión $\vDash (\exists\, \Box R, g)$*:*

$$\vDash (\exists\, \Box R, g) = \sup\{\vDash (\Diamond R, E(\pi_i))\} = \sup\{\inf\{\vDash (R, E_i^j)\}\}$$
$$\forall\, E(\pi_i) \in E(g),\ \forall\, E_i^j \in E(\pi_i)$$

De donde se deduce que:

$$\sup\{\inf\{\vDash (R, E_i^j)\}\} \not\leq \inf\{\sup\{\vDash (R, E_i^j)\}\}$$
$$\inf\{\sup\{\vDash (R, E_i^j)\}\} \not\leq \sup\{\inf\{\vDash (R, E_i^j)\}\}$$

□

15.4.2 Ordenación parcial de resultados de verificación

Las propiedades enumeradas en el apartado anterior permiten establecer una relación de orden parcial entre los cuatro resultados de verificar una propiedad sobre un grafo MUS. En la figura 15.2 puede verse el diagrama Hasse de dicha relación.

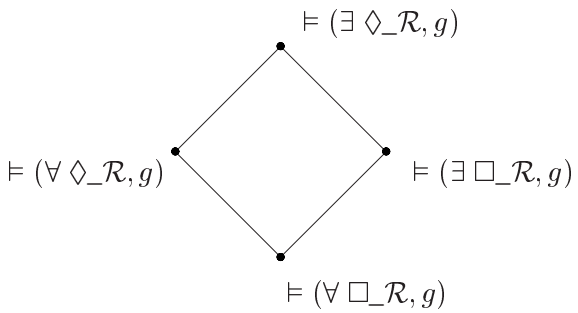


Figura 15.2. Relación de orden parcial entre los grados de satisfacción definidos

Así tenemos que el grado de satisfacción del metarrequisito $\exists\Diamond\_\mathcal{R}$ será siempre mayor o igual que el grado de satisfacción del metarrequisito $\forall\Diamond\_\mathcal{R}$ (propiedad 15.3) y éste, a su vez, mayor o igual que el obtenido por el metarrequisito $\forall\Box\_\mathcal{R}$ (propiedad 15.7). De la misma forma, el grado de satisfacción del metarrequisito $\exists\Diamond\_\mathcal{R}$ será siempre mayor o igual que el grado de satisfacción del metarrequisito $\exists\Box\_\mathcal{R}$ (propiedad 15.8), y éste, a su vez, mayor o igual que el obtenido por el metarrequisito $\forall\Box\_\mathcal{R}$ (propiedad 15.5). Esta relación define un orden parcial, ya que los grados de satisfacción de los metarrequisitos $\forall\Diamond\_\mathcal{R}$ y $\exists\Box\_\mathcal{R}$ no pueden ser ordenados (propiedad 15.9).

15.5 Ejemplo de obtención de los grados de satisfacción de propiedades SCTL sobre un grafo MUS

EJEMPLO 15.1. Partimos del grafo MUS de la figura 15.3, cuyos resultados de verificación, en todos y cada uno de sus estados, de las propiedades $R_1 \equiv (a \Rightarrow b)$ y $R_2 \equiv (c \Rightarrow a)$ se indican en la tabla siguiente:

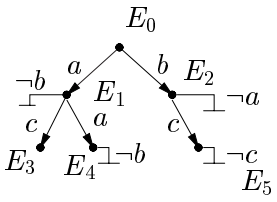


Figura 15.3. Grafo MUS

$\models (R_1, E_j)$	a	b
$\models (R_1, E_0) = 1$	1	1
$\models (R_1, E_1) = 0$	1	0
$\models (R_1, E_2) = \widehat{\frac{1}{2}}$	0	$\frac{1}{2}$
$\models (R_1, E_3) = \frac{1}{2}$	$\frac{1}{2}$	$\frac{1}{2}$
$\models (R_1, E_4) = \frac{1}{4}$	$\frac{1}{2}$	0
$\models (R_1, E_5) = \frac{1}{2}$	$\frac{1}{2}$	$\frac{1}{2}$

$\models (R_2, E_j)$	c	a
$\models (R_2, E_0) = \frac{3}{4}$	$\frac{1}{2}$	1
$\models (R_2, E_1) = 1$	1	1
$\models (R_2, E_2) = 0$	1	0
$\models (R_2, E_3) = \frac{1}{2}$	$\frac{1}{2}$	$\frac{1}{2}$
$\models (R_2, E_4) = \frac{1}{2}$	$\frac{1}{2}$	$\frac{1}{2}$
$\models (R_2, E_5) = \widehat{\frac{1}{2}}$	0	$\frac{1}{2}$

Nos interesa obtener los grados de satisfacción de ambas propiedades $R_1 \equiv (a \Rightarrow b)$ y $R_2 \equiv (c \Rightarrow a)$ en la totalidad del grafo. Para ello es imprescindible obtener primero las trazas de estados del grafo:

$$E(\pi_1) = E_0 E_1 E_3$$
$$E(\pi_2) = E_0 E_1 E_4$$
$$E(\pi_3) = E_0 E_2 E_5$$

De la definición 15.7 se deducen los siguientes valores, para las distintas trazas y para ambos requisitos:

$\_\mathcal{R}$	$\models (\Box\_\mathcal{R}, E(\pi_1))$	$\models (\Box\_\mathcal{R}, E(\pi_2))$	$\models (\Box\_\mathcal{R}, E(\pi_3))$
R_1	0	0	$\frac{1}{4}$
R_2	$\frac{1}{2}$	$\frac{1}{2}$	0

y de la definición 15.8 se deducen los siguientes valores, para las distintas trazas y ambos requisitos:

$\_\mathcal{R}$	$\models (\Diamond\_\mathcal{R}, E(\pi_1))$	$\models (\Diamond\_\mathcal{R}, E(\pi_2))$	$\models (\Diamond\_\mathcal{R}, E(\pi_3))$
R_1	1	1	1
R_2	1	1	$\frac{3}{4}$

Aplicando la definición 15.10 se obtienen los valores:

$$\models (\exists\ \Diamond R_1, g) = 1$$
$$\models (\exists\ \Diamond R_2, g) = 1$$

De lo que se deduce que, al menos en alguna de las posibles vías de evolución del grafo se satisfacen ambas propiedades, es decir, ambas son finalidades existenciales sobre el grafo g.

Aplicando la definición 15.11 se obtienen los valores:

$$\models (\forall\ \Diamond R_1, g) = 1$$
$$\models (\forall\ \Diamond R_2, g) = \frac{3}{4}$$

De lo que se deduce que la propiedad R_1 se satisface al menos en un estado en todas y cada una de las posibles trazas de evolución del grafo, es decir, es una finalidad universal sobre g. Sin embargo, la propiedad R_2 en el estado actual de especificación del grafo, no es una finalidad universal, pero su grado de satisfacción, $\frac{3}{4}$, permite que, añadiendo especificación al grafo g, sí pueda llegar a ser una finalidad universal sobre el mismo.

Aplicando la definición 15.12 se obtienen los valores:

$$\models (\exists\ \Box R_1, g) = \frac{1}{4}$$
$$\models (\exists\ \Box R_2, g) = \frac{1}{2}$$

De lo que se deduce que ni la propiedad R_1 ni la R_2 se satisfacen en todos y cada uno de los estados de alguna de las posibles trazas de evolución del sistema en su

estado actual. Además, independientemente de la especificación que se añada al grafo, R_1 nunca podrá ser una invarianza existencial sobre g, sin embargo R_2 sí podrá llegar a ser una invarianza existencial.

Aplicando la definición 15.13 se obtienen los valores:

$$\vDash (\forall\, \Box R_1, g) = 0$$
$$\vDash (\forall\, \Box R_2, g) = 0$$

De lo que se deduce que ni la propiedad R_1 ni la R_2 se satisfacen en todos los estados del sistema, no son invarianzas universales sobre g; además, independientemente de la evolución que siga g, ni R_1 ni R_2 podrán satisfacerse nunca en todos los estados.

Con los resultados almacenados en las dos tablas anteriores, ya se puede obtener la cuádrupla $\vDash (R_1, g) = (1, 1, \frac{1}{4}, 0)$ y $\vDash (R_2, g) = (1, \frac{3}{4}, \frac{1}{2}, 0)$. □

15.6 Conclusiones

Partiendo de la definición de los grados de satisfacción de una propiedad en un estado de un grafo subespecificado MUS, se ha aumentado la capacidad expresiva de los mismos de forma que se permita el almacenamiento de información de verificación de una propiedad sobre el grafo MUS completo.

Con dicho objetivo se han definido cuatro metarrequisitos o metapropiedades que, tras ser instanciados con un requisito concreto y evaluados sus grados de satisfacción, permiten conocer hasta qué punto se satisface dicho requisito sobre las trazas de evolución del grafo: si se satisface al menos en un estado o en todos los de una traza; o bien si se satisface al menos en una traza; o si se satisface en cualquiera de las vías de evolución posibles, etc. Además dichos grados de satisfacción permiten almacenar la potencial satisfacción del requisito instanciador en iteraciones posteriores del proceso de diseño.

En el capítulo 16 se detalla cómo se almacena, se recupera y se reutiliza dicha información para reducir, en la medida de lo posible, la carga computacional de las tareas de verificación.

Apéndices del capítulo

15.A Algoritmo de síntesis de resultados en las trazas

El algoritmo recursivo, representado en pseudocódigo en el algoritmo 15.1, indica cómo se realiza la síntesis de la información de verificación de una propiedad o requisito R sobre cada una de las trazas de un grafo MUS. El objetivo es conseguir los resultados $\Box\_\mathcal{R}$ (definido en 15.7) y $\Diamond\_\mathcal{R}$ (definido en 15.8) para cada una de las vías de evolución posible. En la figura 15.4 las líneas curvas indican el recorrido que siguen las iteraciones de este algoritmo para recorrer el grafo.

Algoritmo 15.1 Obtención de los resultados de verificación de un requisito (R) sobre las trazas de un grafo MUS (grafo): $\Box\_R$ y $\Diamond\_R$

– Obtener el resultado de verificación del requisito R en el nodo actual, almacenarlo en la variable resultado_nodo.
– Recuperar de la lista de resultados en las trazas de estados el último resultado almacenado, que será el que haya que actualizar con la contribución del estado actual. Almacenar dicho resultado en la variable resultado_traza.
– Actualizar el resultado de resultado_traza:
  resultado_traza. $\Diamond\_R$ = resultado_traza. $\Diamond\_R \vee$ resultado_nodo
  resultado_traza. $\Box\_R$ = resultado_traza. $\Box\_R \wedge$ resultado_nodo
– Almacenar el contenido de resultado_traza en la lista de resultados de las trazas de estados del grafo.
– Obtener el conjunto de nodos siguientes al nodo actual y almacenarlo en la variable nodos_siguientes.
si (no hay ningún nodo siguiente al actual) **entonces**
  – Fin de la iteración. {Se acaba una traza de evolución}
fin de la condición
para todo (nodo_auxiliar en la lista nodos_siguientes) **entonces**
  – Almacenar el último resultado de verificación, de la lista de resultados en las trazas de estados, en la variable ultimo_resultado.
  – Llamada recursiva a este mismo algoritmo pero ahora para actualizar los resultados de verificación del nodo nodo_auxiliar.
  si (hay algún nodo siguiente al actual) **entonces**
    – Almacenar el contenido de la variable ultimo_resultado en la lista de resultados de verificación de las trazas del grafo. {Sólo se almacena el resultado si hay alguna traza más}
  fin de la condición
fin de la iteración

La lista obtenida tras la ejecución del algoritmo 15.1 almacena tantas posiciones como trazas posibles haya en el grafo y, para cada una de ellas, los grados de satisfacción de los metarrequisitos $\Box\_R$ y $\Diamond\_R$. A la hora de extender dichos

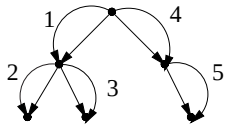


Figura 15.4. Orden de recorrido del grafo

resultados sobre todo el grafo MUS para poder obtener el valor $\vDash (R, g)$, definido en 15.14, se procederá tal y como se detalla en el pseudocódigo del algoritmo 15.2.

Algoritmo 15.2 Obtención del resultado de verificación de una propiedad R sobre un grafo MUS $\vDash (R, grafo)$

para todo (traza de estados del grafo) **entonces**
$\vDash (\forall\, \Box R, grafo) \;\; = \vDash (\forall\, \Box R, grafo) \wedge \vDash (\forall\, \Box R, traza).$
$\vDash (\exists\, \Box R, grafo) \;\; = \vDash (\exists\, \Box R, grafo) \vee \vDash (\exists\, \Box R, traza).$
$\vDash (\forall\, \Diamond R, grafo) \;\; = \vDash (\forall\, \Diamond R, grafo) \wedge \vDash (\forall\, \Diamond R, traza).$
$\vDash (\exists\, \Diamond R, grafo) \;\; = \vDash (\exists\, \Diamond R, grafo) \vee \vDash (\exists\, \Diamond R, traza).$
fin de la iteración

CAPÍTULO 16

Gestión de la información de verificación

Una vez definido qué información de verificación es relevante y cómo se va a almacenar asociada a cada componente (capítulo 15), resta detallar cómo gestionarla para hacer posible su reutilización. Esta gestión se divide fundamentalmente en tres grandes bloques: definición de cotas de resultados de verificación, obtenidas a partir de los componentes próximos funcionalmente al dado; recuperación de la información relevante almacenada en la base de datos; y clasificación y almacenamiento de la nueva información de verificación obtenida. El principal objetivo de este capítulo es desgranar estos bloques, ofreciendo una visión global de todo el proceso, y profundizar en la obtención de cotas de resultados de verificación.

16.1 Diferentes planteamientos

La situación de partida a la hora de reutilizar información de verificación se produce en el momento en el que es preciso verificar una propiedad SCTL sobre un modelo incompleto MUS (figura 8.4). La verificación formal de propiedades, basada en técnicas de *model checking* en nuestro entorno de desarrollo, presenta una carga computacional muy elevada que puede ser aliviada, en gran medida, por la reutilización de resultados de verificación. Es preciso destacar que la reutilización en este campo se presenta no como una alternativa a la verificación tradicional, sino como un complemento a ésta y a otras técnicas, como estrategias de reducción de estados y mejoras en la eficiencia de los propios algoritmos de verificación.

Nuestra propuesta se basa totalmente en las relaciones de proximidad funcional definidas en el capítulo 10, que permiten establecer relaciones de tipo

contenido-continente entre grafos MUS y también entre propiedades SCTL. Estas relaciones nos permitirán recuperar modelos incompletos que contengan o que estén contenidos en el dado y, sobre ellos, trabajaremos para localizar los resultados de verificación que nos interesen.

La reutilización de información de verificación puede plantearse en términos diferentes según la relación de proximidad existente entre los grafos y entre las propiedades cuyos resultados de verificación se reutilizarán:

– componentes próximos estructuralmente al dado, o bien

– componentes próximos semánticamente al dado,

y, en ambos casos:

– reutilizar resultados de verificación de una propiedad idéntica a la dada;

– reutilizar resultados de verificación de una propiedad próxima estructuralmente a la dada; o bien

– reutilizar resultados de verificación de una propiedad próxima semánticamente a la dada.

La opción más estricta, obviamente, es la que contempla sólo relaciones de parecido semántico entre grafos, relación TC^{∞}, y ésta será la que adoptaremos nosotros. Así que recuperaremos de la biblioteca de componentes aquellos que contengan o que estén contenidos por el grafo sobre el que deseamos obtener resultados de verificación. Respecto a los resultados de verificación susceptibles de ser reutilizados, centraremos nuestro campo de acción en un único frente: resultados de la misma propiedad sobre los grafos recuperados de la biblioteca. El proceso a seguir se detalla en el apartado 16.2.

Para posibilitar la reutilización de verificación de una propiedad en un grafo que contenga, según la relación de trazas completas no acotadas, al dado, o que esté contenido por él según esta misma relación, se hace imprescindible profundizar un poco más en algunos fundamentos teóricos, a los que se dedica el apartado 16.3.

16.2 Identificación de etapas

En la figura 16.1 se han esquematizado las fases necesarias para reutilizar resultados de verificación. El proceso comienza con la obtención de los patrones de búsqueda de la propiedad a verificar —básicamente TC^{∞}(propiedad)— y del grafo sobre el que se quiere obtener su grado de satisfacción —básicamente TC^{∞}(grafo). Con ambos patrones se recuperarán de la biblioteca aquellos componentes sucesores y antecesores del dado donde, además, se haya verificado previamente la propiedad deseada.

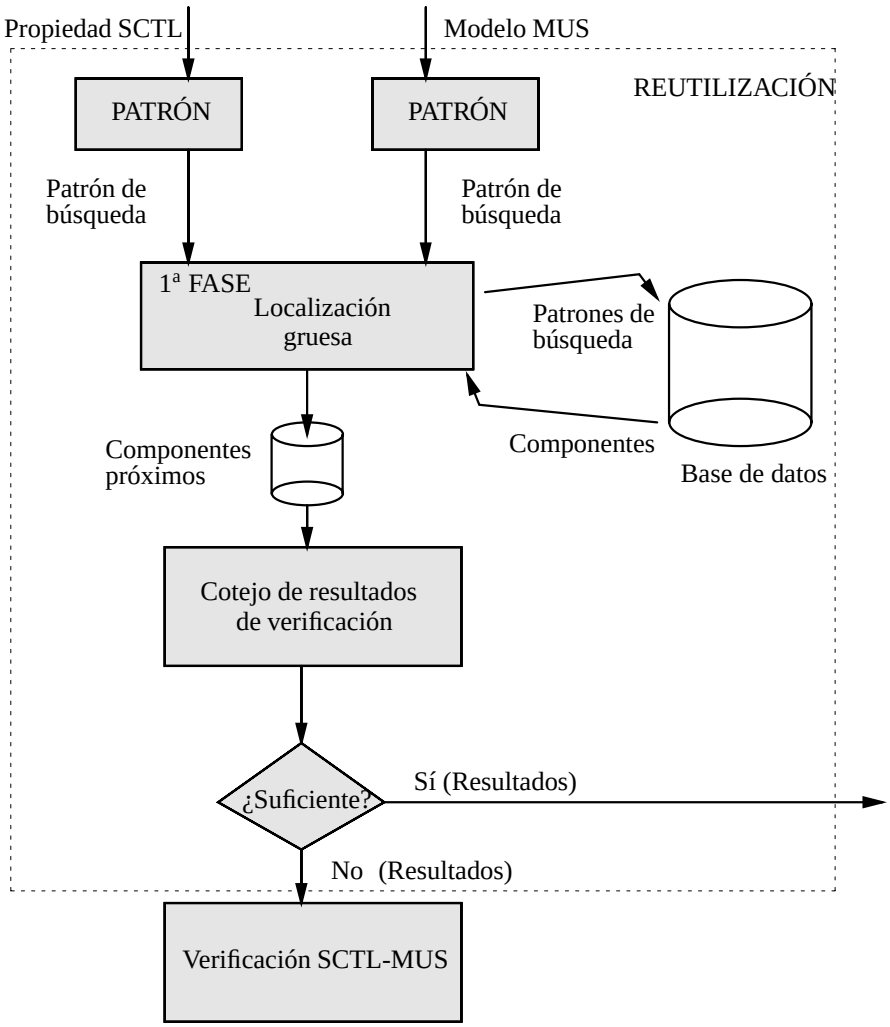


Figura 16.1. Esquema de la reutilización de componentes

Una vez recuperada de la biblioteca toda la información necesaria, almacenada en los componentes localizados, será necesario cotejar los resultados de verificación existentes y extraer alguna conclusión sobre el grado de satisfacción de la propiedad sobre el grafo dado —normalmente interesará deducir si dicha propiedad es una finalidad o una invarianza, universal o existencial, sobre el grafo. Si con la información recuperada no se llega a una conclusión definitiva, se procederá según el algoritmo de verificación tradicional, aunque puede reducirse el número de verificaciones en función de los resultados obtenidos. En otro caso, habremos conseguido obtener el grado de satisfacción de la propiedad en el grafo sin recurrir al algoritmo de verificación.

16.3 Definiciones y notación preliminar

Simulación y bisimulación

Las relaciones de simulación y bisimulación son habitualmente utilizadas para establecer relaciones de equivalencia y orden entre estados de estructuras o entre estructuras representantes del comportamiento de un sistema. Aquí particularizaremos ambas definiciones para establecer relaciones de simulación y bisimulación entre estados de grafos MUS y entre grafos MUS.

Definición 16.1. *Sea g un grafo, una relación binaria $\sqsubseteq_{\mathrm{e}}$ en $\mathcal{E} \times \mathcal{E}$ es una relación de simulación entre dos estados $E_1, E_2 \in \mathcal{E}$, denotándose $E_1 \sqsubseteq_{\mathrm{e}} E_2$, si se satisface que*

$$\forall {E_1}' \mid E_1 \stackrel{\omega}{\rightarrow} {E_1}', \text{ entonces } \exists\, {E_2}' \mid E_2 \stackrel{\omega}{\rightarrow} {E_2}' \text{ y } {E_1}' \sqsubseteq_{\mathrm{e}} {E_2}'$$
$$\text{si } E_1 \stackrel{\omega}{\not\rightarrow} \text{ entonces } E_2 \stackrel{\omega}{\not\rightarrow}$$

Donde $E_1 \stackrel{\omega}{\not\rightarrow}$ expresa que la acción ω está especificada como falso *en el estado E_1, es decir, el sistema no puede evolucionar con esa acción desde el estado E_1.*

Definición 16.2. *Sean g y ${g}'$ dos grafos MUS deterministas, una relación binaria $\sqsubseteq_{\mathrm{e}}$ es una relación de simulación entre ellos, denotándose $g \sqsubseteq_{\mathrm{e}} {g}'$, si sus correspondientes estados iniciales mantienen una relación de simulación, es decir, si se cumple que $E_0 \sqsubseteq_{\mathrm{e}} {E_0}'$. En este caso podemos decir que ${g}'$ simula a g.*

Definición 16.3. *Sea g un grafo, una relación binaria $\cong_e$ en $\mathcal{E} \times \mathcal{E}$ es una relación de bisimulación entre dos estados $E_1, E_2 \in \mathcal{E}$, denotándose $E_1 \cong_e E_2$, si se satisface que*

$\forall {E_1}' \mid E_1 \stackrel{\omega}{\rightarrow} {E_1}'$, *entonces* $\exists\, {E_2}' \mid E_2 \stackrel{\omega}{\rightarrow} {E_2}'$ *y* ${E_1}' \cong_e {E_2}'$
si $E_1 \stackrel{\omega}{\not\rightarrow}$ *entonces* $E_2 \stackrel{\omega}{\not\rightarrow}$

$\forall {E_2}' \mid E_2 \stackrel{\omega}{\rightarrow} {E_2}'$, *entonces* $\exists\, {E_1}' \mid E_1 \stackrel{\omega}{\rightarrow} {E_1}'$ *y* ${E_1}' \cong_e {E_2}'$
si $E_2 \stackrel{\omega}{\not\rightarrow}$ *entonces* $E_1 \stackrel{\omega}{\not\rightarrow}$

Definición 16.4. *Sean g y g' dos grafos MUS deterministas, una relación binaria $\cong_e$ es una relación de bisimulación entre ellos, denotándose $g \cong_e g'$, si sus correspondientes estados iniciales mantienen una relación de bisimulación, es decir, si se cumple que $E_0 \cong {E_0}'$. En este caso podemos decir que g y g' son equivalentes bajo bisimulación.*

Definición 16.5. *Dadas dos vías de evolución π y π', se dice que sus respectivas trazas de estados $E(\pi) = E_0 E_1 \ldots E_n$ y $E(\pi') = {E_0}'{E_1}' \ldots E_m$, donde $n \leq m$, son correspondientes en simulación si y sólo si se satisface que $\forall i = 1, \ldots, n,\ E_i \sqsubseteq_{\mathrm{e}} {E_i}'$.*

Definición 16.6. *Dados dos estados E y E' tal que se satisface que $E \sqsubseteq_{\mathrm{e}} E'$, entonces para cada vía de evolución π comenzando en el estado E existe una vía de evolución π', correspondiente en simulación, comenzando en E'.*

Orden de grados de satisfacción en nivel de conocimiento

El orden parcial que define la lógica SCTL (definición 15.9) es un orden en el *nivel de verdad* de la propiedad. Podemos definir un orden alternativo $\leq_{\mathrm{c}}$ en nivel de conocimiento (Fernández-Vilas, 2002), obteniéndose tres niveles de conocimiento de una propiedad SCTL (figura 16.2):

- $\{1, \widehat{\frac{1}{2}}, 0\}$: nivel superior de conocimiento ya que, independientemente de la especificación que se añada al sistema, ya sabemos cuál es el grado de satisfacción de la propiedad en el mismo.

- $\{\frac{1}{4}, \frac{3}{4}\}$: nivel medio de conocimiento ya que en el estado actual del sistema sabemos que la propiedad está subespecificada, aunque conocemos su tendencia de satisfacción. Es decir, en un modelo de evolución posterior, el valor de verdad de la propiedad será ϕ' tal que $\frac{1}{4} \leq_{\mathrm{c}} \phi'$ (respectivamente $\frac{3}{4} \leq_{\mathrm{c}} \phi'$) para el valor de verdad actual $\frac{1}{4}$ (respectivamente $\frac{3}{4}$).

➢ $\{\frac{1}{2}\}$: el menor nivel de conocimiento del grado de satisfacción de la propiedad, ya que no tenemos ningún dato sobre su tendencia de satisfacción en evoluciones posteriores del sistema.

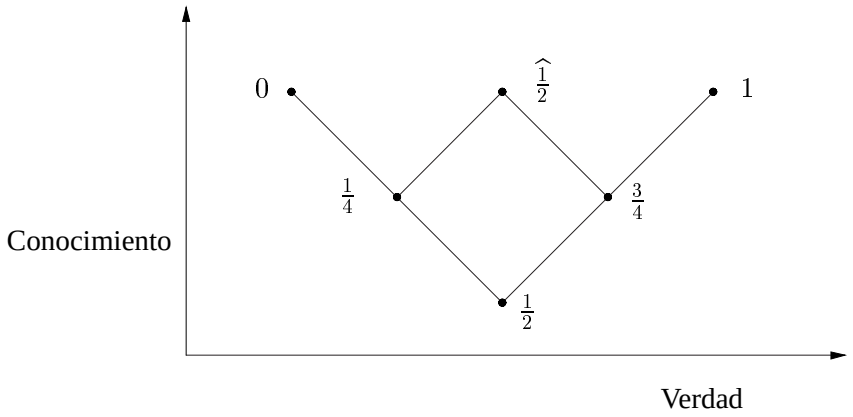


Figura 16.2. Representación del orden de verdad y del orden de conocimiento de los grados de satisfacción

Tal y como se ve en la figura 16.2, el orden $\leq_c$ no es un retículo, sino un semiretículo completo, ya que aunque para cada par de elementos existe un ínfimo y un mínimo, sin embargo no siempre existe un supremo y además no existe un máximo.

Para el orden definido se cumple que los operadores $\wedge$, $\vee$, $\neg$ y $\rightarrow$ son monótonos:

- $\wedge$ es un operador monótono respecto a $\leq_c$:

$$a \leq_c a' \text{ y } b \leq_c b', \text{ entonces } (a \wedge b) \leq_c (a' \wedge b')$$

- $\vee$ es un operador monótono respecto a $\leq_c$:

$$a \leq_c a' \text{ y } b \leq_c b', \text{ entonces } (a \vee b) \leq_c (a' \vee b')$$

- El operador causal $\rightarrow$ es monótono respecto a $\leq_c$:

$$a \leq_c a' \text{ y } b \leq_c b', \text{ entonces } (a \rightarrow b) \leq_c (a' \rightarrow b')$$

- El operador $\neg$ es monótono respecto a la relación de orden $\leq_c$:

$$a \leq_c b, \text{ entonces } \neg a \leq_c \neg b$$

16.4 Particularización para grafos deterministas

Dados dos grafos g y g' cualesquiera se satisface que la relación de equivalencia de bisimulación es *más fuerte* que la relación de equivalencia $=^{\infty}_{\mathrm{TC}}$, es decir, se cumple que si $g \cong_e g'$, entonces se satisface que $g =^{\infty}_{\mathrm{TC}} g'$. De forma análoga la relación de orden de simulación $\sqsubseteq_{\mathrm{e}}$ es *más fuerte* que la relación de orden definida por $\sqsubseteq^{\infty}_{\mathrm{TC}}$, es decir, si se cumple que $g \sqsubseteq_{\mathrm{e}} g'$, entonces se satisface que $g \sqsubseteq^{\infty}_{\mathrm{TC}} g'$.

Propiedad 16.1. *Para el caso de grafos deterministas, se cumple que la relación de equivalencia de simulación y la relación de equivalencia $=^{\infty}_{\mathrm{TC}}$ son idénticas, es decir, también se cumple que si $g =^{\infty}_{\mathrm{TC}} g'$, entonces $g \cong_e g'$. De la misma manera, el orden definido por $\sqsubseteq_{\mathrm{e}}$ es equivalente al definido por $\sqsubseteq^{\infty}_{\mathrm{TC}}$, es decir, si $g \sqsubseteq^{\infty}_{\mathrm{TC}} g'$, entonces $g \sqsubseteq_{\mathrm{e}} g'$.*

La propiedad anterior puede justificarse basándose en la relación $\sqsubseteq^{\infty}_{\mathrm{TC}}$ y en las propiedades de los grafos deterministas. Dados g y g', satisfaciendo que $g \sqsubseteq^{\infty}_{\mathrm{TC}} g'$, entonces sabemos que para toda traza $a_0 a_1 \ldots$ de g, existe una traza $a_0{}' a_1{}' \ldots$ de g' tal que la primera está contenida en la segunda.

Por ser ambos grafos deterministas, sabemos que para toda traza de g existe una única traza de estados $E(\pi)$ posible, es decir, la secuencia de eventos definida en $a_0 a_1 \ldots$ provoca la transición del sistema por los estados de $E(\pi)$ (análogamente para g').

Dado que $g \sqsubseteq^{\infty}_{\mathrm{TC}} g'$, podemos deducir que $E(\pi)$ y $E(\pi')$ son trazas correspondientes en simulación (definición 16.5), y como esto se mantiene para todas las trazas a partir del estado inicial, entonces sabemos que $E_0 \sqsubseteq_{\mathrm{e}} E_0{}'$, de donde podemos concluir que $g \sqsubseteq_{\mathrm{e}} g'$. □

Estas dos conclusiones, obtenidas para grafos deterministas —con los que nosotros trabajamos— nos permitirán la obtención de cotas superiores e inferiores en los resultados de verificación de una propiedad sobre un grafo a partir de los resultados de verificación de dicha propiedad en los grafos continentes y contenidos según la relación TC^{∞}.

16.5 Orden de grados de satisfacción entre estados simulables

Propiedad 16.2. *Dados dos estados E y E', donde $E \sqsubseteq_{\mathrm{e}} E'$, y una propiedad R, se satisface que $\vDash (R, E) \leq_{\mathrm{c}} \vDash (R, E')$, es decir el grado de conocimiento de la propiedad en el estado simulador es siempre mayor o igual que en el estado simulado.*

Demostración 16.1. *Se puede demostrar la propiedad anterior por inducción*

sobre la estructura de la propiedad R, basándonos en la monotonicidad de los operadores lógicos $\wedge$, $\vee$, $\neg$ y del operador causal $\rightarrow$ respecto al orden $\leq_c$.

1. *Sea $R = (true \Rightarrow a_i) \mid a_i \in \Lambda$, un requisito atómico donde a_i es un evento perteneciente al conjunto de acciones de especificación. En este caso sabemos que se satisface, por la definición 16.1 de simulación, que $\vDash (R, E) \leq_c \vDash (R, E')$.*

   Dado un estado E' que simula al estado E, sabemos que todos los eventos especificados como verdadero *y como* falso *en E mantienen su especificación en E'. Además, los eventos caracterizados como* subespecificados *en E pueden mantener su nivel de especificación en E' o bien pasar a estar especificados como* verdadero *o como* falso*, es decir, a un grado de conocimiento mayor según $\leq_c$.*

2. *$R = R_1 \vee R_2$. Entonces sabemos que*

$$\vDash (R, E) = \vDash (R_1, E) \vee \vDash (R_2, E)$$

   Aplicando la hipótesis de inducción, entonces $\vDash (R_1, E) \leq_c \vDash (R_1, E')$ y $\vDash (R_2, E) \leq_c \vDash (R_2, E')$. Como el operador lógico $\vee$ es monótono respecto al orden $\leq_c$, entonces

$$\vDash (R_1, E) \vee \vDash (R_2, E) \leq_c \vDash (R_1, E') \vee \vDash (R_2, E')$$

3. *$R = R_1 \wedge R_2$. En esta situación también se cumple*

$$\vDash (R_1, E) \wedge \vDash (R_2, E) \leq_c \vDash (R_1, E') \wedge \vDash (R_2, E')$$

   sin más que aplicar la propiedad de monotonicidad del operador lógico $\wedge$ de forma equivalente a como se hizo en el punto 2.

4. *$R = R_1 \Rightarrow R_2$. Por la hipótesis de inducción sabemos que se satisface $\vDash (R_1, E) \leq_c \vDash (R_1, E')$ y $\vDash (R_2, E) \leq_c \vDash (R_2, E')$ y dado que la operación causal $\rightarrow$ es monótona, entonces*

$$\vDash (R_1 \Rightarrow R_2, E) = \vDash (R_1, E) \rightarrow \vDash (R_2, E) \leq_c \vDash (R_1 \Rightarrow R_2, E')$$

5. *$R = R_1 \Rightarrow\!\bigcirc R_2$. Por la definición del grado de satisfacción de una propiedad con esta sintaxis (apartado 9.3.2) sabemos que:*

$$\vDash (R_1 \Rightarrow\!\bigcirc R_2, E) = \vDash (R_1, E) \rightarrow \bigwedge_{E_i \in \perp(R_E)} \vDash (R_2, E_i)$$

   Como las operaciones $\rightarrow$ y $\wedge$ son monótonas y, debido a la hipótesis de inducción, podemos concluir que se cumple $\vDash (R, E) \leq_c \vDash (R, E')$

6. $R = R_1 \Rightarrow\!\!\odot R_2$. *Por la definición del grado de satisfacción de una propiedad con esta sintaxis (apartado 9.3.2) sabemos que:*

$$\vDash (R_1 \Rightarrow\!\!\odot R_2, E) \;\; = \vDash (R_1, E) \;\; \rightarrow \;\; \bigwedge_{E_i \in \perp(R_E)} \vDash (R_2, E_i)$$

Como las operaciones $\rightarrow$ y $\wedge$ son monótonas y, debido a la hipótesis de inducción, podemos concluir que se cumple $\vDash (R, E) \;\; \leq_{\mathrm{c}} \;\; \vDash (R, E')$

7. $R = \neg R_1$. *Por la hipótesis de inducción, sabemos que se satisface* $\vDash (R_1, E) \leq_{\mathrm{c}} \vDash (R_1, E')$ *y, dado que el operador $\neg$ es monótono respecto a la relación $\leq_{\mathrm{c}}$, entonces también se satisface que* $\vDash (R, E) \leq_{\mathrm{c}} \vDash (R, E')$.

□

16.6 Orden de grados de satisfacción entre trazas correspondientes en simulación

Dadas dos trazas $E(\pi) \sqsubseteq_{\mathrm{e}} E(\pi')$ correspondientes en simulación (definición 16.5), sabemos, por definición, que todo el comportamiento especificado en la traza $E(\pi)$ estará contenido en la traza $E(\pi')$. Nos interesará reutilizar el máximo conocimiento posible que se tenga de los grados de satisfacción de un requisito R sobre la traza $E(\pi)$ para reducir las tareas de verificación sobre la traza $E(\pi')$ y viceversa. En este apartado se detalla qué información es posible reutilizar en ambas situaciones.

Conociendo los resultados sobre $E(\pi)$

Suponiendo que conocemos los resultados de verificación de R sobre $E(\pi)$, nos interesa compendiar qué podemos extraer sobre los resultados de R en $E(\pi')$ sin necesidad de verificar R sobre esta última traza. Principalmente, interesará conocer los grados de $\vDash (\Diamond R, E(\pi'))$ y $\vDash (\Box R, E(\pi'))$.

Es necesario notar que, debido a la definición de trazas correspondientes en simulación, la traza $E(\pi')$ puede constar de más estados que la $E(\pi)$, estados que estaban totalmente subespecificados en la traza simulada.

En la tabla 16.1 podemos ver qué información extraer sobre $\vDash (\Diamond R, E(\pi'))$ y $\vDash (\Box R, E(\pi'))$ a partir de $\vDash (\Diamond R, E(\pi))$:

- ➢ Si una propiedad R es una finalidad sobre la traza simulada, seguirá siendo una finalidad sobre la traza que la simula.

- ➢ Si una propiedad R tiene un grado de satisfacción de finalidad sobre la traza simulada mayor o igual en conocimiento que $\frac{3}{4}$, seguirá manteniéndose su

grado de satisfacción como finalidad en la traza que la simula, salvo que éste sea 1 (ver punto anterior). Es decir, si en la traza simulada *podría llegar a ser una finalidad* es imposible que en la traza que la contiene *no pueda llegar a serlo*.

➢ Si una propiedad R no es una finalidad sobre la traza simulada, entonces tampoco es una invarianza sobre la traza que la simula.

➢ Si una propiedad R tiene un grado de satisfacción de finalidad sobre la traza simulada mayor o igual en conocimiento que $\frac{1}{4}$, mantiene el nivel de conocimiento para la invarianza de la traza que la simula, salvo que éste valor sea 0 (ver punto anterior). Es decir, si en la traza simulada *no puede llegar a ser una finalidad* entonces en la traza que la contiene *no puede llegar a ser una invarianza*.

En la tabla 16.2 podemos ver que la información que se deduce a partir de los resultados $\vDash (\Box R, \pi)$ es idéntica a la que se deduce de los resultados de $\vDash (\Diamond R, \pi)$ de la tabla 16.1.

$\vDash (\Diamond R, E(\pi)) = 1$	$\vDash (\Diamond R, E(\pi')) = 1$
$\vDash (\Diamond R, E(\pi)) = \widehat{\frac{1}{2}}$	$\vDash (\Diamond R, E(\pi')) \geq_c \frac{3}{4}$
$\vDash (\Diamond R, E(\pi)) = \frac{3}{4}$	
$\vDash (\Diamond R, E(\pi)) = 0$	$\vDash (\Box R, E(\pi')) = 0$
$\vDash (\Diamond R, E(\pi)) = \widehat{\frac{1}{2}}$	$\vDash (\Box R, E(\pi')) \geq_c \frac{1}{4}$
$\vDash (\Diamond R, E(\pi)) = \frac{1}{4}$	

Tabla 16.1. Resultados de verificación proporcionados por $\vDash (\Diamond R, E(\pi))$

Conociendo los resultados sobre $E(\pi')$

Si tenemos la situación contraria, entonces conociendo información de verificación sobre $E(\pi')$ tendremos que obtener qué podemos concluir sobre los resultados de verificación en $E(\pi)$. En la tabla 16.3 se ha compendiado la información que se deduce de los resultados de $\vDash (\Diamond R, E(\pi'))$:

➢ Si el grado de satisfacción de una propiedad R, como finalidad en la traza simuladora, es igual o menor en conocimiento que 0, entonces el grado de satisfacción de esta propiedad como invarianza y como finalidad sobre la

$\models (\Box R, E(\pi)) = 1$	$\models (\Diamond R, E(\pi\,')) = 1$
$\models (\Box R, E(\pi)) = \widehat{\frac{1}{2}}$	$\models (\Diamond R, E(\pi\,')) \geq_c \frac{3}{4}$
$\models (\Box R, E(\pi)) = \frac{3}{4}$	
$\models (\Box R, E(\pi)) = 0$	$\models (\Box R, E(\pi\,')) = 0$
$\models (\Box R, E(\pi)) = \widehat{\frac{1}{2}}$	$\models (\Box R, E(\pi\,')) \geq_c \frac{1}{4}$
$\models (\Box R, E(\pi)) = \frac{1}{4}$	

Tabla 16.2. Resultados de verificación proporcionados por $\models (\Box R, E(\pi))$

traza simulada mantiene el nivel de conocimiento, es decir, si el grado de satisfacción implica que no es una finalidad, o bien que está parcialmente especificada pero sin posibilidades de ser una finalidad, o bien está totalmente subespecificada, entonces implica que la propiedad sobre la traza simulada mantiene estos resultados como invarianza y como finalidad.

➢ Si el grado de satisfacción de una propiedad R como finalidad en la traza simuladora es contradictorio ($\widehat{\frac{1}{2}}$) o parcialmente subespecificado pero con opciones de poder llegar a satisfacerse con un nivel de especificación mayor ($\frac{3}{4}$), entonces sólo sabemos que el grado de satisfacción de esta propiedad como finalidad y como invarianza en la traza simulada puede adoptar cualquier valor, salvo el posible (1). Es decir, no es una finalidad ni una invarianza en $E(\pi)$.

En la tabla 16.4 se han reunido los resultados de verificación que sobre la traza $E(\pi)$ pueden ser deducidos a partir de $\models (\Box E, \pi')$:

$\models (\Diamond R, E(\pi\,')) \leq_c 0$	$\models (\Diamond R, E(\pi)) \leq_c 0$
	$\models (\Box R, E(\pi)) \leq_c 0$
$\models (\Diamond R, E(\pi\,')) = \widehat{\frac{1}{2}}$ ó $\models (\Diamond R, E(\pi\,')) = \frac{3}{4}$	$\models (\Diamond R, E(\pi)) \in \Phi - \{1\}$
	$\models (\Box R, E(\pi)) \in \Phi - \{1\}$

Tabla 16.3. Resultados de verificación proporcionados por $\models (\Diamond R, E(\pi\,'))$

- Si el grado de satisfacción de una propiedad R, como invarianza en la traza simuladora, es igual o menor en conocimiento que 1, entonces el grado de satisfacción de esta propiedad como invarianza y como finalidad sobre la traza simulada mantiene el nivel de conocimiento, es decir, si el grado de satisfacción implica que es una invarianza, o bien que está parcialmente especificada pero sin posibilidades de ser una invarianza, o bien está totalmente subespecificada, entonces implica que la propiedad sobre la traza simulada mantiene estos resultados como invarianza y como finalidad.

- Si el grado de satisfacción de una propiedad R como invarianza en la traza simuladora es contradictorio ($\widehat{\frac{1}{2}}$) o parcialmente subespecificado sin opciones de poder llegar a satisfacerse con un nivel de especificación mayor ($\frac{1}{4}$), entonces sólo sabemos que el grado de satisfacción de esta propiedad como finalidad y como invarianza en la traza simulada puede adoptar cualquier valor, salvo el no posible (0), es decir, puede llegar a ser una finalidad y una invarianza en $E(\pi)$.

$\vDash (\Box R, E(\pi\,')) \leq_{\mathrm{c}} 1$	$\vDash (\Diamond R, E(\pi)) \leq_{\mathrm{c}} 1$
	$\vDash (\Box R, E(\pi)) \leq_{\mathrm{c}} 1$
$\vDash (\Box R, E(\pi\,')) = \widehat{\frac{1}{2}}$ ó $\vDash (\Box R, E(\pi\,')) = \frac{1}{4}$	$\vDash (\Diamond R, E(\pi)) \in \Phi - \{0\}$
	$\vDash (\Box R, E(\pi)) \in \Phi - \{0\}$

Tabla 16.4. Resultados de verificación proporcionados por $\vDash (\Box R, E(\pi\,'))$

16.7 Orden de grados de satisfacción entre grafos simulables

De forma análoga a como se procedió en el apartado 16.6 para extraer información de reutilización entre trazas correspondientes en simulación, en este apartado intentaremos establecer qué resultados de verificación podemos reutilizar entre dos grafos simulables. La situación es semejante y partimos de dos grafos MUS satisfaciendo $g \sqsubseteq_{\mathrm{e}} g\,'$. Nos interesará reutilizar el máximo conocimiento posible que se tenga de los grados de satisfacción de un requisito R sobre el grafo g para reducir las tareas de verificación sobre la traza g' y viceversa. En este apartado se detalla qué información es posible reutilizar en ambas situaciones.

Conociendo los resultados sobre g

Suponiendo que conocemos los resultados de verificar una propiedad R sobre el grafo simulado g, es decir la cuádrupla de valores $\vDash (R, g)$, se ha deducido qué valores de la cuádrupla $\vDash (R, g')$ podemos extraer a partir de ellos.

En la tabla 16.5 se muestra qué conclusiones podemos sacar según los grados de satisfacción de R como finalidad existencial sobre g:

- ➢ Si la propiedad R no es una finalidad existencial en g, entonces tampoco será una invarianza universal en g'.
- ➢ Si la propiedad R es una finalidad existencial en g, también será una finalidad existencial sobre g'.
- ➢ Si la propiedad R no puede llegar a ser una finalidad existencial sobre g, entonces tampoco podrá llegar a ser una invarianza universal sobre g'.

<table>
<tr><td>$\vDash (\exists \Diamond R, g) = 0$</td><td>$\vDash (\forall \Box R, g') = 0$</td></tr>
<tr><td>$\vDash (\exists \Diamond R, g) = 1$</td><td>$\vDash (\exists \Diamond R, g') = 1$</td></tr>
<tr><td>$\vDash (\exists \Diamond R, g) = \widehat{\frac{1}{2}}$</td><td rowspan="2">$\vDash (\forall \Box R, g') \geq_c \frac{1}{4}$</td></tr>
<tr><td>$\vDash (\exists \Diamond R, g) = \frac{1}{4}$</td></tr>
</table>

Tabla 16.5. Resultados de verificación proporcionados por $\vDash (\exists \Diamond R, g)$

En la tabla 16.6 vemos qué conclusiones podemos sacar según los grados de satisfacción de R como finalidad universal sobre g:

- ➢ Si la propiedad R no es una finalidad universal en g, entonces tampoco será una invarianza universal en g'.
- ➢ Si la propiedad R es una finalidad universal en g, entonces será una finalidad existencial sobre g'.
- ➢ Si la propiedad R tiene un grado de satisfacción contradictorio ($\widehat{\frac{1}{2}}$) o parcialmente subespecificado con opción a llegar a ser una finalidad existencial sobre g ($\frac{3}{4}$), entonces la propiedad R puede llegar a ser una finalidad existencial sobre el grafo simulador (g').

En la tabla 16.7 vemos qué conclusiones podemos sacar según los grados de satisfacción de R como invarianza existencial sobre g:

$\vDash (\forall \Diamond R, g) = 0$	$\vDash (\forall \Box R, g') = 0$
$\vDash (\forall \Diamond R, g) = 1$	$\vDash (\exists \Diamond R, g') = 1$
$\vDash (\forall \Diamond R, g) = \widehat{\frac{1}{2}}$	$\vDash (\exists \Diamond R, g') \geq_c \frac{3}{4}$
$\vDash (\forall \Diamond R, g) = \frac{3}{4}$	

Tabla 16.6. Resultados de verificación proporcionados por $\vDash (\forall \Diamond R, g)$

- ➢ Si la propiedad R no es una invarianza existencial en g, entonces tampoco será una invarianza universal en g'.
- ➢ Si la propiedad R es una invarianza existencial en g, entonces también será una finalidad existencial sobre g'.
- ➢ Si la propiedad R no puede llegar a ser nunca una finalidad existencial sobre g, entonces tampoco podrá llegar a ser una invarianza universal sobre g'.

$\vDash (\exists \Box R, g) = 0$	$\vDash (\forall \Box R, g') = 0$
$\vDash (\exists \Box R, g) = 1$	$\vDash (\exists \Diamond R, g') = 1$
$\vDash (\exists \Box R, g) = \widehat{\frac{1}{2}}$	$\vDash (\forall \Box R, g') \geq_c \frac{1}{4}$
$\vDash (\exists \Box R, g) = \frac{1}{4}$	

Tabla 16.7. Resultados de verificación proporcionados por $\vDash (\exists \Box R, g)$

Y por último, en la tabla 16.8 vemos qué conclusiones podemos sacar según los grados de satisfacción de R como invarianza universal sobre g:

- ➢ Si la propiedad R no es una invarianza universal en g, entonces tampoco será una invarianza universal en g'.
- ➢ Si la propiedad R es una invarianza universal en g, entonces también será una finalidad existencial sobre g'.
- ➢ Si la propiedad R tiene un grado de satisfacción contradictorio ($\widehat{\frac{1}{2}}$) o parcialmente subespecificado con opción a llegar a ser una invarianza universal ($\frac{3}{4}$), estoces R podrá llegar a ser una finalidad existencial sobre g'.

$\vDash (\forall\, \Box R, g) = 0$	$\vDash (\forall\, \Box R, g\,') = 0$
$\vDash (\forall\, \Box R, g) = 1$	$\vDash (\exists\, \Diamond R, g\,') = 1$
$\vDash (\forall\, \Box R, g) = \widehat{\frac{1}{2}}$	$\vDash (\exists\, \Diamond R, g\,') \geq_c \frac{3}{4}$
$\vDash (\forall\, \Box R, g) = \frac{3}{4}$	

Tabla 16.8. Resultados de verificación proporcionados por $\vDash (\forall\, \Box R, g)$

Conociendo los resultados sobre $g\,'$

Conociendo ahora los resultados de verificación de una propiedad R sobre el grafo simulador $g\,'$, ¿qué resultados de verificación de R en g pueden ser extrapolados? En este caso, de la cuádrupla disponible $\vDash (R, g')$, sólo los resultados de verificación de R como invarianza universal y como finalidad existencial pueden dar alguna información.

$\vDash (\forall\, \Box R, g\,') \leq_{\mathrm{c}} 1$	$\vDash (\exists\, \Diamond R, g) \leq_{\mathrm{c}} 1$
	$\vDash (\forall\, \Diamond R, g) \leq_{\mathrm{c}} 1$
	$\vDash (\exists\, \Box R, g) \leq_{\mathrm{c}} 1$
	$\vDash (\forall\, \Box R, g) \leq_{\mathrm{c}} 1$
$\vDash (\forall\, \Box R, g\,') = \widehat{\frac{1}{2}}$ ó $\vDash (\forall\, \Box R, g\,') = \frac{1}{4}$	$\vDash (\exists\, \Diamond R, g) \in \Phi - \{0\}$
	$\vDash (\forall\, \Diamond R, g) \in \Phi - \{0\}$
	$\vDash (\exists\, \Box R, g) \in \Phi - \{0\}$
	$\vDash (\forall\, \Box R, g) \in \Phi - \{0\}$

Tabla 16.9. Resultados de verificación proporcionados por $\vDash (\forall\, \Box R, g\,')$

En la tabla 16.9 puede verse qué resultados de verificación podemos extraer de los grados de satisfacción de R como invarianza universal sobre g':

- Si R es una invarianza universal sobre g' (1) o bien está parcialmente subespecificada, aunque con posibilidad de llegar a serlo ($\frac{3}{4}$), o bien está totalmente subespecificada ($\frac{1}{2}$), entonces todos los valores de la cuádrupla $\vDash (R, g)$ están limitados a uno de esos tres valores.

➢ Si el grado de satisfacción de R como invarianza universal en g' es contradictorio ($\widehat{\frac{1}{2}}$) o bien parcialmente subespecificado pero sin opción a llegar a ser nunca cierto ($\frac{1}{4}$), entonces todos los valores de la cuádrupla $\vDash (R, g)$ pueden adoptar cualquier grado de satisfacción, salvo el 0.

En la tabla 16.10 puede verse qué resultados de verificación podemos extraer de los grados de satisfacción de R como finalidad existencial sobre g':

➢ Si R no es una finalidad existencial sobre g' (0) o bien está parcialmente subespecificada, sin posibilidad de llegar a serlo ($\frac{1}{4}$), o bien está totalmente subespecificada ($\frac{1}{2}$), entonces todos los valores de la cuádrupla $\vDash (R, g)$ están limitados a uno de esos tres valores.

➢ Si el grado de satisfacción de R como finalidad existencial en g' es contradictorio ($\widehat{\frac{1}{2}}$) o bien parcialmente subespecificado pero con opción a llegar a ser cierto con un grado mayor de especificación ($\frac{3}{4}$), entonces todos los valores de la cuádrupla $\vDash (R, g)$ pueden adoptar cualquier grado de satisfacción, salvo el 1.

$\vDash (\exists \Diamond R, g') \leq_c 0$	$\vDash (\exists \Diamond R, g) \leq_c 0$
	$\vDash (\forall \Diamond R, g) \leq_c 0$
	$\vDash (\exists \Box R, g) \leq_c 0$
	$\vDash (\forall \Box R, g) \leq_c 0$
$\vDash (\exists \Diamond R, g') = \widehat{\frac{1}{2}}$ ó $\vDash (\exists \Diamond R, g') = \frac{3}{4}$	$\vDash (\exists \Diamond R, g) \in \Phi - \{1\}$
	$\vDash (\forall \Diamond R, g) \in \Phi - \{1\}$
	$\vDash (\exists \Box R, g) \in \Phi - \{1\}$
	$\vDash (\forall \Box R, g) \in \Phi - \{1\}$

Tabla 16.10. Resultados de verificación proporcionados por $\vDash (\exists \Diamond R, g')$

16.7.1 Resultados de verificación del estado inicial

En los apartados anteriores hemos visto cómo la cuádrupla $\vDash (R, g)$ podía ofrecernos información sobre los grados de satisfacción de ese mismo requisito en un grafo que lo simule (que lo contenga según la relación TC^{∞}) o en un grafo simulado por éste (que esté contenido según la relación TC^{∞}). En este apartado

trataremos de extraer aún más información teniendo en cuenta la particularidad de que el estado inicial está contenido en todas y cada una de las trazas de estados del grafo.

Propiedad 16.3. *Si una propiedad R se satisface en el estado inicial de un grafo g, entonces sabemos que* $\models (\forall \Diamond R, g) = 1$ *y además* $\models (\forall \Diamond R, g') = 1, \forall g' \mid g \sqsubseteq_{\mathrm{e}} g'$.

Esta propiedad ofrece más información que la suministrada en la tabla 16.6. En esta tabla podemos ver que si una propiedad es una finalidad existencial sobre un grafo g, entonces podemos asegurar que va a ser una finalidad existencial sobre cualquier grafo g' que lo simule. Con la condición impuesta sobre el estado inicial y, dado que éste es común a todas las trazas de evolución posibles del sistema, podemos asegurar que si la propiedad se satisface en él, entonces se satisface en todas y cada una de las trazas, con lo que pasa a ser una finalidad universal en todos los posibles grafos que lo simulen.

16.7.2 Conclusiones

Dados dos grafos $g \sqsubseteq_{\mathrm{e}} g'$ y, tras estudiar la información de verificación que podemos reutilizar resumida en las tablas anteriores, podemos concluir que:

- Conociendo los resultados de $\models (R, g)$ sólo podemos establecer cotas sobre los grados de satisfacción de $\models (\exists \Diamond R, g')$ y $\models (\forall \Box R, g')$. Para extraer alguna información sobre el grado de satisfacción de $\models (\forall \Diamond R, g')$ es preciso recurrir al resultado de $\models (R, E_0)$.

- Conociendo los resultados de $\models (R, g')$ sólo los resultados de $\models (\exists \Diamond R, g')$ y $\models (\forall \Box R, g')$ nos ofrecen cotas sobre los valores de la cuádrupla $\models (R, g)$.

16.8 Ejemplo de aplicación

El principal propósito de esta sección es clarificar todos los conceptos referentes a la reutilización de información de verificación utilizando para ello un pequeño ejemplo de aplicación.

En las figuras 16.4(a) y 16.4(b) pueden verse dos grafos MUS no comparables según el criterio TC^{∞} y, consecuentemente, no comparables según la relación de simulación definida en 16.2. Conocemos los resultados de verificar sobre ellos los

requisitos $R_1 = (a \Rightarrow b)$ y $R_2 = (c \Rightarrow a)$:

$$\models (R_1, g_1) = \{1, 1, \frac{1}{4}, 0\} \qquad \models (R_2, g_1) = \{1, \frac{3}{4}, \frac{1}{2}, 0\}$$
$$\models (R_1, g_2) = \{1, \frac{1}{2}, \frac{1}{2}, \frac{1}{4}\} \qquad \models (R_2, g_2) = \{1, 1, \frac{1}{2}, 0\}$$

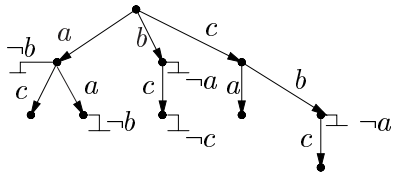
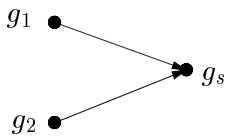
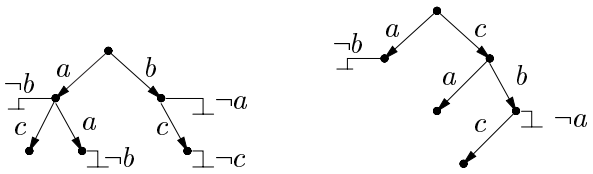


Figura 16.3. Grafo g_s donde se desea conocer los resultados de verificar las propiedades R_1 y R_2

Nos interesa conocer qué resultados de verificación de estas mismas propiedades, sobre el grafo de la figura 16.3, podemos extraer a partir de estos datos, sabiendo que la relación de orden $\sqsubseteq^{\infty}_{\mathrm{TC}}$ entre los grafos g_1, g_2 y g_s es la indicada en la figura siguiente:





(a) g_1 (b) g_2

Figura 16.4. Dos grafos MUS a partir de los que deducir información de verificación

Información sobre R_1

Sabemos que la información de verificación de R_1 sobre g_1 está especificada en la cuádrupla $\models (R_1, g_1) = \{1, 1, \frac{1}{4}, 0\}$, de donde podemos concluir los siguientes datos:

1. Conociendo que $\vDash (\exists \Diamond R_1, g_1) = 1$, y por los valores de la tabla 16.5, sabemos que $\vDash (\exists \Diamond R_1, g_s) = 1$.

2. Sabiendo que $\vDash (\forall \Diamond R_1, g_1) = 1$ no podemos extraer ninguna información concluyente (tabla 16.6), pero como $\vDash (R_1, E_0) = 1$, entonces por las características del estado de inicio podemos deducir que $\vDash (\forall \Diamond R_1, g_s) = 1$.

3. El hecho de que $\vDash (\exists \Box R_1, g_1) = \frac{1}{4}$ no nos proporciona ningún dato relevante (tabla 16.7).

4. Como $\vDash (\forall \Box R_1, g_1) = 0$, entonces podemos concluir, analizando los resultados de la tabla 16.8, que $\vDash (\forall \Box R_1, g_s) = 0$.

Así que la información que nos proporciona el grafo g_1 puede resumirse en la cuádrupla $\vDash (R_1, g_s) = \{1, 1, \phi, 0\}$, donde ϕ expresa cualquier valor de satisfacción. De modo que, sin tener que recurrir a la información ofrecida por el grafo g_2 y sin tener que realizar ninguna labor de verificación podemos conocer *a priori* que la propiedad R_1 es una finalidad universal en g_s, es decir, es una propiedad de viveza independientemente del comportamiento del sistema. Además, también podemos decir que no es una invarianza universal en g_s, es decir, no es una propiedad de seguridad.

El valor que falta en la cuádrupla, grado de satisfacción de R_1 como invarianza existencial, no es posible obtenerlo a partir de la información disponible, así que sería necesario estudiar detalladamente cada una de las trazas de g_s para deducir alguna cota sobre él.

Información sobre R_2

Para obtener los grados de satisfacción de R_2 en g_s procederemos de igual forma, estudiando primero toda la información que podamos extraer de la verificación de esta propiedad sobre g_1, $\vDash (R_2, g_1) = \{1, \frac{3}{4}, \frac{1}{2}, 0\}$:

1. Dado que $\vDash (\exists \Diamond R_2, g_1) = 1$, entonces por los valores de la tabla 16.5 sabemos que $\vDash (\exists \Diamond R_2, g_s) = 1$.

2. El valor $\vDash (\forall \Diamond R_2, g_1) = \frac{3}{4}$ no nos ofrece ningún dato concluyente sobre $\vDash (\forall \Diamond R_2, g_s)$ (tabla 16.6).

3. El valor $\vDash (\exists \Box R_2, g_1) = \frac{1}{2}$ tampoco nos permite deducir ningún valor sobre $\vDash (\exists \Box R_2, g_s)$ (tabla 16.7).

4. Sin embargo sabiendo que $\vDash (\forall \Box R_2, g_1) = 0$, entonces podemos concluir que $\vDash (\forall \Box R_2, g_s) = 0$ (tabla 16.8).

Para estudiar los valores que nos quedan recurriremos a la información proporcionada por el grafo g_2:

1. Conociendo que el valor $\vDash (R_2, E_0) = 1$, entonces es posible deducir que $\vDash (\forall \Diamond R_2, g_s) = 1$.

2. El valor $\vDash (\exists \Box R_2, g_2) = \frac{1}{2}$ tampoco nos permite deducir ningún valor sobre $\vDash (\exists \Box R_2, g_s)$ (tabla 16.7).

En este caso podemos concluir que la cuádrupla $\vDash (R_2, g_s)$ tiene el valor $\{1, 1, \phi, 0\}$, donde ϕ puede ser cualquier valor de satisfacción. Es decir, la propiedad R_2 es una propiedad de viveza en g_s, satisfaciéndose por cualquiera de las vías por las que transiciones el sistema y, además, no es una propiedad de seguridad en dicho grafo. Para llegar a estas conclusiones hemos recurrido a la información almacenada en dos grafos diferentes y no comparables según $\sqsubseteq^{\infty}_{\mathrm{TC}}$, pero que están contenidos, según la relación de trazas completas no acotadas en el dado. □

PARTE VI

Ejemplo de aplicación

CAPÍTULO 17

Ejemplo de aplicación

Con el objetivo de aplicar las facilidades proporcionadas por el entorno de reutilización detallado en esta tesis, se incluye este capítulo donde se aplicará la reutilización de componentes y de información de verificación para el desarrollo de un proceso emisor y un proceso receptor sobre un canal que produce errores, aunque no pérdidas de mensajes, para un protocolo de parada y espera. Para ello, en una primera parte, se detallan los requisitos funcionales que deben satisfacer dichos sistemas y se ofrece una visión resumida de los componentes almacenados en la biblioteca de componentes. Seguidamente se explica cómo reutilizar, en este entorno concreto, alguno de los modelos ya existentes y, por último, se intentará reutilizar la información de verificación disponible para conseguir averiguar el grado de satisfacción de una propiedad dada sobre el sistema en desarrollo, sin necesidad de ejecutar el algoritmo de model checking.

17.1 Descripción del protocolo

Cualquier protocolo de comunicación establece un conjunto de parámetros que rigen de forma ordenada la comunicación entre dos entidades, de forma que éstas logren entenderse. El conjunto de protocolos más sencillo es aquél donde el emisor de datos, o proveedor de la información, espera un acuse de recibo del receptor antes de continuar con la transmisión. Este conjunto de protocolos se conoce como protocolos de parada y espera (Tanenbaum, 1997). Bajo esta base común se pueden construir diversidad de protocolos teniendo en cuenta aspectos tan diferentes como la fiabilidad del canal sobre el que se establece la comunicación, la posibilidad de que la comunicación sea en ambos o en un único sentido, que el acuse de recibo se asocie unívocamente a un único mensaje o a un conjunto de ellos, etc.

El ejemplo que trataremos aquí será el desarrollo de todos los participantes —emisor, receptor y canal sincronizador— para un protocolo parada y espera donde la comunicación se realiza en un único sentido (*símplex*) y donde el canal introduce ruido en la transmisión. El hecho de que el canal produzca interferencias afectará a nuestro protocolo, ya que éste deberá prever estas circunstancias y aún así lograr un intercambio de información efectivo. Con el objetivo de simplificar el protocolo, consideraremos que el canal introduce errores de transmisión, aunque no provoca la pérdida de datos, además estos errores sólo ocurren en tramas de información lo suficientemente largas, es decir, podemos obviar sus efectos en aquellos mensajes que sólo marcan las pautas de la comunicación, como es el caso de los acuses de recibo.

17.2 Identificación de eventos en la comunicación

Vamos a suponer que la comunicación se establece entre un ente emisor que se conecta con un receptor para transmitir una trama de información. El receptor, por su parte, simplemente permite el establecimiento de la comunicación y recibe los datos enviados. El conjunto de eventos que permiten el establecimiento del diálogo son los siguientes:

- El proceso transmisor está preparado para enviar una trama de información hacia el receptor, así que solicita a éste el establecimiento de una conexión mediante la primitiva **connect**.

- El proceso receptor, tras recibir la solicitud de una conexión, permite el establecimiento de la misma, utilizando la primitiva **confirm**.

- Una vez establecido el diálogo entre ambas entidades, el transmisor procede a enviar el conjunto de datos, **tx_msg**.

- Cuando el canal permite la transmisión de una trama de datos, en ocasiones introduce algún error que provoca la incorrecta recepción del mismo. Para modelar esta acción utilizamos el evento **error**.

- Tras la recepción de la trama de información el proceso receptor analiza la secuencia y detecta si se ha producido algún error en la transmisión. Si se ha detectado algún error, este proceso avisa al transmisor utilizando la primitiva **tx_nack** para que éste reenvíe los datos, si la transmisión fue correcta, la asiente enviando la primitiva **tx_ack**. A su vez, los eventos que se producen en el transmisor son la recepción del asentimiento (**rx_ack**) o de la solicitud de retransmisión (**rx_nack**).

- Una vez finalizada la transmisión de los datos, el proceso transmisor finaliza la conexión enviando la primitiva **release**.

17.3 Biblioteca de componentes

Antes de describir el proceso de reutilización para la generación de los modelos implicados en el protocolo propuesto, describiremos el estado de la biblioteca de componentes sobre la que se realizará la búsqueda. En la figura 17.1 se muestra el estado inicial de la biblioteca, que consta de ocho grafos, ordenada según el criterio NE^{∞} y, en la figura 17.2 según la relación de orden parcial definida por TC^{∞}. En ambas figuras se han substituido los nombres reales de cada evento por una letra para facilitar la legibilidad de las mismas.

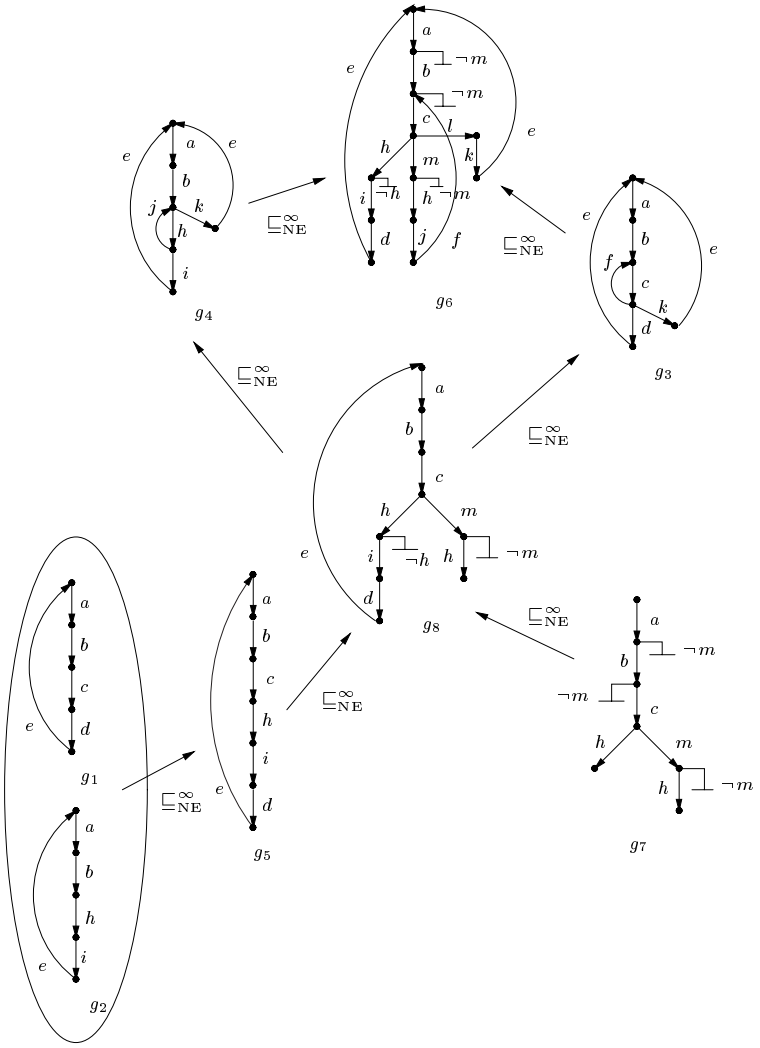


Figura 17.1. Biblioteca de componentes ordenada según el criterio NE^{∞}

La equivalencia entre letras y eventos es la que sigue:

a = **connect**	f = **rx_nack**	k = **t_out**
b = **confirm**	g = **t_out**	l = **loss**
c = **tx_msg**	h = **rx_msg**	m = **error**
d = **rx_ack**	i = **tx_ack**	
e = **release**	j = **tx_nack**	

donde puede verse que, además de los eventos propios de nuestro protocolo de comunicación, existen otros a mayores, propios de los alfabetos de los componentes almacenados.

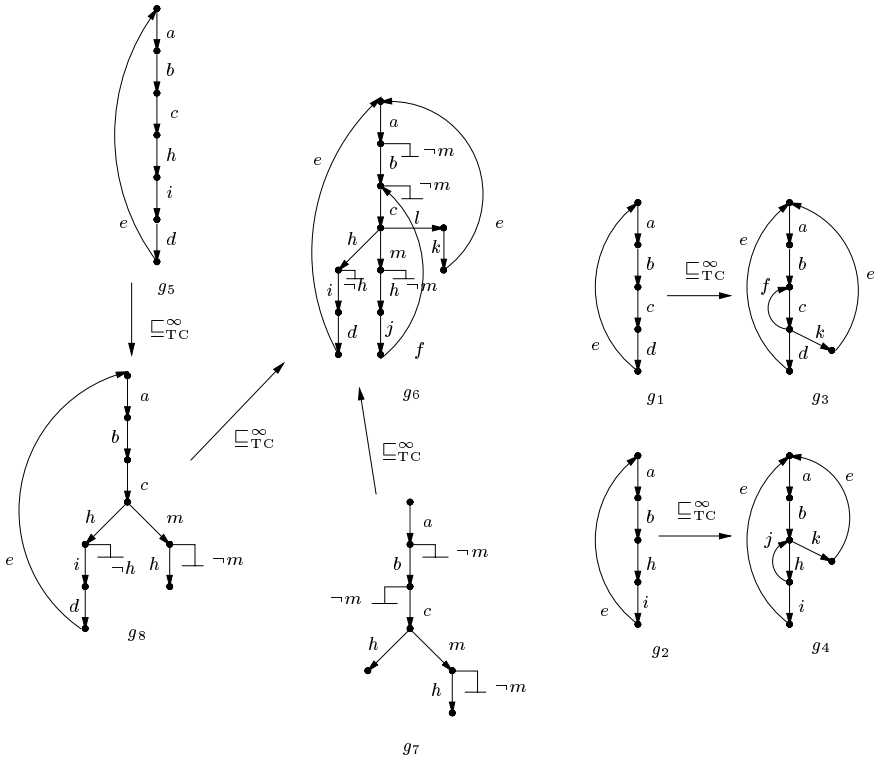


Figura 17.2. Biblioteca de componentes ordenada según el criterio TC^{∞}

17.4 Especificación del proceso emisor

Una vez establecidos los eventos involucrados en el diálogo y teniendo clara la secuencia en la que se pueden producir, se expresa esta sucesión de acciones según la lógica SCTL (apartado 9.1) para cada una de las entidades involucradas. Comenzaremos por el desarrollo del proceso emisor:

$$
\begin{aligned}
R_1 = &\text{ connect } \Rightarrow\bigcirc (\text{confirm } \Rightarrow\bigcirc (\text{tx\_msg } \Rightarrow\bigcirc (\text{rx\_ack} \\
&\Rightarrow\bigcirc (\text{release } \Rightarrow\bigcirc R_1))))\\
R_2 = &\text{ connect } \Rightarrow\bigcirc (\text{confirm } \Rightarrow\bigcirc R_{21})\\
R_{21} = &\text{ tx\_msg } \Rightarrow\bigcirc (\text{rx\_nack } \Rightarrow\bigcirc R_{21})
\end{aligned}
$$

Como lo que nos interesa es localizar un componente funcionalmente próximo, de entre los almacenados en la biblioteca de componentes, se hace imprescindible la obtención de los patrones de búsqueda a partir de la especificación SCTL. De acuerdo con los criterios explicados en los apartados 13.3.1 y 13.3.2 nos bastará con obtener el patrón NE^{∞} para la localización estructural y el patrón TC^{∞} para la localización semántica (apéndice B):

$$TC^{\infty}(R_1, R_2) = ((abcde)+, ab(cf)+, (abcfcde)+, (abc(fc) + de)+)$$

y $NE^{\infty}(R_1, R_2) = (4+, 5+, 7+, 7+)$, donde se han substituido ya los eventos por su letra correspondiente para facilitar la lectura de los siguientes apartados.

Una vez que se han obtenido los patrones de búsqueda en la biblioteca se procederá a recuperar aquellos componentes más próximos estructural y semánticamente.

17.4.1 Recuperación estructural

En la primera fase de este proceso de recuperación se localizarán, en la biblioteca de componentes, aquellos que son sucesores, antecesores y equivalentes al patrón de búsqueda según el criterio NE^{∞}. Tras la búsqueda en la biblioteca tenemos que los componentes sucesores son g_3 y g_4, el componente antecesor es g_8, y no ha sido posible la localización de ningún componente NE^{∞}-equivalente. Los valores de NE^{∞} de cada uno son los que se indican a continuación:

$$
\begin{aligned}
NE^{\infty}(g_8) &= (5, 7+)\\
NE^{\infty}(g_3) &= (4+, 5+, 5+, 7+, 7+, 7+)\\
NE^{\infty}(g_4) &= (4+, 5+, 6+, 6+, 7+, 7+)
\end{aligned}
$$

En la segunda fase de este proceso se calculará la distancia del número total de evoluciones no acotadas entre el patrón de búsqueda $NE^{\infty}(g)$ y los vectores $NE^{\infty}(g_i)$, para cada uno de los componentes recuperados en la primera fase. En

este caso se han obtenido:

$$\begin{aligned}
\mathrm{d}^{\infty}_{\mathrm{NE}}(g, g_8) &= \| \ (5, 4+, 5+, 7+) \ \| = \sqrt{86} \\
\mathrm{d}^{\infty}_{\mathrm{NE}}(g, g_3) &= \| \ (5+, 7+) \ \| = \sqrt{52} \\
\mathrm{d}^{\infty}_{\mathrm{NE}}(g, g_4) &= \| \ (6+, 6+) \ \| = \sqrt{50}
\end{aligned}$$

Concluyendo, entonces, que el componente más próximo, estructuralmente hablando, es g_4.

17.4.2 Recuperación semántica

Para localizar aquellos componentes más próximos semánticamente al patrón requerido se procederá también en dos fases, una primera de localización *gruesa* y una segunda de refinamiento de la búsqueda. En la primera etapa se buscarán los componentes antecesores, sucesores y equivalentes al patrón proporcionado según el criterio TC^{∞}. Para poder cotejar las trazas de evolución no acotadas es necesario conocer los eventos que provocan cada una de las transiciones, eventos que en la figura 17.2 fueron substituidos por letras para una mayor legibilidad. Así que el patrón de búsqueda puede expresarse, bajo esta nueva notación, como:

$$TC^{\infty}(g) = ((abcde)+, ab(cf)+, (abcfcde)+, (abc(fc) + de)+)$$

y, tras la primera fase de la localización semántica, se recuperan los componentes g_1 y g_3 como antecesor y sucesor, respectivamente, del dado:

$$\begin{aligned}
TC^{\infty}(g_1) &= (\ (abcde)+) \\
TC^{\infty}(g_3) &= (\ (abcde)+, ab(cf)+, (abc(cf) + de)+, \\
&\qquad (abcfcde), (abcke)+, (abcfcke)+, (abc(fc) + ke)+)
\end{aligned}$$

En una segunda fase de refinamiento de la búsqueda semántica tendremos que localizar el vector de ajuste funcional entre el patrón y cada uno de los componentes recuperados, para ello es preciso obtener primero el consenso funcional y la adaptación funcional entre ellos (apartado 13.4.2).

Consenso funcional

$$\begin{aligned}
\rho(g, g_1) &= (abcde)+ \\
\rho(g, g_3) &= ((abcde)+, ab(cf)+, (abcfcde)+, (ab(cf) + de)+)
\end{aligned}$$

Déficit funcional

$$\begin{aligned}
\delta(g, g_1) &= (ab(cf)+, (abcfcde)+, (abc(fc) + de)+) \\
\delta(g, g_3) &= \emptyset
\end{aligned}$$

Exceso funcional

$$\varepsilon(g, g_1) = \emptyset$$
$$\varepsilon(g, g_3) = ((abcke)+, (abcfcke)+, (abc(fc) + ke)+)$$

Adaptación funcional

$$\Delta(g, g_1) = (ab(cf)+, (abcfcde)+, (abc(fc) + de)+)$$
$$\Delta(g, g_3) = ((abcke)+, (abcfcke)+, (abc(fc) + ke)+)$$

Vector de ajuste funcional

$$\Theta(g, g_1) = (((abcde)+),\ (ab(cf)+, (abcfcde)+, (abc(fc) + de)+))$$
$$\Theta(g, g_3) = (((abcde)+, ab(cf)+, (abcfcde)+, (ab(cf) + de)+),$$
$$((abcke)+, (abcfcke)+, (abc(fc) + ke)+))$$

(a) Orden parcial (b) Orden total

Figura 17.3. Diagramas de orden definidos por el vector de ajuste funcional

Ordenando los vectores de ajuste funcional obtenidos según la relación de orden parcial definida en el apartado 13.4.2 se obtiene el diagrama Hasse de la figura 17.3(a). Dado que no es suficiente con esta ordenación parcial para llegar a ninguna conclusión, es necesario recurrir a la relación de orden total detallada en la definición 13.1. Según esta última y dado que $\sharp\Delta(g, g_1) = \sharp\Delta(g, g_3)$ y que $\sharp\rho(g, g_3) > \sharp\rho(g, g_1)$ se concluye, como resultado de la búsqueda, que $\Theta(g, g_3) \subseteq \Theta(g, g_1)$, con lo que el componente g_3 es el más parecido semánticamente al patrón solicitado (apartado 13.4.2), tal y como puede verse en la figura 17.3(b).

17.4.3 Adaptación del componente

El usuario deberá decidir entre la adaptación semántica del componente g_3 y la adaptación estructural del componente g_4. En este punto se evidencia una carencia del entorno de reutilización, ya que no proporciona ninguna sugerencia que ayude al usuario a la selección del componente. Tras observación directa de ambos modelos, se puede concluir que el modelo g_4 presenta eventos que no están

presentes en la semántica del proceso que se desea obtener, por lo que, además de posibles adaptaciones en la estructura del grafo, será necesario modificar su semántica, es decir, su alfabeto y las secuencias que sobre él se especifiquen. Así que, la opción parece clara y un usuario experimentado se decantaría por la selección del componente g_3.

Para adaptar dicho componente a la consulta especificada será necesario añadir primero la funcionalidad de la que carezca, $\delta(g, g_3)$, y posteriormente eliminar aquella superflua, $\varepsilon(g, g_3)$. En este caso sólo será necesario reducir funcionalidad, concretamente la especificada por las trazas:

$$\varepsilon(g, g_3) = ((abcke)+, (abcfcke)+, (abc(fc) + ke)+)$$

para ello se procede tal y como se indicó en el capítulo 14, de forma que tras eliminar la primera traza $(abcke)+$, suprimiendo el estado al que el sistema transiciona tras la ocurrencia del evento k y la transición siguiente e, automáticamente se han suprimido las dos trazas restantes, ya que eran consecuencia de la especificación de dicho estado y sus transiciones. Como resultado de este proceso de adaptación se obtiene el componente deseado, representado en la figura 17.4.

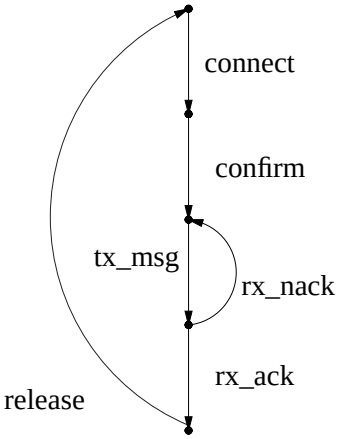


Figura 17.4. Transmisor en un canal con errores, pero sin pérdidas

Al clasificar y almacenar este nuevo componente en la biblioteca el orden se ve alterado tal y como se indica en las figuras 17.5 y 17.6

17.5 Especificación del proceso receptor

A la hora de desarrollar el proceso receptor es necesario expresar su comportamiento bajo la sintaxis de SCTL, de forma que éste será:

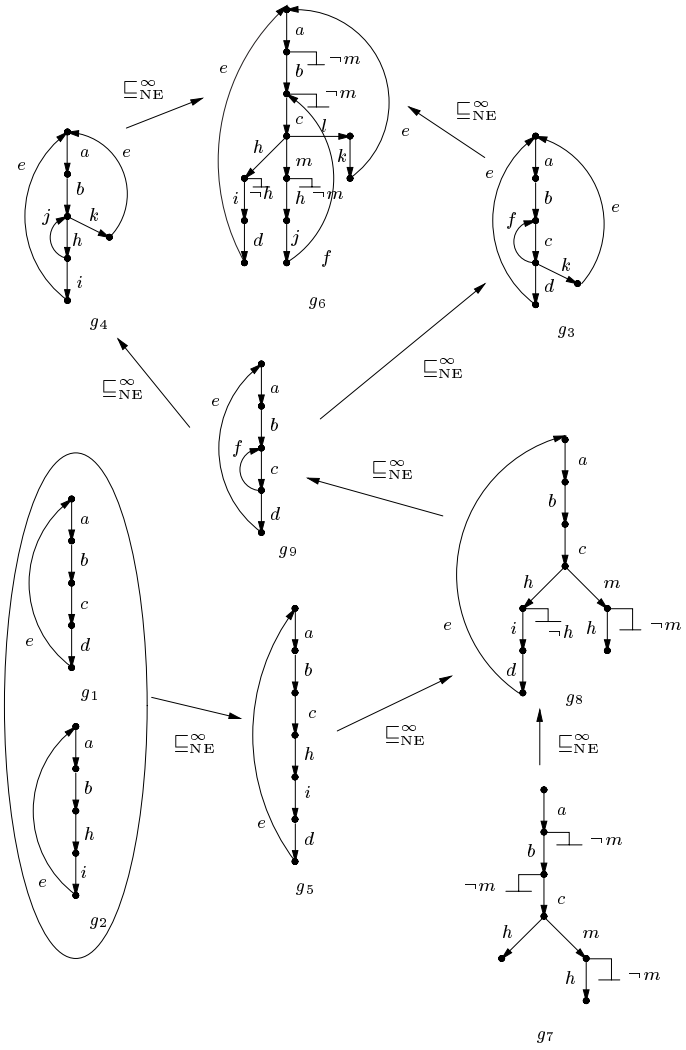


Figura 17.5. Biblioteca de componentes ordenada según el criterio NE^{∞}

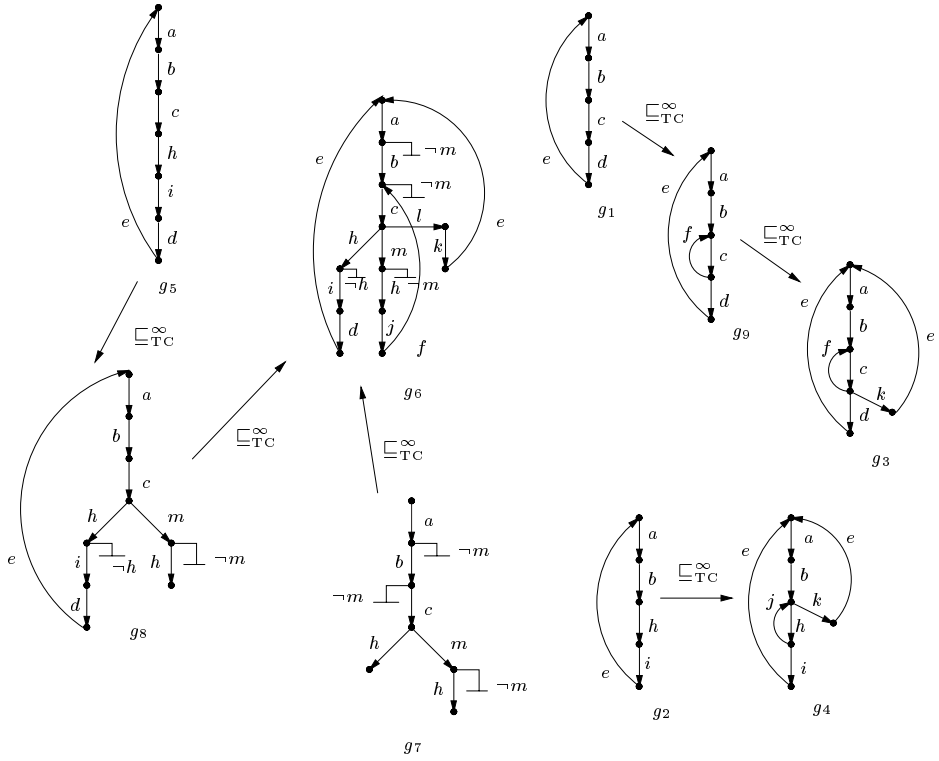


Figura 17.6. Biblioteca de componentes ordenada según el criterio TC^{∞}

$$
\begin{aligned}
R_1 &= \text{connect } \Rightarrow\!\bigcirc (\text{confirm } \Rightarrow\!\bigcirc (\text{rx\_msg } \Rightarrow\!\bigcirc (\text{tx\_ack} \\
&\quad \Rightarrow\!\bigcirc (\text{release } \Rightarrow\!\bigcirc R_1)))) \\
R_2 &= \text{connect } \Rightarrow\!\bigcirc (\text{confirm } \Rightarrow\!\bigcirc R_{21}) \\
R_{21} &= \text{rx\_msg } \Rightarrow\!\bigcirc (\text{tx\_nack } \Rightarrow\!\bigcirc R_{21})
\end{aligned}
$$

Obteniendo los valores de NE^{∞} y TC^{∞} para estos requisitos, igual que se hizo con el proceso transmisor (apartado 17.4) tendremos los siguientes patrones de búsqueda:

$$
\begin{aligned}
TC^{\infty}(R_1, R_2) = \ (\ & (abhie)+, ab(hj)+, \\
& (abhjhie)+, (abh(jh) + ie)+)
\end{aligned}
$$

y $NE^{\infty}(R_1, R_2) = (4+, 5+, 7+, 7+)$. Donde se han substituido ya los eventos de los requisitos por su letra correspondiente, para facilitar la legibilidad.

17.5.1 Recuperación estructural

En esta fase de la localización de componentes se intenta recuperar aquellos que sean sucesores, antecesores y equivalentes según el criterio NE a la consulta. En este caso podemos encontrar directamente que existe un componente (g_3) NE^{∞}-equivalente al dado y éste será el de mayor parecido estructural.

17.5.2 Recuperación semántica

Tras la primera fase de recuperación semántica, se recuperan los componentes g_2 y g_4, antecesor y sucesor respectivamente del patrón de la consulta:

$$
\begin{aligned}
TC^{\infty}(g_2) &= (\ (abhie)+) \\
TC^{\infty}(g_4) &= (\ (abhie)+, ab(hj)+, (abh(jh) + ie)+, \\
&\qquad (abhjhie), (abcke)+, (abhjke)+, (ab(hj) + ke)+)
\end{aligned}
$$

En una segunda fase de refinamiento sería necesario obtener el vector de ajuste funcional entre el patrón y cada uno de los dos elementos recuperados. Procediendo de forma similar a la del apartado 17.4.2 se obtiene que dichos vectores son los siguientes:

$$\Theta(g, g_2) = (((abhie)+),\ (ab(hj)+, (abhjhie)+, (abh(jh) + ie)+))$$
$$\Theta(g, g_4) = (((abhie)+, ab(hj)+, (abhjhie)+, (abh(jh) + ie)+),$$
$$((abke)+, (abhjke)+, (ab(hj) + ke)+))$$

De donde se concluye que $\Theta(g, g_4) \subseteq \Theta(g, g_2)$, con lo que el componente resultante de esta fase de localización es g_4.

17.5.3 Adaptación del componente

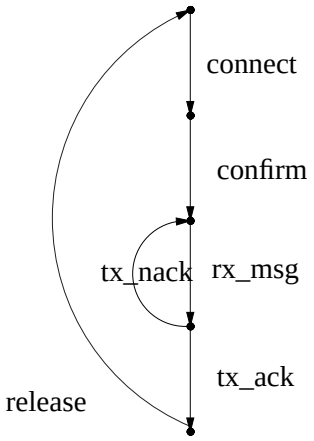


Figura 17.7. Receptor en un canal con errores, pero sin pérdidas

En este caso se le ofrece al usuario la posibilidad de escoger entre el componente g_9, NE^{∞}-equivalente al solicitado, y el componente g_4, el más cercano semánticamente. Dado que uno de ellos es directamente equivalente en estructura, podemos suponer que un usuario familiarizado seleccionaría este último.

La adaptación en este caso es sencilla, ya que directamente consistiría en renombrar los eventos de la siguiente forma:

rx_msg $\leftrightarrow$ tx_msg
tx_ack $\leftrightarrow$ rx_ack
tx_nack $\leftrightarrow$ rx_nack

obteniéndose el componente de la figura 17.7. Al introducir este nuevo componente en la biblioteca el orden se ve alterado tal y como se indica en las figuras 17.8 y 17.9

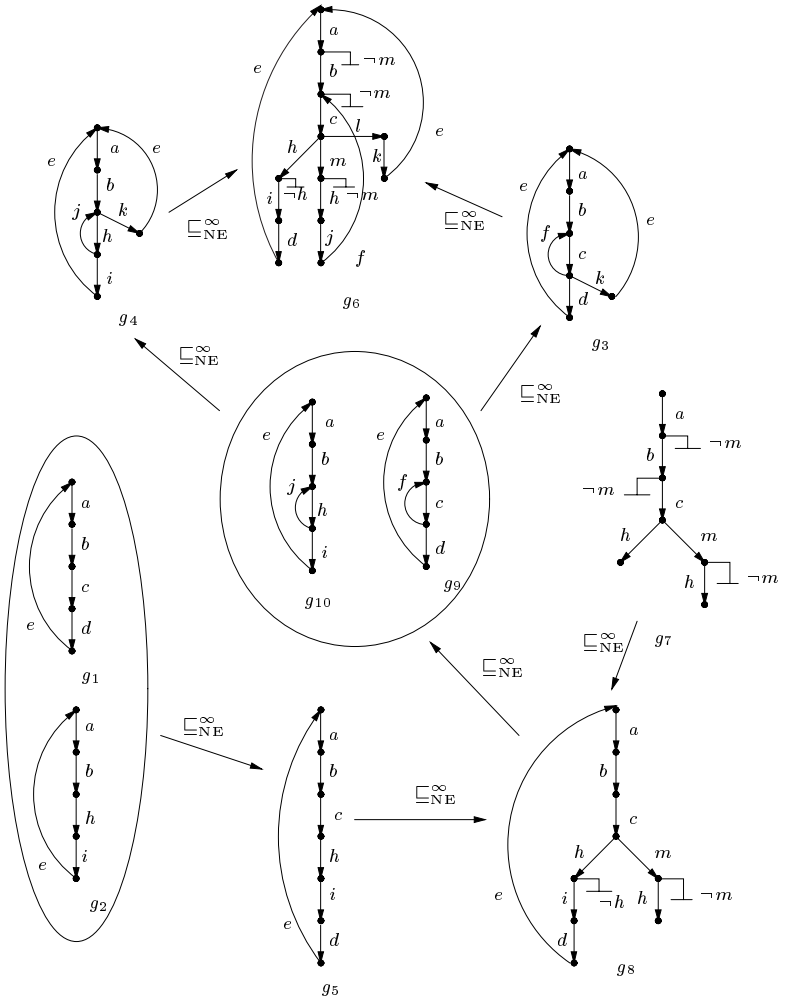


Figura 17.8. Biblioteca de componentes ordenada según el criterio NE^{∞}

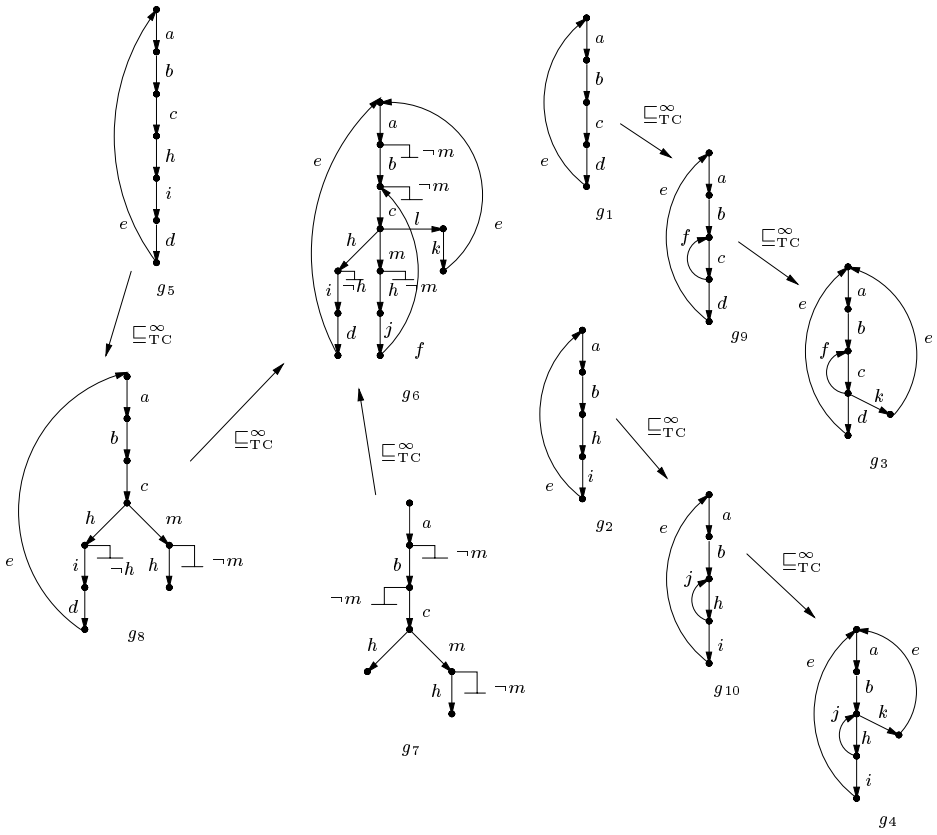


Figura 17.9. Biblioteca de componentes ordenada según el criterio TC^{∞}

17.6 Obtención del sistema completo

Tras la obtención del proceso sincronizador o canal, cuyo procedimiento es idéntico a los seguidos para conseguir los procesos emisor y receptor, se realiza la composición de los tres procesos, utilizando un algoritmo de solapamiento (García-Duque, 2000). En la figura 17.10 se refleja el modelo resultante de esta composición, al que se han añadido, además, los siguientes requisitos funcionales:

$$\begin{aligned} \text{connect} &\Rightarrow \bigcirc\, \neg\text{error} \\ \text{confirm} &\Rightarrow \bigcirc\, \neg\text{error} \\ \text{error} &\Rightarrow \bigcirc\, \neg\text{error} \\ \text{tx\_ack} &\Rightarrow \neg\text{rx\_msg} \end{aligned}$$

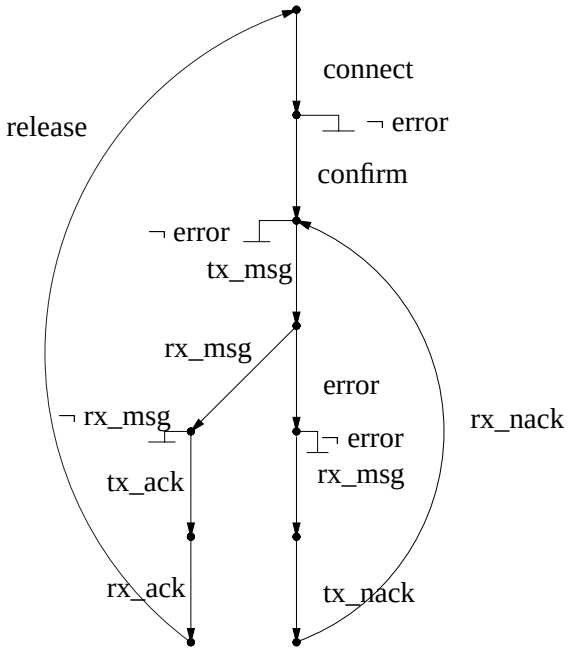


Figura 17.10. Grafo MUS del sistema global

17.7 Verificación de algunas propiedades

Una vez que se dispone del grafo que representa el comportamiento global del sistema, puede ser necesario verificar algunas propiedades para comprobar que la composición de los tres elementos se comporta tal y como se esperaba. En este

punto veremos cómo la reutilización de información de verificación resulta de especial ayuda aplicándola a dos propiedades:

$$R_1 = \text{connect } \Rightarrow\bigcirc \text{ confirm}$$
$$R_2 = \text{rx\_msg } \Rightarrow \text{ error}$$

17.7.1 Verificación de R_1

La primera propiedad, $R_1 = (\text{connect } \Rightarrow\bigcirc \text{ confirm})$ refleja el deseo de que siempre que se produzca una petición de conexión, *connect*, inmediatamente después se produzca una aceptación de dicha conexión, *confirm*. De este modo, lo ideal sería que esta propiedad se verificase en todas las posibles ejecuciones del sistema, es decir, interesa que sea una finalidad universal.

Localización de información

Para poder reutilizar alguna información de verificación localizamos en la biblioteca de componentes (figura 17.9) aquellos componentes antecesores y sucesores al dado donde se ha verificado dicha propiedad, obteniendo como resultado de esta búsqueda los componentes antecesores g_8 y g_7 y el componente sucesor g_6. Los resultados de verificación sobre ellos son los siguientes:

$$\vDash (R_1, g_8) = \{1, 1, \frac{1}{2}, \frac{1}{2}\}$$
$$\vDash (R_1, g_7) = \{1, 1, \frac{1}{2}, \frac{1}{2}\}$$
$$\vDash (R_1, g_6) = \{1, 1, \frac{1}{2}, \frac{1}{2}\}$$

Donde, además, sabemos que $\vDash (R_1, E_0|_{g_8}) = 1$ y que $\vDash (R_1, E_0|_{g_7}) = 1$.

Extracción de conclusiones

Partiendo de la información disponible, se deducen los siguientes grados de satisfacción para R_1 sobre nuestro grafo g:

1. Dado que $\vDash (\exists \Diamond R_1, g_8) = 1$ y $\vDash (\exists \Diamond R_1, g_7) = 1$, entonces sabemos que $\vDash (\exists \Diamond R_1, g) = 1$.
2. Como $\vDash (R_1, E_0|_{g_8}) = \vDash (R_1, E_0|_{g_7}) = 1$, entonces podemos deducir que $\vDash (\forall \Diamond R_1, g) = 1$.

De los resultados de verificación sobre g_6 podemos deducir que:

1. Dado que $\vDash (\forall\, \Box R_1, g_5) = \frac{1}{2}$, entonces $\vDash (\forall\, \Box R_1, g) \leq_{\mathrm{c}} 1$.

Así que, como resultado de toda esta información disponible tendremos la información $\vDash (R_1, g) = \{1, 1, \phi_1, \phi_2\}$, donde se satisface que $\phi_2 \in \{\frac{1}{2}, \frac{3}{4}, 1\}$, y dada la relación de orden parcial entre los cuatro resultados de verificación almacenados sabemos que $\phi_2 = \vDash (\forall\, \Box R_1, g) \leq \vDash (\exists\, \Box R_1, g) = \phi_1$, con lo que $\phi_1 \in \{\frac{1}{2}, \frac{3}{4}, 1\}$:

$$\vDash (R_1, g) = \{1, 1, \phi_1 \in \{\frac{1}{2}, \frac{3}{4}, 1\}, \phi_2 \in \{\frac{1}{2}, \frac{3}{4}, 1\}\}$$

De forma que podemos concluir que la propiedad R_1 es una finalidad universal sobre g —se satisface en todas las posibles vías de evolución del sistema— y, aunque no se puede considerar una invarianza existencial ni universal, sí puede considerarse una potencial invarianza existencial y universal, es decir, incrementando la especificación sobre el grafo sí podría llegar a cumplirse.

17.7.2 Verificación de R_2

La propiedad $R_2 = (\text{rx\_msg} \Rightarrow \text{error})$ expresa la posibilidad de que en un mismo estado del modelo se produzca una recepción de datos, *rx_msg*, y un error en la transmisión, *error*. Esta propiedad debería satisfacerse en todas las ejecuciones del sistema, ya que siempre que se produzca un envío de datos es posible la existencia de errores en el canal, es decir, sería deseable que estuviésemos ante una finalidad universal. Además es necesario notar que tras producirse un error de transmisión se producirá la recepción del mensaje y no debería permitirse en este caso la repetición de un error en la transmisión, ya que no se ha duplicado ésta, por lo que sería deseable que no fuese posible que esta propiedad alcanzase el grado de invarianza universal.

Localización de información

La localización de componentes es idéntica a la que se realizó para la propiedad anterior, con lo que obtenemos los componentes antecesores g_8 y g_7 y el componente sucesor g_6. En ellos los resultados de verificación son los siguientes:

$$\vDash (R_2, g_8) = \{1, 1, \frac{1}{4}, 0\}$$
$$\vDash (R_2, g_7) = \{1, 1, \frac{1}{4}, 0\}$$
$$\vDash (R_2, g_6) = \{1, 1, \frac{1}{4}, 0\}$$

Extracción de conclusiones

Partiendo de la información disponible sobre g_7 y g_8 podemos deducir los siguientes grados de satisfacción para R_2 sobre nuestro grafo g:

1. Dado que $\vDash (\exists \Diamond R_2, g_8) = \vDash (\exists \Diamond R_2, g_7) = 1$, entonces sabemos que $\vDash (\exists \Diamond R_2, g) = 1$.
2. Como $\vDash (\exists \Box R_2, g_7) = \vDash (\exists \Box R_2, g_8) = \frac{1}{4}$, entonces sabemos que se cumple $\vDash (\forall \Box R_2, g) \geq_c \frac{1}{4}$, es decir su valor se encuentra restringido al conjunto $\{\frac{1}{4}, \widehat{\frac{1}{2}}, 0\}$. Como $\vDash (\forall \Box R_2, g_8) = \vDash (\forall \Box R_2, g_7) = 0$, entonces podemos concluir que $\vDash (\forall \Box R_2, g) = 0$.

Sin embargo la información de verificación de R_2 sobre g_6 no aporta ningún dato a mayores. Así que los valores de la cuádrupla $\vDash (R_2, g)$ que hemos podido obtener de la reutilización de información de verificación son los siguientes:

$$\vDash (R_2, g) = \{1, \phi_1 \in \Phi, \phi_2 \in \Phi, 0\}$$

Según ellos sabemos que R_2 es una finalidad existencial sobre g y que, además, no llega a alcanzar el grado de invarianza universal. De estas dos conclusiones se deduce que el modelo del sistema se comporta tal y como se deseaba respecto al hecho de que no llegase a ser invarianza universal, sin embargo no hemos podido averiguar si llega a ser una finalidad universal. Sabemos que sí es una finalidad existencial, es decir, por al menos una de las posibles ejecuciones, el modelo g satisface la propiedad R_2, pero no es posible saber, sólo con esta información, si la satisface en todas y cada una de las posibles vías de evolución. Para ello sería necesario analizar más detenidamente el modelo g.

PARTE VII

Conclusiones y líneas futuras

CAPÍTULO 18

Conclusiones y líneas de trabajo futuro

El trabajo de esta tesis establece las bases formales que permiten la reutilización de modelos, posiblemente incompletos, de sistemas software y, además, la reutilización de la información de verificación asociada a los mismos. En este capítulo se realiza un breve compendio de las distintas líneas de trabajo seguidas, recapitulando los distintos objetivos logrados en cada una. Por último, se esbozan las líneas de trabajo futuro que se han identificado como consecuencia de las bases sentadas por el presente trabajo.

18.1 Conclusiones

El trabajo expuesto permite la reutilización de modelos completos o incompletos de sistemas, y de su información de verificación asociada, dentro de un modelo de desarrollo ya existente, cuya principal característica es la total formalización. Con este objetivo se han desgranado a lo largo de este documento las bases necesarias para proporcionar un entorno de reutilización adaptado a la situación de partida que, básicamente, se pueden organizar en tres grandes líneas:

– la gestión de la biblioteca de componentes;

– la recuperación de modelos, completos o incompletos, almacenados en dicha biblioteca con la intención de ser utilizados como punto de partida para el desarrollo de nuevos sistemas; y

– la recuperación de información de verificación que alivie la carga computacional de un algoritmo de verificación basado en técnicas de *model checking*.

18.1.1 Gestión de la biblioteca de componentes

La gestión de la biblioteca de componentes es vital para permitir la eficacia y eficiencia necesarias en la localización de componentes potencialmente reutilizables, así que una buena clasificación y ordenación de los mismos en la base de datos facilitará en gran medida este proceso de búsqueda.

Se optó por una clasificación basada en la representación formal de los componentes reutilizables básicamente por dos razones: de esta forma la integración de todo el entorno en el proceso de desarrollo de partida se facilita en gran medida; y, dado que se trata de reutilizar componentes de un elevado nivel de abstracción —expresados ya bajo un modelo formal—, su propio contenido podía ser utilizado como índice de clasificación en la base de datos.

Clasificación de componentes reutilizables

Dado que se persigue la recuperación de componentes cuya funcionalidad sea cercana a la solicitada, la consecuencia directa es que la ordenación de los componentes de la biblioteca debería hacerse atendiendo a criterios de proximidad funcional. Con este propósito fueron identificados cuatro criterios diferentes que permiten la ordenación parcial de componentes y el establecimiento de distintas relaciones de equivalencia de funcionalidad.

Los criterios definidos se basan en la semántica de trazas, ya que éstas son una aproximación fiable de la funcionalidad expresada en cada componente. Aunque se podría haber optado por otras relaciones más exactas (como relaciones de simulación y bisimulación más estrictas), éstas ralentizaban la obtención de los patrones de clasificación y búsqueda, y, dado que la eficiencia es crucial en cualquier entorno de reutilización, fueron descartadas en favor de las relaciones actuales. Los criterios adoptados pueden ser divididos, básicamente, entre aquellos que evalúan la similitud estructural entre modelos —número de evoluciones y número de evoluciones no acotadas— y la similitud semántica entre los mismos —trazas completas y trazas completas no acotadas.

Recuperación de componentes reutilizables

La recuperación de componentes se planteó en todo momento como una recuperación aproximada, ya que cualquier localización exacta implicaba la intervención de algún proceso de verificación formal, lo que contravenía directamente los objetivos de la tesis. Esta recuperación aproximada permite la localización eficiente de un subconjunto de componentes cercanos a la funcionalidad deseada y

que después son filtrados, aplicando criterios de cuantificación de funcionalidad, con el objetivo de extraer aquél o aquéllos más apropiados a los intereses del usuario. Así que, además de aproximada, la recuperación de componentes se plantea como un proceso escalonado, con una primera etapa de localización gruesa y una posterior de refinamiento de la búsqueda.

La meta de la primera etapa de localización es la recuperación de un conjunto de componentes reutilizables funcionalmente próximos a la consulta realizada, es decir, componentes cuya funcionalidad está contenida en la deseada, o bien esta última está totalmente expresada en ellos. Como resultado se obtienen aquellos componentes cuya funcionalidad establece cotas inferiores, superiores y equivalentes a la pedida, según el criterio de cotejo seleccionado. Estos resultados dependen claramente del tamaño de la biblioteca, ya que cuando mayor sea éste, mayor será la probabilidad de que los componentes recuperados en esta fase estén, realmente, cercanos en funcionalidad a la consulta.

En la segunda fase de refinamiento de la búsqueda, es necesario dilucidar cuál, de entre los primeramente seleccionados, será el componente que se ofrece al usuario como el más adecuado a sus intereses. Para ello no basta con las relaciones de tipo *contenido-continente* utilizadas para la primera fase, sino que es necesario cuantificar cuál, de entre ellos, es aquél cuya funcionalidad expresada es objetivamente más próxima a la pedida. Con este fin se han definido dos métricas que permiten evaluar distancias estructurales —distancia del número de evoluciones totales y distancia del número de evoluciones totales no acotadas—, y un conjunto de criterios que cuantifican las diferencias semánticas entre componentes —intersección funcional, exceso de especificación, defecto en la misma, etc— facilitando así la reordenación de este subconjunto de componentes en función del patrón de funcionalidad buscado.

Con esta recuperación escalonada se ha conseguido fusionar dos tendencias bien distintas: la ordenación estática o independiente de la consulta presente en la primera fase, menos eficaz; y la ordenación dinámica o totalmente dependiente de la consulta presente en la segunda, mucho más eficaz pero menos eficiente, consiguiendo así que prevalezcan las mejores características de cada técnica, obteniéndose buenos resultados en tiempos aceptables.

Una vez seleccionado el componente óptimo, será necesario adaptar su funcionalidad para que satisfaga la solicitada. Para ello se añaden aquellas trazas de evolución de las que carece y se eliminan las superfluas. El principal problema que presenta la adaptación de componentes, reutilización de *cajas blancas*, es el mantenimiento de la información de verificación asociada al componente reutilizado. En este trabajo sólo se ha estudiado el mantenimiento de la información de verificación entre componentes TC^{∞}-relacionados, así que en el caso de que la adaptación de un componente implique la obtención de otro relacionado con él según TC^{∞}, las conclusiones obtenidas para la reutilización de verificación se mantendrían. Sin embargo no se ha estudiado el impacto general que suponen las

tareas de adaptación en cuanto a la vigencia de la información de verificación, y sería interesante realizar un estudio exhaustivo que permitiese conocer la validez de dichos resultados en componentes adaptados.

18.1.2 Reutilización de información de verificación

Otro de los puntos clave del proceso de desarrollo de sistemas es la verificación de propiedades, llevada a cabo en el modelo de ciclo de vida de partida mediante técnicas de *model checking* aplicadas a propiedades expresadas en SCTL sobre modelos MUS de sistemas. Dada la elevada carga computacional requerida, se propone la reutilización de la información de verificación asociada a cada componente reutilizable como una forma de aliviar su impacto.

Tras estudiar la información de verificación disponible —grados de satisfacción de una propiedad sobre todos y cada uno de los estados de un modelo del sistema— se han sentado las bases que permiten resumir esta información y extraer los grados de satisfacción de dicha propiedad sobre todo el modelo. Así se obtienen cuatro valores, que expresan los grados de satisfacción de una propiedad, pudiendo conocer si ésta puede ser considerada una finalidad existencial o universal sobre el modelo, o bien una invarianza existencial o universal sobre el mismo. Además se ha concluido que es interesante, también, mantener la información de verificación de la propiedad sobre el estado inicial del sistema, motivado precisamente por sus peculiaridades.

Estos grados de satisfacción, provenientes de la verificación SCTL-MUS, no sólo mantienen información sobre la verificación o no de una propiedad, sino que, debido a la introducción del concepto de subespecificación que se realiza en esta metodología, permiten establecer otros cuatro grados de satisfacción intermedios. Éstos guardan información sobre la potencial evolución hacia la total satisfacción o la no satisfacción de una propiedad tras el incremento de la funcionalidad descrita en el modelo en iteraciones futuras.

Una vez compendiada la información de verificación sobre cada grafo o componente, se ha estudiado cómo esta puede ser reutilizada entre componentes relacionados por el criterio de trazas completas y trazas completas no acotadas. En este caso se han obviado las relaciones de inclusión por estructura ya que la semántica es indispensable en la expresión de propiedades. De esta forma se permite la extracción de conclusiones sobre el grado de verificación de una propiedad en el componente pedido a partir de los grados de satisfacción de dicha propiedad en otros componentes *contenidos* y *continentes* del dado.

18.1.3 Implementación

Paralelamente a toda la labor teórica descrita se ha realizado una labor de implementación de aquellos algoritmos necesarios, con el objetivo de comprobar la

validez del entorno de clasificación y recuperación propuestos. A lo largo de este documento se ha reiterado la necesidad de una implementación eficiente de los algoritmos de manejo de los componentes reutilizables y, aunque en principio el lenguaje C parece el más adecuado en este sentido, se optó finalmente por una implementación utilizando el lenguaje C++, ya que el paradigma de orientación a objetos resultó más apropiado en este entorno. Esta elección vino también condicionada por la posibilidad de incorporar la biblioteca de algoritmos y estructuras de datos LEDA (L*ibrary of* E*fficient* D*ata* T*ypes and* A*lgorithms*) (Mehlhorn y Maher, 1999), ya que ésta presenta innumerables ventajas a la hora de trabajar con grafos y estructuras de datos complejas de una forma eficiente.

Los algoritmos básicos, sobre los que se asientan todos los demás, son aquellos que permiten la obtención de los patrones de búsqueda, según los cuatro criterios definidos, a partir de un grafo MUS y a partir de un conjunto de requisitos funcionales expresados en SCTL. Debido a las relaciones existentes entre los cuatro patrones, se optó por extraer sólo uno a partir del grafo, el más completo —trazas completas no acotadas—, y a partir de él obtener los tres restantes. Estos algoritmos se han optimizado en la medida de lo posible para evitar los recorridos superfluos del grafo.

Otros algoritmos cruciales son aquellos que permiten la obtención de redes de componentes, organizadas según las cuatro relaciones de orden parcial definidas. De esta forma se permite la clasificación y el almacenamiento de un componente en la biblioteca sin necesidad de realizar una confrontación con todos y cada uno de los ya existentes y, aún así, asegurar su correcto posicionamiento relativo al resto. Estos algoritmos de clasificación son los aplicados en la primera fase de recuperación aproximada de componentes próximos funcionalmente a uno requerido. Sin embargo, para la localización refinada, es preciso la implementación de otros algoritmos, algoritmos que permitan la intersección de trazas completas y trazas completas no acotadas y la evaluación de diferencias de funcionalidad entre ellas, excesos y defectos funcionales. La reordenación del conjunto de componentes en esta segunda fase, se realiza utilizando los mismos algoritmos que para la clasificación.

A la hora de adaptar un componente a la funcionalidad requerida en el patrón de búsqueda se necesitó implementar un algoritmo que permitiese la inclusión de nuevas vías de evolución y otro que permitiese la eliminación de trazas, de forma que como resultado se obtenga un grafo totalmente ajustado a lo solicitado.

En cuanto a la reutilización de información de verificación, se ha implementado un algoritmo que permite cotejar los grados de satisfacción disponibles de una propiedad sobre distintos componentes contenidos en el dado, y que también coteja los grados de satisfacción disponibles de la misma propiedad sobre distintos componentes contenedores del dado. Como resultado se proporciona al usuario toda la información de verificación que es posible concluir a partir de los datos almacenados en la biblioteca.

La incorporación de las estructuras de datos facilitadas por LEDA permitieron también la elaboración de una interfaz gráfica que facilita al usuario la interactuación con la herramienta, proporcionando un acceso sencillo e intuitivo a la misma, facilitando la visualización de grafos, las visualización de las distintas relaciones de orden parcial entre componentes almacenados, etc.

18.2 Líneas de trabajo futuro

Aunque las líneas de trabajo futuro que se detallan en este apartado son de naturaleza diversa pueden ser englobadas en dos grandes categorías: aquellas que están íntimamente ligadas con el trabajo descrito y que permiten ahondar en algunas tareas inconclusas; y aquellas que pueden crecer paralelamente a éste.

18.2.1 Continuación del trabajo

- Una de las ideas más interesantes para desarrollar es el concepto de *búsqueda composicional*. Tal y como está planteado el entorno en la actualidad sólo se localiza a aquellos componentes próximos funcionalmente al patrón pedido. Sería interesante la posibilidad de fragmentar la búsqueda de forma que si no es posible localizar un único componente, se intente la localización de subcomponentes. Para ello sería necesario establecer los mecanismos que permitan la fragmentación de la búsqueda y su posterior composición, manteniendo la funcionalidad en la composición de los componentes localizados.
  Esta misma filosofía del *divide y vencerás* podría aplicarse a la reutilización de información de verificación. Sería útil, en algunos casos, desglosar la propiedad a verificar en subpropiedades de las que sea más sencillo encontrar información de verificación. Posteriormente, se analizaría cómo componer los resultados obtenidos para concluir los grados de satisfacción de la propiedad completa a partir de los grados de satisfacción de sus subpropiedades.

- Tal y como se ha introducido en las conclusiones, en este trabajo no se ha estudiado el impacto que presentan las diferentes tareas de adaptación sobre la información de verificación asociada a un componente reutilizado. Sería conveniente analizar qué información de verificación se mantiene, aún tras la adaptación, y cuál no.

- Profundización en el estudio de las relaciones estructurales. En el estado actual prácticamente no se aprovechan las relaciones de proximidad estructural para la reutilización de componentes ni para la reutilización de su información de verificación. Sin embargo la posibilidad de adaptar la semántica requerida a la estructura de los componentes recuperados, es una

posibilidad muy potente que merecería la pena ser contemplada y estudiada con más detalle.

- Otra línea de acción deseable sería la posibilidad de evaluar de forma no relativa la conveniencia de reutilizar o no un componente. Los criterios de cotejo propuestos permiten decidir entre un conjunto de componentes potenciales a aquél o aquéllos más apropiados, pero no permiten cuantificar de forma absoluta si compensa o no adaptar dichos componentes para que satisfagan la funcionalidad requerida. En la actualidad esta decisión se deja en manos del usuario, pero sería conveniente automatizar el proceso definiendo umbrales de decisión —posiblemente basados en heurísticos— y facilitar la decisión al usuario, aportando sugerencias sobre las tareas de adaptación precisas para los componentes.

- Sería interesante realizar un estudio exhaustivo que relacione los beneficios obtenidos al reutilizar los componentes de este entorno, frente a los costes asociados a todo el proceso de reutilización. Sería preciso, entonces, analizar los costes computacionales de localización y adaptación, frente a los costes de síntesis y verificación tradicionales; y analizar los costes de mantenimiento del propio entorno —de creación de los componentes reutilizables, mantenimiento de la base de datos, etc. También habría que cuantificar los beneficios obtenidos con la reutilización, este análisis es complejo ya que no basta con la contabilización de los beneficios inmediatos —evitar tareas de síntesis o verificación formal— sino que habría que tener en cuenta los beneficios futuros —evitar futuras tareas de verificación formal debido a la información de verificación aneja al componente recuperado.

- Por último, y no menos importante, la ampliación de la implementación gráfica, que aunque en la actualidad cubre la funcionalidad básica del entorno de reutilización, debería ser integrada en la herramienta gráfica global del proceso de diseño y desarrollo de sistemas. Directamente relacionado con este trabajo, sería conveniente la definición global de la base de datos, posiblemente distribuida, que permita el almacenamiento de todo el conocimiento disponible en el entorno de desarrollo: modelos incompletos y completos de sistemas, documentación asociada, transformaciones y pruebas, etc.

18.2.2 Líneas complementarias

- El entorno de reutilización por composición propuesto se centra en la primera fase del proceso de desarrollo de sistemas software, pero no interviene en la fase de refinamientos sucesivos en el entorno transformacional LIRA. Intentando ser más ambiciosos en nuestro entorno de reutilización, podrían propiciarse las bases que permitan la reutilización de elementos típicos en

este tipo de entornos transformacionales, obteniendo un entorno de reutilización por generación. La reutilización de refinamientos y optimizaciones podría realizarse definiendo una jerarquía de almacenamiento apropiada para estos nuevos elementos.

- Los componentes reutilizables que tratamos en este entorno permiten especificar la ordenación temporal de eventos del sistema, sin embargo no es posible modelar el intercambio de información. Se propone la ampliación del entorno de reutilización de forma que se permita el cotejo de funcionalidad entre componentes que sí permitan el intercambio de datos. El trabajo desarrollado por (Gil-Solla, 2000) establece el punto de partida para esta línea de trabajo.
  Además, siguiendo con la flexibilización en la definición de componentes reutilizables, podría incluirse la reutilización de modelos con requisitos temporales. En este ámbito, (Fernández-Vilas, 2002) propone un formalismo que permite la especificación de este tipo de requisitos con restricciones de tiempo, y la síntesis de los mismos con el fin de presentar al usuario un prototipo adecuado del sistema.

- En el entorno de trabajo actual, se parte de la especificación formal de requisitos funcionales formulada en SCTL. Sin embargo, esta especificación formal podría combinarse con descripciones textuales que identificasen el entorno de aplicación o ámbito en el que el usuario está trabajando. El mantenimiento de estas descripciones textuales permitiría ampliar en una fase más el entorno de reutilización descrito, conjugando diferentes técnicas de gestión de la base de datos: por una parte la gestión formal descrita en esta tesis, y, por otra parte, una gestión basada en el análisis textual de los componentes. De esta forma sería posible identificar, a partir de la descripción textual proporcionada por el usuario, la familia de aplicaciones sobre la que va a trabajar, limitando así las labores de localización y clasificación en la biblioteca de componentes, al subconjunto formado por los componentes reutilizables pertenecientes al ámbito de la aplicación —conocido como reutilización vertical (apartado 1.3.1).

- Cuando se trata de desarrollar aplicaciones complejas, éstas son normalmente abordadas por diferentes agentes. Estos usuarios enriquecen la fase de identificación de requisitos proporcionando diferentes perspectivas del sistema (*viewpoints*). Tras una primera fase de descripción funcional, es necesario analizar todas las especificaciones proporcionadas por los agentes y establecer una segunda fase de negociación, donde todos ellos llegarán a un acuerdo sobre la funcionalidad final del sistema a desarrollar (García-Duque y Pazos-Arias, 2001). El entorno de reutilización descrito en esta tesis es especialmente útil en esta mecánica de trabajo, ya que no sólo se incrementará la probabilidad de localizar información apropiada en la base

de datos, sino que, combinando la gestión semántica de la base de datos tal y como se describió en el punto anterior, los agentes podrían partir directamente de los modelos funcionales desarrollados por cualquier otro agente involucrado en la especificación del sistema, reduciendo aún más los costes de localización y síntesis.

PARTE VIII

Apéndices

CAPÍTULO A

Funcionalidad de grafos MUS

A.1 Introducción

Es objetivo de este apéndice explicitar el pseudocódigo de los algoritmos utilizados para extraer las informaciones NE, NE^{∞}, TC y TC^{∞} de los grafos MUS. Dado el coste computacional requerido para recorrer los grafos y extraer estas informaciones y, dadas las relaciones que se han identificado entre los cuatro criterios, se ha optado por recorrer el grafo una única vez, para poder extraer la información TC^{∞} y, a partir de ella, deducir las otras tres. Así obtenemos toda la información precisa sin que por ello se vea afectada negativamente la eficiencia.

En todos los pseudocódigos se ha procurado primar la legibilidad y no explicitar las particularidades de cada una de las estructuras utilizadas para almacenar todas y cada una de las informaciones precisas.

A.2 Algoritmo de obtención de la información TC^{∞} dado un grafo MUS

En este apartado se detalla el pseudocódigo del algoritmo recursivo, algoritmo A.1, utilizado para obtener el conjunto de trazas completas no acotadas $TC^{\infty}(g)$ a partir de un grafo MUS g (definición 10.33).

Este algoritmo consigue extraer la secuencia de eventos posibles y no posibles que constituyen las diferentes trazas del grafo, además identifica las recursiones y almacena, para cada traza, la secuencia, o secuencias, que constituyen el bucle de comportamiento. Toda esta información se almacena en la estructura $TC^{\infty}(g)$ que recibe como parámetro y que se va modificando en las sucesivas iteraciones del algoritmo.

Algoritmo A.1 Obtención de TC^{∞} ($TC^{\infty}(g)$, nodo_actual)

– Acceder a la información que mantiene almacenada el nodo_actual. {tratamiento de las acciones negadas del nodo y de las acciones posibles}
si (nodo_actual tiene acciones no posibles) **entonces**
  si ($TC^{\infty}(g)$ no está vacío) **entonces**
    – Extraer la última traza almacenada en $TC^{\infty}(g)$ y guardarla en otra variable (traza_actual).
  fin de la condición
  – Conservar el contenido de traza_actual en otra variable traza_nueva.
  para todo (evento en el conjunto de acciones falsas de nodo_actual) **entonces**
    – Concatenar la secuencia almacenada en traza_nueva con la secuencia "¬ evento".
    – Almacena el contenido de la variable traza_nueva en $TC^{\infty}(g)$.
  fin de la iteración
  – Almacena el contenido de la variable traza_actual en $TC^{\infty}(g)$.
fin de la condición
– Obtener el conjunto de nodos siguientes a nodo_actual almacenándolos en la variable nodos_siguientes.
si (nodos_siguientes está vacía) **entonces**
  – Fin de la iteracción.
fin de la condición
para todo (nodo en nodos_siguientes) **entonces**
  si ($TC^{\infty}(g)$ no está vacía) **entonces**
    – Extraer la última traza de $TC^{\infty}(g)$ y almacenarla en la variable traza_actual.
  fin de la condición
  – Conservar la traza almacenada en traza_actual en la variable traza_nueva.
  – Concatenar el evento que produce la transición desde nodo_actual hasta nodo al final de la traza almacenada en traza_nueva.
  – Añadir la traza modificada traza_nueva a $TC^{\infty}(g)$.
  – Llamada recursiva a $TC^{\infty}(g)$: $TC^{\infty}(TC^{\infty}(g)$, nodo).
  si (queda algún nodo en nodos_siguientes) **entonces**
    – Almacena la traza almacenada en traza_actual.
  fin de la condición
fin de la iteración

A.3 Algoritmo de obtención de la información NE^{∞} dada la información TC^{∞}

En este apartado se detalla el pseudocódigo del algoritmo (algoritmo A.2) utilizado para obtener el conjunto del número de transiciones no acotadas de un grafo MUS g (definición 10.19). Debido a la relación de orden que existe entre el criterio TC^{∞} y NE^{∞} puede extraerse directamente esta última sin más que recorrer cada una de las trazas no acotadas del grafo y computar el número de evoluciones de que consta, teniendo siempre en cuenta si estamos ante una recursión o no.

Algoritmo A.2 Obtención de $NE^{\infty}(g)$ partiendo de $TC^{\infty}(g)$

para todo (traza en $TC^{\infty}(g)$) **entonces**
– Continuar con la ejecución si traza contiene algún evento especificado como no posible.
– Computar el número de eventos posibles en dicha traza y almacenar el resultado en $NE^{\infty}(g)$.
– Si estamos ante una traza recursiva, almacenarlo en $NE^{\infty}(g)$.
fin de la iteración
– Devolver $NE^{\infty}(g)$.

A.4 Algoritmo de obtención de la información TC dada la información TC^{∞}

En este apartado se detalla el pseudocódigo del algoritmo (algoritmo A.3) utilizado para obtener el conjunto de trazas completas de todas y cada una de las vías de evolución de un grafo MUS g (definición 10.26). Al igual que para la obtención de la información $NE^{\infty}(g)$, en este caso TC puede extraerse directamente del resultado $TC^{\infty}(g)$. Para ello se recorren todas y cada una de las trazas almacenadas en $TC^{\infty}(g)$ y se extrae la secuencia de eventos posibles sin tener en cuenta las recursiones que se produzcan.

Algoritmo A.3 Obtención de $TC(g)$ partiendo de $TC^{\infty}(g)$

para todo (traza en $TC^{\infty}(g)$) **entonces**
– Eliminar la información de recursividad que pueda tener.
– Almacenar la traza resultante en la estructura $TC(g)$.
fin de la iteración
– Devolver $TC(g)$.

A.5 Algoritmo de obtención de la información NE dada la información NE^{∞}

En el algoritmo A.4 se detalla el pseudocódigo del algoritmo utilizado para obtener el conjunto del número de transiciones de todas y cada una de las vías de evolución de un grafo MUS g (definición 10.12). Esta información puede obtenerse a partir de la almacenada en NE^{∞} sin más que eliminar de esta última la información relativa a las recursiones.

Algoritmo A.4 Obtención de $NE(g)$ partiendo de $NE^{\infty}(g)$

para todo (cómputo en $NE^{\infty}(g)$) **entonces**
– Eliminar la información relativa a las posibles recursiones.
– Almacenar el resultado en $NE(g)$.
fin de la iteración
– Devolver $NE(g)$.

CAPÍTULO B

Funcionalidad de requisitos SCTL

B.1 Introducción

Al comienzo del diseño de un sistema, el usuario especifica un conjunto de requisitos funcionales que, en nuestro proceso de desarrollo, son expresados en la sintaxis SCTL. La síntesis de estos requisitos permite obtener un modelo MUS de comportamiento del sistema o prototipo inicial. Con el entorno de reutilización propuesto se persigue reducir o eliminar estas tareas de síntesis inicial mediante la localización de algún modelo funcionalmente semejante a lo solicitado por el usuario.

Las tareas de búsqueda y recuperación de componentes reutilizables (ver capítulo 13) se basan en el cotejo de la funcionalidad estructural y semántica expresada bajo las funciones NE, NE^{∞}, TC y TC^{∞}, que definen el perfil o patrón de los componentes. Así que resulta imprescindible extraer en este mismo *formato* la funcionalidad expresada en la consulta, en este caso por los requisitos SCTL. En el apéndice A se detalló cómo obtener este perfil funcional para los modelos MUS y en este apéndice se explicará como hacerlo para el caso de requisitos SCTL. De esta manera tendremos las ventajas siguientes:

- una vez obtenido el perfil funcional de los requisitos, el proceso de localización es idéntico al que tendríamos partiendo de un componente, es transparente al hecho de que se trata ahora de un requisito SCTL, y
- nos ahorramos el paso de SCTL-MUS, para obtener un grafo y un componente, y después la extracción de la información de las trazas del componente resultante.

B.2 Trazas de evolución de un requisito SCTL

En este apartado explicaremos cómo obtener, a partir de un requisito R, expresado en SCTL, sus trazas completas $TC(R)$ lo que nos dará la suficiente información para enfocar la búsqueda del componente almacenado que mejor se adapte a la funcionalidad requerida.

EJEMPLO B.1. Dado el requisito siguiente:

$$R \equiv ((a \Rightarrow\bigcirc b) \wedge (c \Rightarrow\bigcirc d)) \Rightarrow\bigcirc(e \Rightarrow\bigcirc f)$$

queremos obtener el conjunto de sus trazas de evolución:

$$TC(R) = \{ab, aef, cd, cef\}$$

□

En el algoritmo recursivo B.1 se explica cómo obtener un resultado como el indicado en el ejemplo B.1. Para este algoritmo se parte del requisito R expresado en notación inversa[1] de modo que, en el caso del ejemplo B.1 tendríamos que, como entrada, el requisito vendría dado por:

$$\Rightarrow\bigcirc \wedge \ \Rightarrow\bigcirc a\, b \ \Rightarrow\bigcirc c\, d \ \Rightarrow\bigcirc e\ f$$

Utilizar esta notación nos permite, además de un procesado rápido del requisito, obviar una condición de salida, ya que sólo se generan las llamadas necesarias, es decir, sólo se activa una recursión cuando se detecta un subrequisito que puede ser premisa, consecuencia o parte de un operador lógico de otro requisito que lo contiene.

Es necesario notar que los requisitos R estarán siempre expresados en forma normal positiva, es decir, el operador *not* ($\neg$) sólo aparece precediendo a acciones o eventos.

B.2.1 Operador *a la vez*

Cuando nos encontramos ante la presencia de un operador $\Rightarrow$ ó *a la vez* dentro de un requisito SCTL, éste implica que el comportamiento expresado en la premisa y en la consecuencia son posibles dentro del mismo estado de aplicabilidad. Esto se traduce en que la secuencia de eventos expresados en la premisa, y en la secuencia de eventos especificados en la consecuencia, son dos subtrazas que:

– en caso de que $TC(R)$ esté vacío todavía, serán incorporadas directamente como dos trazas independientes que son;

[1] Esta notación expresa una sucesión de operaciones de la forma *operador operando1 operando2*

Algoritmo B.1 Obtención de TC ($TC(R)$, requisito R)

– Extraer del requisito R el primer término (puede ser operador u operando).
si (el primer término es un operador temporal) **entonces**
– Obtener, a partir del requisito R, el conjunto de acciones que definen la premisa simple de dicho operador y almacenarlas en la variable acciones_premisa (llamada al algoritmo B.2).
fin de la condición
en caso de que (el primer término)
sea (una acción o evento) **entonces**
– Añadir esta acción al resultado $TC(R)$ (algoritmo B.3).

sea (un operador temporal) **entonces**
en caso de que (el tipo del operador)
sea (operador *a la vez*) **entonces**
– Procesar el subrequisito generado por este operador. Llamada a la función que procesa el operador *a la vez* (algoritmo B.4).

sea (operador *después*) **entonces**
– Procesar el subrequisito generado por este operador. Llamada a la función que procesa el operador *después* (algoritmo B.5).

sea (operador *antes*) **entonces**
– Procesar el subrequisito generado por este operador. Llamada a la función que procesa el operador *antes* (algoritmo B.6).

fin (en caso de que)

sea (el operador lógico *and* u *or*) **entonces**
– Procesar el primer operando del operador lógico, obteniendo el resultado de aplicarle a dicho operando la función TC. Para ello se realiza una llamada recursiva a este mismo algoritmo.
– Procesar el segundo operando del operador lógico, obteniendo el resultado de aplicarle a dicho operando la función TC. Para ello se realiza una llamada recursiva a este mismo algoritmo.
– Se realiza una concatenación de los resultados de los dos pasos previos de forma que así se obtiene el resultado actual de $TC(R)$.

sea (el operador lógico *not*) **entonces**
– Procesar el subrequisito generado por este operador (siempre es una acción ya que el requisito R está siempre expresado en forma normal positiva).
– Concatenar al inicio de los eventos obtenidos el operador *not*.

fin (en caso de que)

Algoritmo B.2 Obtención del conjunto de premisas simples de un requisito R

en caso de que (el primer término del requisito R)
sea (una acción) **entonces**
– Añadirla al conjunto de acciones involucradas en la premisa simple de R.

sea (un operador temporal o bien el operador *not*) **entonces**
– Extraer el subrequisito que conforma la premisa de R y realizar una llamada recursiva a este mismo algoritmo pero teniendo como parámetro la premisa de R.

sea (un operador lógico (*and* u *or*)) **entonces**
– Extraer el subrequisito que conforma el primer operando de R y realizar una llamada recursiva a este mismo algoritmo pero teniendo como parámetro este primer operando.
– Extraer el subrequisito que conforma el segundo operando de R y realizar una llamada recursiva a este mismo algoritmo pero teniendo como parámetro este segundo operando.

fin (en caso de que)
– Devolver como resultado el conjunto de acciones recopiladas.

Algoritmo B.3 Procesa una acción o evento dentro del resultado $TC(R)$

si ($TC(R)$ está vacío) **entonces**
– Añade este evento posible a $TC(R)$.
fin de la condición
para todo (traza almacenada en $TC(R)$) **entonces**
– Añade este evento posible al final de la traza.
fin de la iteración

Algoritmo B.4 Procesado del subrequisito generado por un operador *a la vez* ($TC(R)$, requisito R)

– Procesar la premisa del requisito R, para ello se realiza una llamada al algoritmo B.1.
– Procesar la consecuencia del requisito R, obteniendo el resultado de aplicarle a dicha consecuencia la función TC, pero sólo a la consecuencia. Para ello se realiza una llamada al algoritmo B.1.
– Concatenar el resultado actual de $TC(R)$ con el resultado de aplicar TC(consecuencia de R), obtenida en el paso anterior.

Algoritmo B.5 Procesado del subrequisito generado por un operador *después* ($TC(R)$, requisito R, acciones_premisa)

– Procesar la premisa del requisito R, para ello se realiza una llamada al algoritmo B.1.
– Seleccionar, partiendo de la información almacenada en la variables acciones_premisa, aquellas trazas del valor actual de $TC(R)$ sobre las que influye el operador *después* (algoritmo B.7), las trazas de aplicabilidad del operador.
– Procesar la consecuencia del requisito R, obteniendo el resultado de aplicarle a dicha consecuencia la función TC, pero sólo a la consecuencia. Para ello se realiza una llamada al algoritmo B.1.
– Añadir a las trazas seleccionadas de la premisa, resultado del algoritmo B.7, las obtenidas tras el cómputo de la consecuencia, resultado del paso anterior. Es necesario eliminar previamente aquellas trazas que terminen en una acción negada.

Algoritmo B.6 Procesado del subrequisito generado por un operador *antes* ($TC(R)$, requisito R)

– Procesar la premisa, de forma separada, del requisito R, para ello se realiza una llamada al algoritmo B.1. De esta forma se obtiene la secuencia de acciones involucradas en ella.
– Procesar la consecuencia, de forma separada, del requisito R, para ello se realiza una llamada al algoritmo B.1. De esta forma se obtiene la secuencia de acciones involucradas en ella.
– Procesar ambas secuencias de eventos, las obtenidas de la premisa y las obtenidas de la consecuencia (llamada al algoritmo B.8).
– Concatenar las trazas obtenidas del punto anterior con el resultado $TC(R)$.

– en caso de que $TC(R)$ no esté vacío, serán incorporadas en los puntos definidos por el operador procesado anteriormente.

B.2.2 Operador *después*

A la hora de procesar un operador $\Rightarrow\bigcirc$ ó *después* es necesario tener en cuenta sobre qué acciones de la premisa se van a añadir los eventos de la consecuencia, es decir, las trazas de aplicabilidad del operador (ver algoritmo B.2).

Una vez obtenida la secuencia de acciones de la premisa sobre las que tendrá efecto la consecuencia, será necesario procesar dicha consecuencia (con una nueva llamada recursiva al algoritmo B.1) y concatenar las listas de secuencias obtenidas de la premisa y consecuencia, sincronizándolas con las acciones involucradas en la resolución de los estados de aplicabilidad.

En el algoritmo B.7 se explica cómo obtener, del conjunto de trazas de $TC(R)$, aquellas sobre las que influye el operador *después*, ya que a ellas habrá que concatenarles las subtrazas definidas por la consecuencia.

Algoritmo B.7 Obtención de las trazas de aplicabilidad del operador *después*, dadas aquellas acciones expresadas en la premisa simple del requisito

para todo (acción perteneciente a la premisa simple) **entonces**
  si (esta acción se encuentra en una traza de $TC(R)$) **entonces**
    – Copiar dicha traza de $TC(R)$.
    – Crear una nueva traza conformada por la original menos la secuencia que existe a partir de esta acción de la premisa simple.
    si (la acción de la premisa está negada) **entonces**
      – Añadir a la traza anterior la acción negada.
      – Almacenar esta nueva traza en TC(consecuencia de R).
      – Añadir a la traza anterior la acción subespecificada a_{sub}.
    fin de la condición
    – Almacenar esta nueva traza en TC(consecuencia de R).
  fin de la condición
fin de la iteración
– Devolver como resultado TC(consecuencia de R).

B.2.3 Operador *antes*

Para procesar un operador $\Rightarrow\bigodot$ ó *antes*, es necesario tener en cuenta que la secuencia de eventos especificada en la consecuencia ha de ser posible en un estado anterior a aquél donde se satisfará la secuencia de eventos especificada por la premisa. En este punto pueden tomarse básicamente dos alternativas:

– añadir directamente la secuencia de la premisa a continuación de todas y cada una de las secuencias especificadas en la consecuencia, o bien

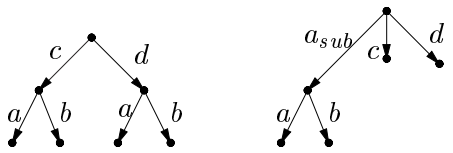
– no realizar la suposición de que la transición desde el estado anterior se va a producir sólo por las secuencias especificadas en la premisa, sino que ésta puede producirse por un evento todavía no especificado.

Nosotros hemos optado por la segunda solución ya que permite una mayor flexibilidad en la búsqueda. Al disponer de una transición cuyo evento está por especificar, es posible abarcar un mayor número de componentes reutilizables que satisfagan la propiedad expresada. En el ejemplo B.2 puede verse un caso concreto.

EJEMPLO B.2. Dado el requisito R:

$$(a \ \Rightarrow \ b) \ \Rightarrow\!\!\bigodot (c \ \Rightarrow \ d)$$

éste se satisface tanto por el grafo de la figura B.1(a) como por el grafo de la figura B.1(b). Pero el segundo grafo (figura B.1(b)) permite una mayor flexibilidad en las búsquedas.



(a) Grafo 1 (b) Grafo 2

Figura B.1. Ejemplo entre dos grafos MUS que satisfacen el requisito funcional R

□

Algoritmo B.8 Procesado de las trazas de la premisa y de las trazas de la consecuencia enlazadas por el operador *antes* ($TC(R)$, TC(premisa de R), TC(consecuencia de R))

– Añadir las secuencias obtenidas tras el procesado de la consecuencia al resultado $TC(R)$.
si (existe una acción a_{sub} en TC(premisa de R)) **entonces**
  – Añadir al comienzo de cada subtraza la acción subespecificada a_{sub}.
  – Almacenar la traza resultante en el resultado $TC(R)$.
en otro caso
  – Añadir las secuencias obtenidas tras el procesado de la premisa en el resultado $TC(R)$.
fin de la condición

B.3 Consideraciones prácticas

Especificación de un conjunto de requisitos SCTL

Todos los algoritmos detallados en este apéndice obtienen la expresión de la función TC aplicada a un único requisito, pero ¿qué ocurre si la funcionalidad expresada por el usuario se encuentra especificada en más de una fórmula lógica SCTL? Un ejemplo sencillo puede ser el siguiente:

$$R_1 : (a \ \Rightarrow \ b) \ \Rightarrow\bigcirc \ (d \ \Rightarrow \ e)$$
$$R_2 : c \ \Rightarrow\bigcirc \ (f \ \Rightarrow \ g)$$

En una situación como ésta el requisito global que expresa la funcionalidad pedida puede expresarse como la composición de todos los requisitos individuales relacionados por el operador lógico *and*, así en este caso se tendrá que

$$R = R_1 \wedge R_2$$

que sería equivalente a extraer las trazas posibles de R_1 a continuación las de R_2 y concatenar ambos conjuntos que, en este caso, resultaría:

$$TC(R) = \{ad, ae, b, cf, cg\}$$

En esta situación uno de los posibles grafos MUS que se podrían sintetizar sería el de la figura B.2.

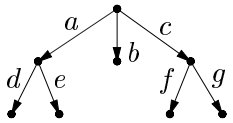


Figura B.2. Ejemplo de grafo MUS de un conjunto de requisitos

Recursividad en los requisitos SCTL

A lo largo de este apéndice se ha tratado de obtener la función TC aplicada a un requisito R de forma que se disponga de su conjunto de trazas completas. Sin embargo no se trata la posibilidad de que este requisito sea recursivo y que tenga bucles de funcionalidad, como por ejemplo el siguiente:

$$R_1 : a \ \Rightarrow\bigcirc \ (b \ \Rightarrow\bigcirc \ c)$$
$$R_2 : e \ \Rightarrow\bigcirc \ R_{21}$$
$$R_{21} : f \ \Rightarrow\bigcirc \ (g \ \Rightarrow\bigcirc \ R_{21})$$

Una de las representaciones MUS de este conjunto de requisitos es la indicada en la figura B.3.

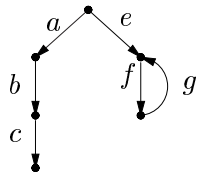


Figura B.3. Ejemplo de grafo MUS de un requisito recursivo

En este caso sería necesario obtener el conjunto de trazas completas no acotadas, TC^{∞} aplicado a R. Para la detección del bucle de funcionalidad es preciso añadir en el algoritmo B.1 la posibilidad de incluir identificadores de requisitos SCTL, como es el caso de R_{21}, de forma que en este caso se localice, dentro de la base de datos de requisitos existentes, aquél que tenga a R_{21} como identificador, obtener sus trazas no acotadas y proseguir con el algoritmo. De esta forma se podrían identificar los bucles de comportamiento y obtener el resultado:

$$TC^{\infty}(R) = \{abc, e(fg)+\}$$

Obtención de los patrones NE y NE^{∞}

Una vez obtenido el patrón $TC(R)$ según el algoritmo B.1 y una vez obtenido el patrón $TC^{\infty}(R)$ según el manejo de la recursividad definido en el apartado anterior, es posible obtener los patrones $NE(R)$ y $NE^{\infty}(R)$. Para ello se procederá tal y como se detalló en el apéndice A donde, a partir de los patrones disponibles, se extraen los otros dos sin necesidad de procesar de nuevo los requisitos. Así que se aplicará el algoritmo A.2 para obtener $NE^{\infty}(R)$ a partir de $TC^{\infty}(R)$, y el algoritmo A.4 para obtener $NE(R)$ a partir de $NE^{\infty}(R)$.

B.4 Ejemplo de aplicación del algoritmo

En este apartado veremos un ejemplo de obtención de las trazas de evolución de un requisito a partir de su expresión SCTL:

$$R_1 \equiv (a \wedge b) \Rightarrow \bigcirc\, c$$

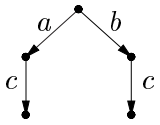


Figura B.4. Grafo característico del requisito R_1

Su grafo característico será el representado en la figura B.4, de donde es inmediato concluir que su conjunto de trazas será el siguiente:

$$TC(R_1) = \{ac, bc\}$$

EJEMPLO B.3. El requisito R_1 expresado en notación inversa

$$\Rightarrow\bigcirc \ \wedge \ a\ b\ c$$

será uno de los parámetros de entrada al algoritmo B.1. El otro será un conjunto de trazas que llamaremos $TC(R_1) = \emptyset$, y que estará vacío en un principio. A continuación veremos cómo se producen cada uno de los pasos en el algoritmo hasta extraer la información deseada:

- **Recursión 1**: ($R_1 = \{ \Rightarrow\bigcirc \ \wedge \ a\ b\ c\}, \quad TC(R_1) = \emptyset$)
  Está procesando un operador, así que lo primero es extraer el conjunto de acciones involucradas en la premisa, acciones_premisa = $\{a, b\}$. Dado que estamos procesando un operador, es necesario realizar una llamada recursiva para procesar su premisa, lo que implica un salto a la recursión 2, que devuelve como resultado $TC(R_1) = \{a, b\}$.
  Llegados a este punto, se ha procesado la premisa del requisito R_1 ($a \ \wedge \ b$), ahora falta por procesar la consecuencia, llamada a la recursión 5, cuyo resultado es TC(consecuencia de R_1)= $\{c\}$.
  Para enlazar las trazas de la premisa con las trazas de la consecuencia es necesario tener en cuenta las acciones involucradas en la premisa, acciones_premisa = $\{a, b\}$, que permiten obtener las trazas de aplicabilidad del operador sobre la premisa. Para ello se llama al algoritmo B.7.

  El resultado final, entonces, será $TC(R_1) = \{ac, bc\}$.

- **Recursión 2**: ($R_1 = \{ \wedge\ a\ b\ c\}, \quad TC(R_1) = \emptyset$)
  En esta iteración se está procesando el operador lógico $\wedge$, con lo que será necesario procesar el primer subrequisito generado por dicho operador (operando a) con una llamada recursiva, recursión 3, cuyo resultado será $TC(R_1) = \{a\}$.
  Después habrá que procesar el segundo subrequisito generado por el operador lógico $\wedge$ (operando b en este caso) con una llamada recursiva,

recursión 4, cuyo resultado será $TC(R_1) = \{b\}$.
Una vez procesados ambos subrequisitos, será necesario concatenarlos según este operador, de ahí que el resultado obtenido sea $TC(R_1) = \{a, b\}$.

➤ **Recursión 3**: ($R_1 = \{a\ b\ c\}, \quad TC(R_1) = \emptyset$)
Está procesando una acción, y tiene una lista vacía como entrada, así que simplemente incluye esa acción en dicha lista, con lo que se modifica el valor de $TC(R_1) = \{a\}$.

➤ **Recursión 4**: ($R_1 = \{b\ c\}, \quad TC(R_1) = \emptyset$)
Está procesando una acción, y tiene una lista vacía como entrada, así que simplemente incluye esa acción en dicha lista, con lo que se modifica el valor de $TC(R_1) = \{b\}$.

➤ **Recursión 5**: ($R_1 = \{c\}, \quad TC(R_1) = \emptyset$)
Está procesando una acción, y tiene una lista vacía como entrada, así que simplemente incluye esa acción en dicha lista, con lo que se modifica el valor de $TC(R_1) = \{c\}$.

Bibliografía

(1993). *WISR 6: 6th Workshop on Institutionalizing Software Reuse*, New York. ftp://gandalf.umcs.maine.edu/pub/WISR/wisr6/.

(1995). *WISR 7: 7th Workshop on Institutionalizing Software Reuse*, Illinois. ftp://gandalf.umcs.maine.edu/pub/WISR/wisr7/.

(1997). *WISR 8: 8th Workshop on Institutionalizing Software Reuse*, Ohio. ftp://gandalf.umcs.maine.edu/pub/WISR/wisr8/.

(1999). *WISR 9: 9th Workshop on Institutionalizing Software Reuse*, Texas. ftp://gandalf.umcs.maine.edu/pub/WISR/wisr9/.

Addy, E. A. (1998). A Framework for Performing Verification and Validation in Reuse-Based Software Engineering. *Annals of Software Engineering*, 5:279–292.

Akers, R. L., Kant, E., Randall, C. J., Steinberg, S. y Young, R. L. (1997). SciNapse: a Problem-Solving Environment for Partial Differential Equations. *IEEE Computational Science and Engineering*, 4(3):32–42.

Arnold, R. y Frakes, W. (1992). Software Reuse and Reengineering. *CASE Trends*, 4(2):44–48.

Atkinson, S. (1995). A Unifying Model for Retrieval from Reusable Software Libraries. Technical Report 95-41, Software Verification Research Centre, Dept. of Computer Science, University of Queensland.

Atkinson, S. (1996). A Formal Model for Integrated Retrieval from Software Libraries. *Technology of Object-Oriented Languages and Systems: TOOLS 21*, páginas 153–167. Monash University Printing Services, Caulfield, Victoria, Australia.

Atkinson, S. (1997). Examining Behavioural Retrieval. En *WISR 8: 8th Workshop on Institutionalizing Software Reuse*.

Barnes, B. H. y Bollinger, T. B. (1991). Making Reuse Cost-effective. *IEEE Software*, 8(1):13–24.

Basili, V. R., Briand, L. C. y Melo, W. L. (1996). Assessing the Impact of Reuse on Quality and Productivity in Object-Oriented Systems. *Communication of ACM*.

Batory, D. y Gerardi, B. J. (1996). Validating Component Composition in Software System Generators. En *International Conference on Software Reuse*, páginas 72–81, Orlando, Florida.

Batory, D. y O'Malley, S. (1992). The Design and Implementation of Hierarchical Software Systems with Reusable Components. *ACM Transactions on Software Engineering and Methodology*, 1(4):355–398.

Biggerstaff, T. J. (1998). A Perspective of Generative Reuse. *Annals of Software Engineering*, 5:169–226.

Biggerstaff, T. J. y Richter, C., editores (1989). *Software Reusability: Concepts and Models*, volumen 1 de *Frontier Series*. ACM press.

Boehm, B. W. (1988). A Spiral Model of Software Development and Enhancement. *IEEE Computer*, 21(5):61–72.

Bolognesi, T. y Briksma, E. (1989). *The Formal Description Technique LOTOS*, capítulo Introduction to the ISO Specification Language LOTOS. North-Holland.

Boyer, R. S. y Moore, J. S. (1979). *A Computational Logic*. Academic Press, New York.

Brown, A. W., editor (1996). *Component-Based Software Engineering: Selected Papers from the Software Engineering Institute*. IEEE Computer Society Press.

Browne, S., Dongarra, J., Hohn, K. y Nielsen, T. (1996). Software Repository Interoperability. Technical Report CS-96-329, University of Tennessee.

Bryant, R. E. (1986). Graph-Based Algorithms for Boolean Function Manipulation. *IEEE Transactions on Computers*, 35(8):677–691.

Caldiera, G. y Basili, V. R. (1991). Identifying and Qualifying Reusable Software Components. *IEEE Computer*, 24(2):61–70.

Chang, C.-L. y Lee, R. C.-T. (1973). *Symbolic Logic and Mechanical Theorem Proving*. Academic Press.

Cheatham, T. E. (1989). Reusability through Program Transformations. En Biggerstaff, T. J. y Richter, C., editores, *Software Reusability: Concepts and Models*, volumen 1 de *Frontier Series*, páginas 321–335. ACM press.

Chen, Y. y Cheng, B. H. C. (1997). Facilitating an Automated Approach to Architecture-based Software Reuse. En *Proceedings of the 12th International Automated Software Engineering Conference*, páginas 238–245. IEEE Computer Society Press.

Cheng, B. H. C. y Jeng, J. J. (1995). Specification Matching for Software Reuse: A Foundation. En *Symposium on Software Reuse*, páginas 97–105. ACM SIGSOFT, ACM Press.

Cheng, T. T., Lock, E. D. y Prymes, S. (1991). Use of Very High Level Languages and Program Generation by Management Professionals. *IEEE Transactions on Software Engineering*, 10(5):552–563.

Clarke, E. y Emerson, E. A. (1981). Synthesis of Synchronization Skeletons for Branching Time Temporal Logic. *Lecture Notes in Computer Science*, 131:52–71. Springer Verlag.

Cleaveland, R., Parrow, J. y Steffen, B. (1993). The Concurrency Workbench: A Semantics-Based Tool for the Verification of Concurrent Systems. *ACM Transactions on Programming Languages and Systems*, 15(1):36–72.

Clements, P., Bass, L., Kazman, R. y Abowd, G. (1996). Predicting Software Quality by Architecture-Level Evaluation. En Brown, A. W., editor, *Component-Based Software Engineering: Selected Papers from the Software Engineering Institute*, páginas 19–25. IEEE Computer Society Press.

Davis, T. (1993). The Reuse Capability Model: A Basis for Improving an Organization's Reuse Capability. En *Advances in Software Reuse, Selected Papers from the Second Int'l Workshop on Software Reusability Advances in Software Reuse*, páginas 126–133, Lucca, Italy.

Díaz-Redondo, R. P. y Pazos-Arias, J. J. (2001). Reuse of Verification Efforts and Incomplete Specifications in a Formalized, Iterative and Incremental Software Process. En *Proceedings of International Conference on Software Engineering (ICSE) Doctoral Symposium*, Toronto, Ontario (Canadá).

Dietz, P., Weigert, T. y Weil, F. (1998). Formal Techniques for Automatically Generating Marshalling Code from High-Level Specifications. En *Workshop on Industrial-Strength Formal Specification Techniques*, Boca Ratón, Florida.

Dorfman, M. y Thayer, R. H., editores (1997). *Software Engineering*. IEEE Computer Society Press.

Dubinsky, E., Freudenberger, S., Schonberg, E. y Schwartz, J. T. (1989). Reusability of Design for Large Software Systems: An Experiment with the SETL Optimizer. En Biggerstaff, T. J. y Richter, C., editores, *Software Reusability: Concepts and Models*, volumen 1 de *Frontier Series*, páginas 275–293. ACM press.

Dunn, M. F. y Knight, J. C. (1993). Certification of Reusable Software Parts. Technical Report CS-93-41, University of Virginia.

Elseaidy, W., Cleaveland, R. y Baugh, J. (1997). Modeling and Verifying Active Structural Control Systems. *Science of Computer Programming*, 29(1):99–122.

Eman, K. E., Drouin, J.-N. y Melo, W. (1997). *SPICE: The Theory and Practices of Software Improvement and Capability Determination*. IEEE Computer Society Press.

ESA (1991). *ESA Software Engineering Standards ESA PSS-05-0, Issue2*. ESA Board of Software Standardisation and Control, European Space Agency, Paris.

Everitt, B. (1986). *Cluster Analysis*. Gower Publishing Company (SSRC).

Favaro, J. (1999). Strategic Analysis of Component-Based Development. En *WISR 9: 9th Workshop on Institutionalizing Software Reuse*.

Feather, M. S. (1989). Reuse in the Context of a Transformation-Based Methodology. En Biggerstaff, T. J. y Richter, C., editores, *Software Reusability: Concepts and Models*, volumen 1 de *Frontier Series*, páginas 337–359. ACM press.

Fenton, N. E. y Pfleeger, S. L. (1997). *Software Metrics: A rigorous and practical approach*. PWS Publishing Company.

Fernández, J.-C., Garavel, H., Kerbrat, A., Mateescu, R. y Mounier, L. (1996). CADP (CAESAR/ALDEBARAN development package): A protocol validation and verification toolbox. *Lecture Notes in Computer Science*, (1102).

Fernández-Vilas, A. (2002). *Tratamiento Formal de Sistemas con Requisitos de Tiempo Real Críticos*. PhD tesis, Departamento de Enxeñería Telemática - Universidade de Vigo.

Fischer, B. (1998). Specification-Based Browsing of Software Component Libraries. En *Proceedings of the 13th International Automated Software Engineering Conference*, páginas 74–83. IEEE Computer Society Press.

Fischer, B., Kievernagel, M. y Struckmann, W. (1995). VCR: A VDM-Based Software Component Retrieval Tool. En *Proceedings of the 17th ICSE Workshop on Formal Methods Application in Software Engineering Practice*, Seatle, EE.UU.

Frakes, W., Prieto-Díaz, R. y Fox, C. (1998). DARE: Domain Analysis and Reuse Environment. *Annals of Software Engineering*, 5:125–141.

Franch, X. (1998). Systematic Formulation of Non-Functional Characteristics of Software. En *3rd IEEE International Conference on Requirements Engineering (ICRE)*, páginas 174–181, Colorado Springs.

Franch, X. y Botella, P. (1998). Putting non-Functional Requirements into Software Architecture. En *IEEE 9th International Workshop on Software Specification and Design (IWSSD)*, páginas 60–67.

Frants, V., Shapiro, J. y Voiskunskii, V. G. (1997). *Automated Information Retrieval*. Library and Information Science Series. Academic Press.

García-Duque, J. (2000). *Especificación, Verificación y Mantenimiento de Requisitos Funcionales con Técnicas de Descripción Formal*. PhD tesis, Departamento de Tecnoloxías das Comunicacións - Universidade de Vigo.

García-Duque, J. y Pazos-Arias, J. J. (2001). Reasoning over Inconsistent Viewpoints: How Level of Agreement can Evolve? En *2nd. International Workshop on Living with Inconsistency. 23th. International Conference on Software Engineering (ICSE)*, Toronto (Canadá).

Gil-Solla, A. (2000). *Diseño y Verificación de Sistemas Distribuidos mediante la Aplicación Combinada de Métodos Formales*. PhD tesis, Departamento de Tecnoloxías das Comunicacións - Universidade de Vigo.

Girardi, M. R. y Ibrahim, B. (1994). Automatic Indexing of Software Artifacts. En *3rd International Conference on Software Reusability*, páginas 24–32, Rio de Janeiro, Brazil.

Gordon, M. (1988). *VLSI Specification, Verification and Synthesis*, capítulo HOL: A Proof Generating System for Higher-Order Logic, páginas 73–128. Kluwer.

Gordon, M. J., Milner, A. J. y Wadsworth, C. P. (1979). Edinburgh LCF: A Mechanised Logic of Computation. *Lecture Notes in Computer Science*, 78. Springer Verlag.

Humphrey, W. (1989). *Managing the Software Process*. Addison-Wesley.

IABG (1992). *The V-Model – General Directive 250, Software Development Standard for the German Federal Armed Forces*. Available from IndustrianlagenBetriebsgesellschaft mbH-IABG, Einsteinstr. 20, D-85521 Ottobrum, Germany.

ISO (1989). *Information Processing Systems – Open Systems Interconnection – LOTOS – A Formal Description Technique Based on an Extended State Transition Model*. ISO/IEC/8807, International Standards Organization.

ISO (1991). *Information technology - Software product evaluation - Quality characteristics and guidelines for their use*. ISO/IEC 9126, International Standards Organization.

ISO (1991, 1994). *Quality Management and Quality Assessment Standards, Part 3: Guidelines for the ISO 9001 to the Development, Supply and Maintenance of Software*. ISO 9000-3, International Standards Organization.

ISO (1995). *Information Technology – Software Life Cycle Processes*. ISO/IEC 12207, International Standards Organization.

ISO (1997). *Draft Technical Report (DTR) Draft Standard for Software Process Assessment (Parts 1–9)*. ISO/IEC 15504, International Standards Organization .

Jeng, J. J. y Cheng, B. H. C. (1993). Using Formal Methods to Construct a Software Component Library. En Sommerville, I. y Paul, M., editores, *Proceedings of 4th Software Engineering Conference in Computer Science*, volumen 717, páginas 397–417, Garmisch, Parkenkirchen. Springer Verlag.

Jilani, L. L. (1997). Retrieving Software Components that Minimize Adaptation Effort. En *Proceedings of 12th Automated Software Engineering Conference*, páginas 255–262.

Jr., E. M. C., Grumberg, O. y Peled, D. A. (2000). *Model Checking*. The MIT Press, Cambridge, Massachusetts.

Kang, K., Cohen, S., Hess, J., Novak, W. y Peterson, S. (1990). Feature-Oriented Domain Analysis (FODA) Feasibility Study. Technical Report CMU/SEI-90-TR-21, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA.

Kang, K. C., Kim, S., Lee, J., Kim, K., Shin, E. y Huh, M. (1998). FORM: A Feature-Oriented Reuse Method with Domain-Specific Reference Architectures. *Annals of Software Engineering*, 5:143–168.

Karlsson, E. A. (1996). *Software Reuse: A Holistic Approach*. Wiley Series in Software based Systems. John Wiley and sons.

Kaufmann, M. y Moore, J. S. (1995). *ACL2: A Computational Logic for Applicative Common Lisp. The User's Manual (Version 1.8)*. ftp://ftp.cli.com/pub/acl2/v1-8/acl2-sources/doc/HTML/acl2-doc.html.

Keidar, I., Khazan, R., Lynch, N. y Shvartsman, A. (2000). An Inheritance-Based Technique for Building Simulation Proofs Incrementally. En *22nd International Conference on Software Engineering (ICSE)*, páginas 478–487, Limerik, Ireland.

Knight, J. C. y Dunn, M. F. (1998). Software Quality through Domain-Driven Certification. *Annals of Software Engineering*, 5:293–315.

Krueger, C. W. (1992). Software Reuse. *ACM Computing Surveys*, 24(2):131–183.

Kurshan, R. P. (1994). The Complexity of Verification. En *Proceedings of 26th ACM Symposium on Theory of Computing (STOC)*, páginas 365–371, Montreal (Canadá).

Kuvaja, P. y Bicego, A. (1994). BOOTSTRAP: a European Assessment Methodology. *Software Quality Journal*, 3:112–127.

Lam, W. (1998). A Case-Study of Requirements Reuse through Product Families. *Annals of Software Engineering*, 5:253–277.

Lim, W. C. (1999). Why the Reuse Percent Metric should never be used alone. En *WISR 9: 9th Workshop on Institutionalizing Software Reuse*.

Liu, C. L. (1985). *Element of Discrete Mathematics*. Computer Science Series. McGraw-Hill, segunda edición.

Maarek, Y. S., Berry, D. M. y Kaiser, G. E. (1991). An Information Retrieval Approach for Automatically Constructing Software Libraries. *IEEE Transactions on Software Engineering*, 17(8):800–813.

Mason, T. y Brown, D. (1991). *Lex & Yacc*. O'Reilly and Associates, Inc.

Mehlhorn, K. y Maher, S. (1999). *The LEDA Platform of Combinatorial and Geometric Computing*. Cambridge University Press.

Mili, A., Mili, R. y Mittermeir, R. (1998). A Survey of Software Reuse Libraries. *Annals of Software Engineering*, 5:349–414.

Mili, H., Mili, F. y Mili, A. (1995). Reusing Software: Issues and Research Directions. *IEEE Transactions on Software Engineering*, 21(6):528–562.

Mili, R., Mili, A. y Mittermeir, R. T. (1997). Storing and Retrieving Software Components: A Refinement Based System. *IEEE Transactions on Software Engineering*, 23(7):445–460.

N. Bjorner et al. (1996). STeP: Deductive-algorithmic Verification of Reactive and Real-time Systems. En *Proceedings of the 8th International Conference on Computer-Aided Verification*, número 1102 de Lecture Notes in Computer Science, páginas 415–418. Springer Verlag.

Neighbors, J. (1981). *Software Construction Using Components*. PhD tesis, Department of Information and Computer Science, University of California.

Neighbors, J. (1984). The Draco Approach to Constructing Software from Reusable Components. *IEEE Transactions on Software Engineering*, 10(5):564–574.

Neighbors, J. (1992). The Evolution from Software Components to Domain Analysis. *International Journal of Software Engineering and Knowledge Engineering*, 2(3):325–354.

Neighbors, J. (1996). Finding Reusable Software Components in Large Systems. En *Third Working Conference on Reverse Engineering (WCRE)*, páginas 2–10. IEEE Computer Society Press.

Nestor-Ribeiro, A. y Mário-Martins, F. (1995). A Fuzzy Query Language for a Software Reuse Environment. En *WISR 7: 7th Workshop on Institutionalizing Software Reuse*.

Ouyang, Y. y Carver, D. L. (1997). Creation of Reusable Components Based on Formal Methods. En *WISR 8: 8th Workshop on Institutionalizing Software Reuse*.

Owre, S., Rushby, J. y Shankar, N. (1992). PVS: A Prototype Verification System. En Kapur, D., editor, *11th International Conference on Automated Deduction (CADE)*, volumen 607 de Lecture Notes in Artificial Intelligence, páginas 748–752. Springer Verlag.

Paulk, M., Weber, C., Curtis, B. y Chrissis, M. (1994). *The Capability Maturity Model: Guidelines for Improving the Software Process*. Series in Software Engineering. Addison-Wesley.

Paulk, M. C., Curtis, B., Chrissis, M. B. y Weber, C. V. (1997). The Capability Maturity Model for Software. En Dorfman, M. y Thayer, R. H., editores, *Software Engineering*, páginas 427–438. IEEE Computer Society Press.

Pazos-Arias, J. J. (1995). *Transformación y Verificación con LOTOS*. PhD tesis, Departamento de Ingeniería de Sistemas Informáticos - Universidad Politécnica de Madrid.

Pazos-Arias, J. J. y García-Duque, J. (2001). SCTL-MUS: A Formal Methodology for Software Development of Distributed Systems. A Case Study. *Formal Aspects of Computing*, 13:50–91.

Penix, J. (1999). REBOUND: A Framework for Automated Component Adaptation. En *WISR 9: 9th Workshop on Institutionalizing Software Reuse*.

Penix, J. y Alexander, P. (1997). Component Reuse and Adaptation at the Specification Level. En *WISR 8: 8th Workshop on Institutionalizing Software Reuse*.

Penix, J. y Alexander, P. (1999). Efficient Specification-Based Component Retrieval. *Automated Software Engineering: An International Journal*, 6(2):139–170.

Popel, B. y Wise, C. (1996). TRILLIUM and the CMM: Differences between the two Models and Assessment Methods. En *Proceedings of SEPG Conference*, páginas 20–23, Atlantic City.

Prieto-Díaz, R. (1985). *A Software Classification Scheme*. PhD tesis, University of California, Irvine.

Prieto-Díaz, R. (1990). Domain Analysis: An Introduction. En *ACM SIGSOFT Software Engineering Notes*, volumen 2, páginas 47–54.

Prieto-Díaz, R. (1991). Implementing Faceted Classification for Software Reuse. *Communications of the ACM*, 34(5):88–97.

Prieto-Díaz, R. (1993). Status Report: Software Reusability. *IEEE Software*, 10(3):61–66.

Prieto-Díaz, R. (1996). Reuse as a New Paradigm for Software Development. En Sarshar, M., editor, *Systematic Reuse: Issues in Initiating and Improving a Reuse Program*, London. Springer Verlag.

Prieto-Díaz, R. y Freeman, P. (1987). Classifying Software for Reusability. *IEEE Software*, 4(1):6–16.

Queille, J. y Sifakis, J. (1982). Specification and Verification of Concurrent Systems in CAESAR. En *Proceedings of Fifth International Symposium on Programming*, volumen 137 de Lecture Notes in Computer Science, páginas 337–351.

R. Constable et al. (1986). *Implementing Mathematics with the NuPRL Proof Development Environment*. Prentice-Hall.

Roscoe, A. (1994). *A Classical Mind: Essays in Honour of C.A.R. Hoare*, capítulo Model-checking CSP, páginas 353–378. Prentice-Hall.

Ross, K. A. y Wright, C. R. B. (1992). *Discrete Mathematics*. Prentice-Hall International Editions.

Sametinger, J. (1997). *Software Engineering with Reusable Components*. Springer Verlag.

Schumann, J. y Fischer (1997). NORA/HAMMR: Making Deduction-Based Software Component Retrieval Practical. En Lowry, M. y Ledru, Y., editores, *Proceedings of the 12th International Conference Automated Software Engineering*, páginas 246–254. IEEE Computer Society Press.

SER (1996). *Software Evolution and Reuse*. ESPRIT Project 9809.

Simos, M., Creps, R., Klingler, C. y Lavine, L. (1995). Organization Domain Modeling (ODM) Guidebook, version 1.0. Technical Report STARS-VC-A023/011/00, Unisys STARTS.

Simos, M. A. (1999). Domain Envisioning: A Lightweight, Incremental Approach to Getting a Company Started with Systematic Reuse. En *WISR 9: 9th Workshop on Institutionalizing Software Reuse*.

Sitaraman, M., editor (1996). *Fourth International Conference on Software Reuse*, Orlando, Florida. IEEE Computer Society Press.

Smaragdakis, Y. y Batory, D. (2000). Application Generators. En *Encyclopedia of Electrical and Electronics Engineering, Supplement 1*.

Stroustrup, B. (1996). Language-technical Aspects of Reuse. En Sitaraman, M., editor, *Fourth International Conference on Software Reuse*, páginas 11–19, Orlando, Florida. IEEE Computer Society Press.

Tanenbaum, A. S. (1997). *Redes de Ordenadores*. Prentice Hall.

Trillium (1994). *TRILLIUM: Model for Telecom Product Development and Support Process Capability, Release 3.0*. Bell Canada Adquisitions, Canada.

van Glabbeek, R. J. (1990). The Linear Time - Branching Time Spectrum. En Baeten, J. y Klop, J., editores, *Proceedings of CONCUR '90: Theories of Concurrency–Unification and Extension*, volumen 458 de Lecture Notes in Computer Science, páginas 278–297. Springer Verlag.

Weide, B. W. (1999). Modular Regression Testing: Connections to Component-Based Software. En *WISR 9: 9th Workshop on Institutionalizing Software Reuse*.

Weide, B. W. y Hollingsworth, J. E. (1994). On Local Certificability of Software Components. Technical Report OSU-CISRC-1/94-TR04, Departament of Computer and Information Sciences, Ohio State University, Columbus, OH.

Zahran, S. (1998). *Software Process Improvement: Practical Guidelines for Bussines Success*. SEI Series in Software Engineering. Addison-Wesley.

Zaremski, A. M. y Wing, J. M. (1997). Specification Matching of Software Components. *ACM Transactions on Software Engineering and Methodology*, 6(4):333–369.

Zave, P. (1991). An Insider's Evaluation of PAISLey. *IEEE Transactions on Software Engineering*, 17(3):211–225.